Path-Sensitive Sparse Analysis without Path Conditions

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

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

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

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

Abstract

Sparse program analysis is fast as it propagates data flow facts via data dependence, skipping unnecessary control flows. However, when path-sensitively checking millions of lines of code, it is still prohibitively expensive because a huge number of path conditions have to be computed and solved via an SMT solver. This paper presents Fusion, a fused approach to inter-procedurally path-sensitive sparse analysis. In Fusion, the SMT solver does not work as a standalone tool on path conditions but directly on the program together with the sparse analysis. Such a fused design allows us to determine the path feasibility without explicitly computing path conditions, not only saving the cost of computing path conditions but also providing an opportunity to enhance the SMT solving algorithm. To the best of our knowledge, Fusion, for the first time, enables whole program bug detection on millions of lines of code in a common personal computer, with the precision of inter-procedural path-sensitivity. Compared to two state-of-the-art tools, Fusion is 10× faster but consumes only 10% of memory on average. Fusion has detected over a hundred bugs in mature open-source software, some of which have even been assigned CVE identifiers due to their security impact.

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

Keywords: Sparse analysis, path sensitivity, program dependence graph, SMT solving.

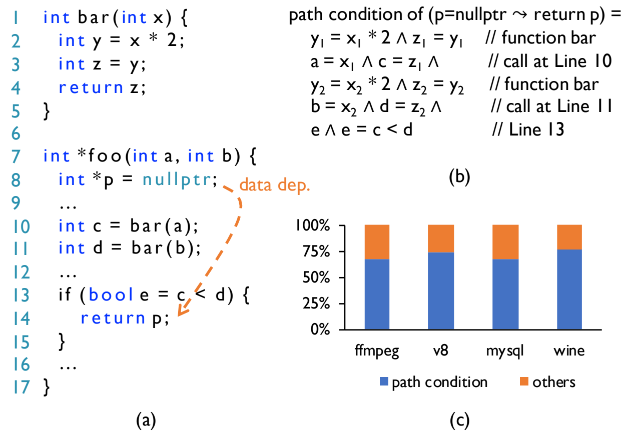
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454086

ACM Reference Format:
Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454086

1 Introduction

Sparse program analysis is often believed to be more scalable than the conventional data flow analysis, because it propagates data flow facts via data dependence, skipping unnecessary control flows [11, 39, 45–49]. However, we observe that this claimed scalability is not apparent and is even negligible when high precision, i.e., inter-procedural path-sensitivity, is required. For instance, Pinpoint, a recent sparse analyzer, achieves inter-procedural path-sensitivity but is still prohibitively expensive. As reported, it may take 5 hours and 150GB of memory to complete a single analysis [46]. Such high cost creates non-negligible obstacles to deployment in practice, especially for small enterprises and individual users.

The core of this scalability limitation, as we observe, is that, although the sparsity allows for skipping unnecessary control flows, a path-sensitive sparse analysis still needs to compute many exponential-sized path conditions due to the incessant function cloning induced by context-sensitivity. For example, in Figure 1(a), the analysis can propagate the null pointer at Line 8 to Line 14 via data dependence, skipping the control flows in between. However, such sparse propagation cannot prevent us from computing and solving the path condition in Figure 1(b), where the return-value condition, $z = y \land y = 2x$, of the function bar is repetitively instantiated at every call site. As the analysis progresses and the null pointer propagates to the caller and upper-level caller functions, this path condition, together with the instantiated return-value condition, will be further instantiated, leading to an explosive growth of the condition size. Even worse, to avoid repetitively analyzing a function, we often cache these path conditions as function summaries, which severely overload the memory and limit the analysis scalability. In



```
1   int bar(int x) {
2     int y = x * 2;
3     int z = y;
4     return z;
5   }
6
7   int *foo(int a, int b) {
8     int *p = nullptr;
9     ...
10    int c = bar(a);
11    int d = bar(b);
12    ...
13    if (bool e = c < d) {
14      return p;
15    }
16    ...
17  }
```

path condition of (p=nullptr ⇝ return p) =

$y_1 = x_1 * 2 \land z_1 = y_1$ // function bar
$a = x_1 \land c = z_1 \land$ // call at Line 10
$y_2 = x_2 * 2 \land z_2 = y_2$ // function bar
$b = x_2 \land d = z_2 \land$ // call at Line 11
$e \land e = c < d$ // Line 13

Figure 1. (a) Code example. (b) Path condition on which the null pointer returns. (c) The memory usage of four projects with millions of lines of code.

our experience, as shown in Figure 1(c), path conditions may consume over 72% of the runtime memory. Given that SMT solving is inherently expensive, the explosive size of path conditions further exacerbates the perceived performance of SMT solving and, in turn, the static analysis as a whole.

To reduce the overhead caused by path conditions without compromising the precision, existing approaches heavily depend on quantifier elimination (QE), formula simplification (FS), or abstract refinement (AR), to reduce the condition size [2, 8, 9, 12, 23, 55]. However, they often introduce extra computational overhead and deteriorate the analysis performance. For instance, given the explosive condition size and the high complexity of QE problems [19, 26], QE-based approaches could give the analysis performance a devastating blow. Similarly, FS-based and AR-based approaches may involve extra SMT solving procedures to simplify the condition [22] or refine the abstraction [2, 3, 31]. Due to the high cost of SMT solving, they often exhibit weak scalability in practice. Our evaluation shows that an analysis with QE, FS, or AR may fail to analyze a program within 12 hours and 100GB of memory.

This paper presents Fusion, a fused design of sparse analysis with the precision of inter-procedural path-sensitivity. As shown in Figure 2, different from the conventional design that computes, solves, and caches many path conditions, in our design, the SMT solver no longer works separately to check the satisfiability of path conditions, but works together with the sparse analysis on its linear-sized program intermediate representation (IR), which is known as the program dependence graph.[1] Our key insight is that program dependence graph and path condition are allotropes, which means

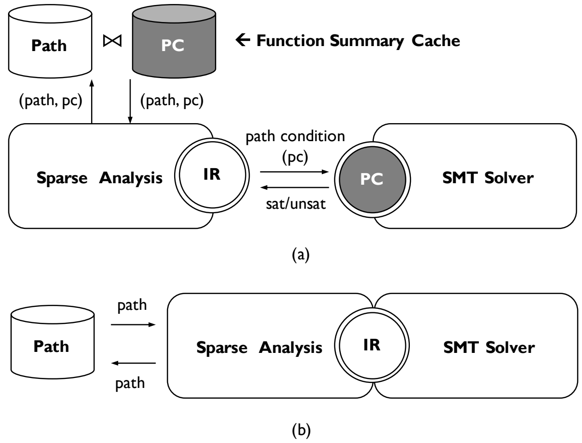


Figure 2. (a) Conventional design: the sparse analysis works on the program dependence graph, computing, solving, and caching path conditions. A path-sensitive summary can be regarded as a path associated with its path condition. (b) Our design: both the sparse analysis and the SMT solver work on the program dependence graph and it is not necessary to compute and summarize the explosive-sized path conditions.

that, although they are in different forms, they encode the same program information, i.e., the control dependence and the data dependence. This fact implies that we can determine path feasibility directly using the IR without explicitly computing and caching a number of explosive-sized path conditions, thus dramatically saving both time and memory. In addition, since our SMT solver works on the program dependence graph, it can also benefit from program information for further acceleration. Such acceleration is not available in a standalone and general-purpose SMT solver, where the program information is lost. We provide more details in § 2.

We have implemented Fusion on top of the LLVM compiler infrastructure [33] and the Z3 SMT solver [20] to detect null exceptions and taint issues in C/C++ programs. The evaluation results demonstrate that Fusion can finish the checking of millions of lines of code within half an hour and 12GB of memory, 2×–48× faster and only 3%–20% of memory compared to existing approaches. The results also imply that, for the first time to the best of our knowledge, Fusion is capable of analyzing millions of lines of code in a common personal computer and simultaneously achieving the precision of inter-procedural path-sensitivity. Fusion has detected over a hundred previously-unknown bugs in mature open-source software, with some even assigned CVE identifiers. In summary, this paper makes three contributions:

- A fused design that enables a scalable inter-procedurally path-sensitive sparse analysis.
- An optimized SMT solving method that works directly on the program dependence graph.
- An extensive experiment that evaluates the scalability and the precision of our approach.

[1] Program dependence graph [25] has many variants, such as the SSA graph [15], the symbolic expression graph [46], and so on. We use the general term, program dependence graph, with no loss of generality.

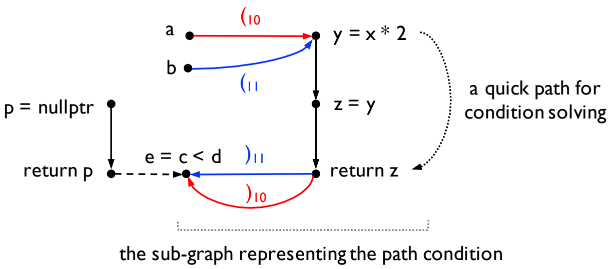


Figure 3. The solid and dashed arrows stand for the data dependence and the control dependence, respectively. Inter-procedural data dependence is labeled by $(_i$ or $)_i$ to represent a call and return pair at Line i.

2 Fusion in a Nutshell

Sparse analysis accepts a program and its program dependence graph as the inputs, and propagates data flow facts via data dependence edges on the graph. Figure 3 shows the program dependence graph of the code in Figure 1. A program dependence graph encodes both the data dependence and the control dependence in a program. We say a statement x is data-dependent on a statement or a variable y if x refers to the value defined by y. We say a statement x is control-dependent on a branch condition y if the truth value of y determines whether the statement x can be executed at runtime. To distinguish different call sites of the same function, the data dependence edges representing call and return are labeled by a unique pair of parentheses.

The Conventional Approach. As discussed before, for the code in Figure 1(a), a conventional approach computes, solves, and caches the path condition in Figure 1(b).

(1) Computing the path condition. Given the size m of the function foo and the size n of the function bar, we can compute the size of the path condition as $O(kn+m)$, where $k = 2$ because the return-value condition of the function bar is instantiated (or cloned) twice. Thus, computing this path condition needs at least $O(kn + m)$ time and space.

(2) Solving the path condition. An SMT solver can solve the path condition via a linear scan with $O(kn + m)$ time and space.[2] That is, since the variables, a and b, are unconstrained,[3] the other variables, e.g., c and d, are also unconstrained due to the equality relations in the formula. Thus, $c < d$ is satisfiable because both c and d are unconstrained.

(3) Caching the path condition. For inter-procedural analysis, we often cache this satisfiable path condition and its corresponding path as a function summary for future instantiation in the caller and up-level caller functions. Thus, we need $O(kn+m)$ space to cache the path condition in memory.

Table 1. The cost of computing, solving, and caching path conditions for the function foo.

	computing	solving	caching
conventional	$O(kn + m)$	$O(kn + m)$	$O(kn + m)$
fusion	$O(n + m)$	$O(n + m)$	-

n: size of bar. m: size of foo. foo calls bar k times.

The Fused Approach. As illustrated in Figure 2, Fusion checks the path feasibility using the program dependence graph, which allows for a more efficient static analysis. As summarized in Table 1, Fusion performs better than the conventional approach in all aspects of computing, solving, and caching the path condition.

(1) Using the slice as the path condition. Since the program dependence graph encodes all of the program dependence relations, it can be directly used, instead of the conventional first-order logic formula, to represent the path condition. As shown in Figure 3, the condition of the path from *p=nullptr* to *return p* is represented as the sub-graph connecting to the path via a control dependence edge. The size of the sub-graph, as well as the time and space for computing the sub-graph, is linear, i.e., $O(n + m)$, much smaller than that of the conventional design. The key here is that the modular structure of the dependence graph allows us to distinguish different call sites without cloning the function *bar.*

(2) Accelerating SMT solving via modularity. The key insight here is that SMT solving shares many techniques with program analysis, e.g., value propagation, which are more efficient using the program dependence graph as it preserves the program structure. In the example, we propagate the data flow fact that "*a* is unconstrained" via the data dependence edges from the vertex *a* to the branch condition *e=c<d*. During the propagation, we can establish a quick path from the vertex *y=2x* to the vertex *return z*. The quick path allows the same propagation from the variable *b* to the branch condition without going through the function *bar*, thus accelerating the propagation. Since the branch condition depends on two unconstrained values, it is satisfiable. Due to the quick path, this solving procedure has a linear complexity $O(n + m)$.

(3) No caching. As shown above, we can check the path feasibility using the program dependence graph. Since the program dependence graph has already been in memory, we do not need to additionally cache any path conditions, thus significantly reducing the time and memory usage compared to the conventional approach.

Summary. The key of our approach is that, instead of being standalone and general-purpose, SMT solvers should work directly on the program dependence graphs to avoid the unnecessary *condition cloning* and *condition caching*, thereby notably improving the scalability of sparse analysis. We formally present our idea in the next section.

[2] While it is simple in the example, SMT solving is more complex in general, which we will detail later.

[3] Intuitively, the variables, a and b, are unconstrained because they are used only once in the path condition.

3 Fusion: the Fused Design

In this section, we first introduce the background of sparse analysis (§ 3.1), with our solution in detail (§ 3.2), followed by a summary of its benefits (§ 3.3).

3.1 Background and Problem Statement

Language. For clarity, we use the simple call-by-value language shown in Figure 4 to model target programs. A program may contain multiple functions, with or without a function body. The semantics of most statements in this language are standard and omitted, except that an identity statement, representing a tautology, is used for simplifying our explanation. Each function parameter must be initialized using the identity statement, e.g., $f(v) = \{v = \langle v \rangle; \cdots\}$. The ite-assignment means that if v_2 is true, v_3 is assigned to v_1. Otherwise, v_4 is assigned to v_1. The return statement and the if-statement assign v_2 to a new variable v_1, which is then used as the return value or the branch condition. In the language, we use $=$ and $\equiv$ to distinguish the assignment operator and the equality operator, and assume that a function only has one return statement as its single exit.

With no loss of generality, we assume the code is in the SSA form [17], where each variable has only one definition and we can merge multiple definitions via a ϕ-assignment, e.g., $v_1 = \cdots;$ **if** $(c = \cdots)$ $\{v_2 = \cdots;\}$ $v_3 = \phi(v_1, v_2)$. In our language, we use $v_3 = \mathit{ite}\,(c, v_2, v_1)$ to replace the ϕ-assignment so that the assignment condition is explicit. Previous work has shown that such replacement is of almost linear complexity [50]. The language abstracts away the pointer operations, because the pointer analysis is not our technical contribution and, in the implementation, we follow the existing work to resolve pointer relations [46]. Following the theory of bounded model checking [4] and many path-sensitive analyzers [2, 7, 46, 48, 49, 53], we assume the code in the language is loop-free as we often unroll loops for a fixed number of times in practice.

Definition 3.1. The program dependence graph of a program in the small language is a triple $G = (V, E_d, E_c)$, where

- V is the vertex set. Each vertex is a statement or, equivalently, the variable defined by the statement.
- $E_d \subseteq V \times V$ is a set of data dependence edges. Each edge is from a statement to the other that refers to the variable defined in the source statement.
- $E_c \subseteq V \times V$ is a set of control dependence edges. Each edge is from a statement to an if-statement — the source statement is reachable at runtime if and only if the if-statement is reachable and the branch condition defined in the if-statement is true.

Given a program in our small language, we can build the control dependence edges E_c in almost linear time [17]. For the data dependence edges E_d, we use the rules in Figure 5 to process each statement. Specially, for a call statement, if it calls a non-empty function, we connect the actual and the formal as well as the return value and its receiver via the data dependence edges, which we refer to as the call edges and the return edges. To distinguish call sites that call the same function, we follow the conventional CFL-reachability method to label each pair of call and return edges with a unique pair of parentheses [42], just as the example in Figure 3. For an empty function, e.g., a third-party library function, we just establish a data dependence relation between the actual and the return value receiver.

$$
\begin{array}{rcll}
\mathit{Program}\ P & := & F+ & \\
\mathit{Function}\ F & := & f(v_1, v_2, \cdots) = \{\ S;\ \} & \\
 & & \mid f(v_1, v_2, \cdots) = \varnothing & \\
\mathit{Statement}\ S & := & v_1 = \langle v_1 \rangle & \textbf{::identity} \\
 & & \mid v_1 = v_2 & \textbf{::assignment} \\
 & & \mid v_1 = v_2 \oplus v_3 & \textbf{::binary} \\
 & & \mid v_1 = \mathit{ite}\,(v_2, v_3, v_4) & \textbf{::if-then-else} \\
 & & \mid v_1 = f(v_2, v_3, \cdots) & \textbf{::call} \\
 & & \mid \textbf{return}\ v_1 = v_2 & \textbf{::return} \\
 & & \mid \textbf{if}\ (v_1 = v_2)\ \{S_1;\} & \textbf{::branching} \\
 & & \mid S_1; S_2 & \textbf{::sequencing}
\end{array}
$$

$$\oplus \in \{\wedge, \vee, +, -, >, <, \equiv, \neq, \cdots\}$$

Figure 4. Language of target programs.

$$
\frac{v_1 = v_2}{(v_2, v_1) \in E_d} \qquad \frac{v_1 = v_2 \oplus v_3}{(v_2, v_1), (v_3, v_1) \in E_d}
$$

$$
\frac{v_1 = \mathit{ite}\,(v_2, v_3, v_4)}{(v_2, v_1), (v_3, v_1), (v_4, v_1) \in E_d}
$$

$$
\frac{v_1 = f(v_2, \cdots) \quad f(u_1, \cdots) = \{u_1 = \langle u_1 \rangle; \cdots; \textbf{return}\ w_1 = w_2;\}}{(v_2, u_1), (w_2, w_1), (w_1, v_1) \in E_d}
$$

$$
\frac{v_1 = f(v_2, \cdots) \quad f(u_1, \cdots) = \varnothing}{(v_2, v_1) \in E_d} \qquad \frac{\textbf{if}\ (v_1 = v_2)\ \{\cdots\}}{(v_2, v_1) \in E_d}
$$

Figure 5. Rules of building data dependence.

Note that the program dependence graph in Fusion is built on the SSA-form code of a given program. Horwitz et al. [32] showed that program dependence graph of non-SSA-form code does not track the order of multiple definitions of the same variable and, thus, is not adequate to represent the program semantics, e.g., path conditions in our context.

Path-Sensitive Sparse Analysis. Algorithm 1 demonstrates how a path-sensitive sparse analysis processes a control flow path, i.e., a list of statements, $s_0, s_1, \cdots, s_n$. In the algorithm, π stands for a data dependence path on the program dependence graph; in_s and out_s stand for the sets of data flow facts before and after a statement; and tr_s is the transfer function of a statement. The sparse analysis propagates the data flow facts via data dependence (Lines 5-6,

Algorithm 1: Sparse analysis of a control-flow path.

```
1  Procedure sparse($s_0, s_1, \cdots, s_n$)
2      $\Pi = \{\}$;
3      for $i = 0 \ldots n$ do
4          $out_{s_i} = tr_{s_i}(in_{s_i})$;
5          if $out_{s_i} \neq \emptyset$ and $(s_i, s_j) \in E_d$ then
6              $in_{s_j} = in_{s_j} \cup out_{s_i}$;
7              if $\exists \pi = (\cdots, s_i) \in \Pi$ then
8                  $\pi = (\cdots, s_i, s_j)$;
9                  $\phi_\pi = \phi_\pi \wedge \phi_{(s_i, s_j)}$;
10             else
11                 $\Pi = \Pi \cup \{(s_i, s_j)\}$;
12     smt_solve $(\bigwedge_{\pi \in \Pi} \phi_\pi)$;
```

Algorithm 2: Inter-procedural sparse analysis.

```
1  Procedure interprocedural_sparse($s_0, s_1, \cdots, s_n$)
2      $\mathbb{S}_1 = \{(\pi_1, tr_{\pi_1}, \phi_{\pi_1}), (\pi_2, tr_{\pi_2}, \phi_{\pi_2}), \cdots\}$;
3      $\mathbb{S}_2 = \{\phi_{ret_1}, \phi_{ret_2}, \cdots\}$;
4      $\Pi = \{\}$;
5      for $i = 0 \ldots n$ do
6          if $\exists (\pi = (s_i, \cdots, s_j), tr_\pi, \phi_\pi) \in \mathbb{S}_1$ then
7              $out_{s_j} = tr_\pi(in_{s_i})$;
8              if $out_{s_j} \neq \emptyset$ and $(s_j, s_k) \in E_d$ then
9                  $in_{s_k} = in_{s_k} \cup out_{s_j}$;
10                 if $\exists \pi' = (\cdots, s_i) \in \Pi$ then
11                     $\pi' = (\cdots, s_i, \cdots, s_j, s_k)$;
12                     $\phi_{\pi'} = \phi_{\pi'} \wedge instantiate(\phi_\pi) \wedge \phi_{(s_j, s_k)}$;
13                 else
14                     $\Pi = \Pi \cup \{(s_i, \cdots, s_j, s_k)\}$;
15         else
16             /* intra-procedural part, which is the same as
               Line 4–Line 11 in Algorithm 1. */
17     smt_solve $(\bigwedge_{\pi \in \Pi} \phi_\pi)$;
```

Algorithm 1), and collects a set $\Pi = \{\pi_1, \pi_2, \cdots\}$ of data dependence paths as well as their path conditions ϕ_{π_i} (Lines 7-11, Algorithm 1). At the end, the procedure solves the path condition to achieve path-sensitivity (Line 12, Algorithm 1).

Example 3.2. Consider the simple program presented as a control flow path in Figure 6(a). Suppose that we perform a taint analysis where *low* means an insensitive value and *high* means the opposite. A conventional analysis propagates all data flow facts along the control flows and find a taint issue at the final statement where a password may be sent to a sensitive site.

Sparse analysis has two unique features known as the spatial sparsity and the temporal sparsity [39]. As shown in Figure 6(b), spatial sparsity means that it only stores the data flow facts used at each statement, and temporal sparsity means that it propagates the data flow facts along the data dependence edges, thus skipping unnecessary control flows.

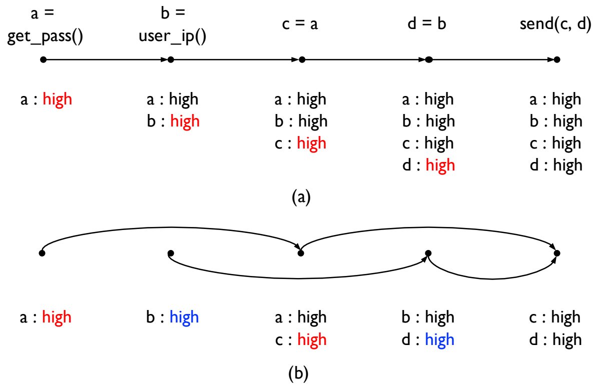


Figure 6. (a) Conventional data flow analysis that propagates all data flow facts along control flows. (b) Sparse analysis propagates data flow facts along data dependence and only stores the data flow facts used in the analysis of a statement.

To path-sensitively check if the sensitive information contained in the variables, *a* and *b*, will be propagated to the final statement, *send(c, d)*, we actually need to check if the two data dependence paths, $\pi_1 = (a, c, send(c, d))$ and $\pi_2 = (b, d, send(c, d))$, on the program dependence graph are simultaneously feasible, i.e., if the conjunction of their path conditions $\phi_{\pi_1} \wedge \phi_{\pi_2}$ is satisfiable.

Problem Statement. Algorithm 2 demonstrates the inter-procedural sparse analysis, which has two scalability problems. The first one is referred to as *condition caching*. In the algorithm, to avoid repetitively analyzing a function, we often cache function summaries in the memory for the future use. Each function summary is either a triple (π, tr_π, ϕ_π) or a condition ϕ_{ret} (Lines 2–3, Algorithm 2). The former summarizes a data dependence path $\pi = (s_0, s_1, ..., s_n)$, where $tr_\pi = tr_{s_n} \circ \cdots \circ tr_{s_1} \circ tr_{s_0}$ is the summarized transfer function and ϕ_π is the path condition. The latter summarizes the condition of a return value, just like the return-value condition $z = y \wedge y = 2x$ of the function bar in Figure 1. As previously discussed, computing and caching these conditions in memory causes a significant time and memory overhead.

The second problem is referred to as *condition cloning*. On one hand, when the propagation of data flow facts reaches a statement at the function entry, the function summary (π, tr_π, ϕ_π) will be instantiated (Lines 6–14, Algorithm 2), leading to a path condition where ϕ_π is cloned (Line 12, Algorithm 2). On the other hand, as demonstrated in Figure 1, a path condition ϕ_π itself may also include many clones of the return-value condition ϕ_{ret} if the path control-depends on the return values from the callees. Such condition cloning leads to an exponential growth of the condition size as the call depth increases, severely limiting the analysis scalability.

Based on the discussion above, we aim to address the following problem to make the path-sensitive sparse analysis shown in Algorithm 2 scalable:

> Given a set Π of data dependence paths explored by the sparse analysis, solve the path condition $\phi_\Pi = \bigwedge_{\pi \in \Pi} \phi_\pi$ without caching and exhaustively cloning conditions.

3.2 Fusing Sparse Analysis and SMT Solving

This section details Fusion in three parts: a linear transformation from the program dependence graph to the path condition, which sets the foundation for our approach (§ 3.2.1); an un-optimized SMT solution that addresses the condition caching problem (§ 3.2.2); and an optimized SMT solution that further addresses the condition cloning problem (§ 3.2.3).

3.2.1 Linear Allotropic Transformation. Our key insight is that, although the program dependence graph and the path conditions are in different forms, they encode the same information, i.e., the data and the control dependence. This section shows that, given a set Π of data dependence paths, we can transform the program dependence graph to the path condition $\phi_\Pi = \bigwedge_{\pi \in \Pi} \phi_\pi$ in linear time and space.

Intra-procedural Transformation. Figure 8 lists the transformation rules for an intra-procedural program dependence graph. These rules can be understood in two steps: slicing (Rules (1)–(3)) and translating (Rules (4)–(6)). The first step produces a slice $G[\Pi] = (V[\Pi], E_d[\Pi], E_c[\Pi])$ of the program dependence graph G with respect to the set Π of data dependence paths. The second step translates the slice $G[\Pi]$ to the path condition ϕ_Π.

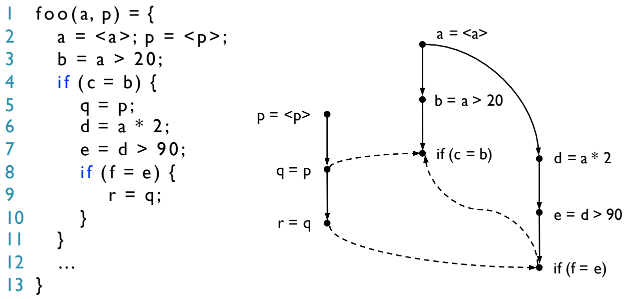
For slicing, Rule (1) processes the ite-statements and identifies the if-branch we propagate the data flow facts. For instance, when we propagate a data flow fact from a statement $v_1 = v_2$ to the statement $v_3 = ite(c, v_1, v_4)$, it is easy to determine that the condition c must be true. In this case, we record the edge to prune, e.g., (v_4, v_3), in a set X_d.

Rules (2) and (3) compute the program slice, which contains the vertices that the paths in Π transitively data- or control-depend on [52]. Rule (2) is to transitively add the control dependence edges, which contain the branch conditions on which the path is feasible. Rule (3) is to add the data dependence edges, which imply how each branch condition is computed. Clearly, the complexity of applying the rules is linear to the slice size.

Example 3.3. Consider the program dependence graph in Figure 7 and $\Pi = \{(p{=}\langle p \rangle, q{=}p, r{=}q)\}$. Rule (2) adds the vertices **if** $(c{=}b)$ and **if** $(f{=}e)$ into $V[\Pi]$. Rule (3) adds all other vertices the two if-statements transitively data-depend on into $V[\Pi]$. Thus, the slice $G[\Pi]$ contains all vertices and edges except those in Π.

Rules (4)–(6) translate the slice to the path condition, i.e., a first-order logic formula in the following language:

$$e := true \mid false \mid v \mid e_1 \oplus e_2 \mid ite\,(e_1, e_2, e_3).$$



```
foo(a, p) = {
  a = <a>; p = <p>;
  b = a > 20;
  if (c = b) {
    q = p;
    d = a * 2;
    e = d > 90;
    if (f = e) {
      r = q;
    }
  }
  ...
}
```

Figure 7. Code example and its program dependence graph.

Rule (4) initializes the path condition. Since a data dependence path is feasible if and only if all the branch conditions it control-depends on are satisfied at runtime, Rule (5) translates each if-statement that the path control-depends on to *true* (denoted as $[\![\cdot]\!]_c$). Rule (6) translates all data dependence relations in the slice to its first-order logic counterpart (denoted as $[\![\cdot]\!]_d$). Apparently, applying the two rules requires a depth-first search on the slice, which is of linear complexity with respect to the slice size or, equivalently, to the size of the resulting path condition.

Example 3.4. Consider Figure 7. Rule (5) interprets the two if-statements that the path depends on into $f \equiv true$ and $c \equiv true$. Rule (6) traverses all vertices in the slice and interprets each statement as $b \equiv a > 20, c \equiv b, d \equiv a{*}2, e \equiv d > 90$, and $f \equiv e$. Thus, the path condition is $f \equiv true \wedge c \equiv true \wedge b \equiv a > 20 \wedge c \equiv b \wedge d \equiv a * 2 \wedge e \equiv d > 90 \wedge f \equiv e$.

Inter-procedural Transformation. The key problem of the inter-procedural analysis is to achieve context-sensitivity. First, we employ Rules (1)–(3) to compute an inter-procedural slice, such as the one in Figure 3. Next, we clone the callee function at each call site to achieve context-sensitivity. Such function cloning allows us to remove the parentheses on the call and return edges. Thus, we can then apply Rules (4)–(6) to compute the path condition except that, we need to consider the following additional translations related to call and return:

$$\frac{(v_1, v_2) \in E_d[\Pi]}{[\![v_2 = \langle v_2 \rangle]\!]_d = v_1 \equiv v_2} \quad (7) \qquad \frac{(v_1, v_2) \in E_d[\Pi] \quad f \neq \varnothing}{[\![v_2 = f(\cdots)]\!]_d = v_1 \equiv v_2} \quad (8)$$

Apparently, the inter-procedural process is still of linear complexity with respect to the size of the resulting condition.

3.2.2 Unoptimized IR-Based SMT Solution. The linear allotropic transformation implies that the program dependence graph is equivalent to the path conditions in terms of the SMT solving. To illustrate, we give a simple sketch of both the conventional SMT solving method and a trivial dependence-graph-based solution in Algorithm 3 and Algorithm 4, respectively.

$$\frac{\pi \in \Pi \quad (v_i, v_1 = ite\,(v_2, v_3, v_4)) \sqsubset \pi \quad v_i \in \{v_3, v_4\}}{(u, v_1) \in X_d \textbf{ where } u \in \{v_3, v_4\} \setminus \{v_i\}} \;(1)$$

$$\frac{\pi \in \Pi \quad v \in \pi \quad (v, v_1), (v_1, v_2), \cdots, (v_{n-1}, v_n) \in E_c}{v_1, v_2, \cdots, v_n \in V[\Pi], (\_, v_1), (v_1, v_2), \cdots, (v_{n-1}, v_n) \in E_c[\Pi]} \;(2) \qquad \frac{v \in V[\Pi] \quad (u, v) \in E_d \setminus X_d}{u \in V[\Pi], (u, v) \in E_d[\Pi]} \;(3)$$

$$\frac{}{\phi_\Pi = true} \;(4) \qquad \frac{(u, v) \in E_c[\Pi]}{\phi_\Pi = \phi_\Pi \wedge [\![v]\!]_c \textbf{ where } [\![\textbf{if } (v_1 = v_2)\{S_1;\}]\!]_c \; = \; (v_1 \equiv true)} \;(5)$$

$$\frac{v \in V[\Pi]}{\phi_\Pi = \phi_\Pi \wedge [\![v]\!]_d \textbf{ where } \begin{array}{rcl} [\![v_1 = v_2]\!]_d & = & v_1 \equiv v_2 \\ [\![v_1 = v_2 \oplus v_3]\!]_d & = & v_1 \equiv v_2 \oplus v_3 \\ [\![v_1 = ite\,(v_2, v_3, v_4)]\!]_d & = & \begin{cases} v_2 \equiv true \wedge v_1 \equiv v_3 & (v_3, v_1) \in E_d[\Pi] \wedge (v_4, v_1) \notin E_d[\Pi] \\ v_2 \equiv false \wedge v_1 \equiv v_4 & (v_3, v_1) \notin E_d[\Pi] \wedge (v_4, v_1) \in E_d[\Pi] \\ v_1 \equiv ite\,(v_2, v_3, v_4) & otherwise \end{cases} \\ [\![\textbf{return } v_1 = v_2]\!]_d & = & v_1 \equiv v_2 \\ [\![\textbf{if } (v_1 = v_2)\{S_1;\}]\!]_d & = & v_1 \equiv v_2 \\ [\![others]\!]_d & = & true \end{array}} \;(6)$$

Figure 8. Transforming an intra-procedural program dependence graph to the path condition of a set Π of data dependence paths. Note that a vertex on the dependence graph stands for both a statement and the variable defined by the statement.

As shown in Algorithm 3, an SMT solver first applies a series of equisatisfiable formula transformations[4] that simplify the input formula and rewrite the formula in a standard form. Clearly, each preprocessing step needs at least linear time and space to scan the input formula. These preprocessing steps play a key role in improving the performance of an SMT solver [20], because the satisfiability of many cases (21% cases in our evaluation) can be decided during this phase, just as shown by the example in Section 2. After preprocessing, the transformed formula is sent to a specific solver according to the theories we employ. For instance, if we employ the bit-vector theory like the previous work [2, 46, 53], i.e., model each variable as a bit vector, the specific solver will bit-blast the condition to a pure Boolean formula and use an SAT solver to determine its satisfiability.

The dependence-graph-based solution accepts the set Π of data dependence paths as its input, applies Rules (1)–(8) to compute the inter-procedural path condition, and calls the conventional SMT solver to determine its satisfiability. Since Rules (1)–(8) are of the linear complexity, Algorithm 4 has the same complexity as the conventional solver.

Impacts on the Scalability. Since Algorithm 3 and Algorithm 4 have the same complexity, it is equivalent to use either the program dependence graph or the path conditions for SMT solving. Such equivalence allows the static analyzer not to compute and cache any path conditions because the program dependence graph is already in memory. To illustrate this advantage, we rewrite the algorithm of the sparse analysis, i.e., Algorithm 2, in Algorithm 5, where we highlight the differences. In comparison, since Algorithm 5 calls the new SMT solution at the end, it no longer needs to compute any condition during the analysis and, thus, does not need to cache any condition in function summaries — $\mathbb{S}_1$ removes ϕ_π from each function summary and $\mathbb{S}_2$ is totally removed. Thus, the overhead caused by condition caching is entirely removed.

Algorithm 3: Conventional SMT solution.

```
Procedure smt_solve($\phi_\Pi$)
    $\phi_\Pi$ = preprocess($\phi_\Pi$);            /* $\Omega$(sizeof($\phi_\Pi$)) */
    if $\phi_\Pi$ is true then
        return sat;
    if $\phi_\Pi$ is false then
        return unsat;
    return specific_solve ($\phi_\Pi$);
```

Algorithm 4: Un-optimized IR-based SMT solution.

```
Procedure ir_based_smt_solve($\Pi$)
    use Rules (1)–(3) to compute a slice;
    clone the callee functions at all call sites in the slice;
    use Rules (4)–(8) to compute $\phi_\Pi$;  /* $O$(sizeof($\phi_\Pi$)) */
    smt_solve ($\phi_\Pi$);
```

3.2.3 Optimized IR-Based SMT Solution.

In addition to discarding the need of caching conditions, fusing static analysis and SMT solving — SMT solving on the program dependence graph — provides many other opportunities for

[4] For instance, given the condition $x > y$, we can transform it to an equisatisfiable condition *true*, because both variables, x and y, in the condition are unconstrained.

Algorithm 5: Optimized inter-procedural analysis.

```
Procedure interprocedural_sparse($s_0, s_1, \cdots, s_n$)
    $\mathbb{S}_1 = \{(\pi_1, tr_{\pi_1},$ ~~$\phi_{\pi_1}$~~$), (\pi_2, tr_{\pi_2},$ ~~$\phi_{\pi_2}$~~$), \cdots\}$;
    ~~$\mathbb{S}_2 = \{\phi_{ret_1}, \phi_{ret_2}, \cdots\}$~~;
    $\Pi = \{\}$;
    for $i = 0 \ldots n$ do
        if $\exists(\pi = (s_i, \cdots, s_j), tr_\pi, \phi_\pi) \in \mathbb{S}_1$ then
            $out_{s_j} = tr_\pi(in_{s_i})$;
            if $out_{s_j} \neq \emptyset$ and $(s_j, s_k) \in E_d$ then
                $in_{s_k} = in_{s_k} \cup out_{s_j}$;
                if $\exists\pi' = (\cdots, s_i) \in \Pi$ then
                    $\pi' = (\cdots, s_i, \cdots, s_j, s_k)$;
                    ~~$\phi_{\pi'} = \phi_{\pi'} \wedge instantiate(\phi_\pi) \wedge \phi_{(s_j, s_k)}$~~;
                else
                    $\Pi = \Pi \cup \{(s_i, \cdots, s_j, s_k)\}$;
        else
            /* intra-procedural part, which is the same as
            Line 4–Line 11 in Algorithm 1, except that, in this
            algorithm, we do not compute any $\phi_\pi$. */
    ir_based_smt_solve ($\Pi$);
```

Algorithm 6: Optimized IR-based SMT solution.

```
1 Procedure ir_based_smt_solve($\Pi$)
2     use Rules (1)–(3) to compute a slice $G[\Pi]$;
3     foreach function $f$ on $G[\Pi]$ do
4         use Rules (4)–(6) to compute the local condition $\phi_{\Pi,f}$;
5         $\phi_{\Pi,f}$ = intraprocedural_preprocess($\phi_{\Pi,f}$);
6     interprocedural_preprocess($G[\Pi]$);
7     clone $\phi_{\Pi,f}$ at call sites with Rules (7)–(8);
8     return smt_solve($\bigwedge_f \phi_{\Pi,f}$);
```

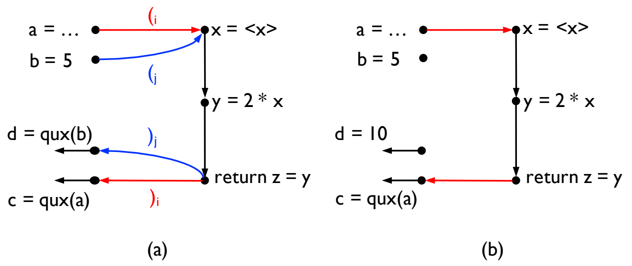


Figure 9. (a) Program dependence graph where parentheses are used to distinguish two call sites, *c=qux(a)* and *d=qux(b)*. (b) After constant propagation, the edge labels are deleted.

further acceleration. Our key insight is that the program dependence graph can provide rich program information to guide the SMT solving. We note that this insight has been studied from different angles in some previous work. For instance, prior work shows that both the data and the control dependence can be utilized to speed up SMT solving [10, 54].

In this paper, we focus on the condition cloning problem in the path-sensitive sparse analysis and propose a solution that utilizes the program structure to optimize the preprocessing steps in the SMT solver. Recall that, we need to instantiate (or clone) the conditions from the callee functions to achieve context-sensitivity. Such cloning leads to the exponential growth of the condition size and, thus, significantly limits the performance of both static analysis and SMT solving. Algorithm 6 illustrates our optimized method, which, compared to Algorithm 4, delays the condition cloning after a series of intra-procedural and inter-procedural preprocessing procedures. We can understand the benefits of Algorithm 6 from the following two aspects.

Reducing the Number of Functions to Clone. In the intra-procedural phase (Lines 3–5, Algorithm 6), we first apply Rules (4)–(6) on the intra-procedural dependence graph, which yields an intra-procedural path condition. Such local path conditions can be preprocessed by a series of standard methods like the Gaussian elimination to reduce the condition size. Thus, the size of the conditions to clone is reduced.

More importantly, the inter-procedural phase aims to eliminate the unnecessary function calls, thereby reducing the function clones (Line 6, Algorithm 6). For instance, we discussed that, after the unconstrained-value propagation, we can already solve the path condition in Figure 3 without the need of cloning the callee function bar. Figure 9 shows the other example, where the call and the return edges, labeled by $(_j$ and $)_j$, are deleted after inter-procedural constant propagation, a classic preprocessing method [5, 43]. Since the callee function qux is called only once after preprocessing, we do not need to clone the intra-procedural condition $\phi_{\Pi,qux}$, i.e., $y \equiv 2 * x \wedge z \equiv y$, to distinguish the call sites.

Speeding up Preprocessing. In addition to reducing the condition size, the modular structure of the program dependence graph also significantly accelerates the preprocessing steps. First, the inter-procedural propagation-style preprocessing can be made more efficient via the modular structure. The example in Figure 3 illustrates that we can avoid repetitively visiting a function by creating a quick path between the entry and the exit of a function.

Second, expensive preprocessing methods, such as the Gaussian elimination, are decomposed into multiple functions. Assume that a preprocessing step in an original solver has the complexity $O(\hbar(sizeof(\wedge_{f \in \mathcal{F}} \phi_{\Pi,f})))$ where $\mathcal{F}$ is the set of all functions and their clones, and $\hbar$ is a high-complexity function. In our framework, we perform this expensive step intra-procedurally to avoid repetitive processes on different clones of the same local condition, thus having the complexity $O(\Sigma_{f \in \mathcal{F}'} \hbar(sizeof(\phi_{\Pi,f}))) + O(sizeof(\wedge_{f \in \mathcal{F} \setminus \mathcal{F}'} \phi_{\Pi,f}))$ where $\mathcal{F}'$ is the set of functions without clones. The first component of the complexity is the total cost of preprocessing each function. The second component is the cost of cloning the preprocessed condition at each call site. The complexity is significantly reduced in practice, because

$$\begin{aligned}
&O(\hbar(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}}\phi_{\Pi,f})))\\
&\quad\geq O(\hbar(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}'}\phi_{\Pi,f}))) + O(\hbar(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}\setminus\mathcal{F}'}\phi_{\Pi,f})))\\
&\quad\geq O(\hbar(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}'}\phi_{\Pi,f}))) + O(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}\setminus\mathcal{F}'}\phi_{\Pi,f}))\\
&\quad\geq O(\textstyle\sum_{f\in\mathcal{F}'}\hbar(\mathit{sizeof}(\phi_{\Pi,f}))) + O(\mathit{sizeof}(\textstyle\bigwedge_{f\in\mathcal{F}\setminus\mathcal{F}'}\phi_{\Pi,f})).
\end{aligned}$$

3.3 Discussion on the Benefits of Fusion

First, the scalability of the path-sensitive sparse analysis is improved due to the following reasons:

(1) *No caching.* The linear transformation from the program dependence graph to the path conditions demonstrates the equivalence of the two data structures in terms of SMT solving. Such equivalence allows us not to compute and cache any path conditions (§ 3.2.2).

(2) *Little cloning.* The modularity of the program dependence graph allows further optimization, which not only reduces the function clones but also speeds up the preprocessing procedures in our SMT solver (§ 3.2.3).

The other consequence is that our fused design also eases the efforts of engineering a path-sensitive sparse analyzer:

(3) *On the analysis side*, with the dependence-graph-based solver, developers no longer need to care about the details of computing path conditions and can focus on the design of the data flow analysis, i.e., the abstract domains and the transfer functions. This is illustrated in Algorithm 5 where we do not compute any condition during the analysis (§ 3.2.2).

(4) *On the SMT solver side*, we can easily implement many SMT optimizations, including ours and those proposed in previous work [10, 54]. This is because the input data structure of our solver is a program IR, which encodes rich program information, such as the data dependence, the control dependence, the modular program structure, and many others (§ 3.2.3).

4 Implementation

We have implemented Fusion on top of the LLVM compiler infrastructure (Version 3.6) [33] and the Z3 SMT solver (Version 4.5) [20] to detect null exceptions and taint issues in C/C++ code. This section briefly discusses the implementation of the sparse analysis and the SMT solver in Fusion.

Sparse Analysis in Fusion. Fusion accepts the LLVM bitcode and the program dependence graph as its inputs. The program dependence graph is built offline using Pinpoint, one previous work on sparse analysis [46]. Since our implementation aims to detect bugs rather than to rigorously verify the correctness of a program, we made a few reasonably unsound (a.k.a., soundy [35]) assumptions following previous bug detectors [2, 46, 49, 53]. For example, for field sensitivity, we regarded each field of a class or struct as an independent object. All members of an array or union were assumed to be the alias of one another. We did not handle global variables, exceptions (long jumps), inline assembly, and C style function pointers but used a class hierarchy analysis to resolve virtual calls. Recursive calls are handled as loops by unrolling each cycle twice on the call graph.

In addition to null exceptions, we also implemented two taint analyses for checking relative path traversal (CWE-23[5]) and transmission of private resources (CWE-402[6]). The former allows attackers to access files outside of a restricted folder and is modeled as a data dependence path from an external input to file operations, e.g., from *input=gets(...)* to *fopen(...)*. The latter may leak private data to attackers and is modeled as a data dependence path from sensitive data to I/O operations, e.g., from *password=getpass(...)* to *sendmsg(...)*.

SMT Solver in Fusion. In our dependence-graph-based solver, we implemented the intra-procedural preprocessing procedures, including forward and backward constant propagation, equality propagation, unconstrained-variable elimination, Gaussian elimination, and strength reduction. We have also implemented inter-procedural preprocessing procedures, such as constant propagation, equality propagation, and the "unconstrained" property propagation. All these preprocesses are standard [20] and their details are omitted. If the preprocesses cannot decide the satisfiability, we model each variable in the path condition as a bit vector. The length of each bit vector is the bit width, e.g., 32, of the variable type, e.g., an integer type. The specific solver (Line 7, Algorithm 3) then calls Z3's bit-blaster to convert a bit-vector condition to a pure Boolean condition. Z3's SAT solver will determine the satisfiability of the Boolean condition.

5 Evaluation

Fusion is continuously scanning open-source software and, to date, has detected over a hundred previously-unknown bugs. Some of the bugs were even assigned CVE identifiers due to their security impact. All these bugs and CVE identifiers have been made available online.[7] This section focuses on the evaluation of our main contribution — how the fused design scales up path-sensitive sparse analysis — by comparing Fusion to existing industrial-strength techniques.

Baseline Approaches. First, we compared Fusion to Pinpoint [46], the most recent sparse analyzer with the same precision as Fusion but following a non-fused design. Thus, comparing it to Pinpoint will clearly show the value of our fused design. We also implemented several variants of Pinpoint by arming it with quantifier elimination, formula simplification, and abstract refinement, which are expected to reduce the size of path conditions. In addition, we also evaluated the key component, i.e., the dependence-graph-based SMT solver, by comparing it to a state-of-the-art solver, Z3 [20]. This aims

[5] https://cwe.mitre.org/data/definitions/23.html

[6] https://cwe.mitre.org/data/definitions/402.html

[7] https://fusion-scan.github.io

Table 2. Subjects for evaluation.

ID	Program	Size (KLoC)	# Functions	# Vertices	# Edges
1	mcf	2	26	22.8K	28.9K
2	bzip2	3	74	93.8K	120.4K
3	gzip	6	89	165.3K	221.5K
4	parser	8	324	824.2K	1,114.1K
5	vpr	11	272	376.3K	478.0K
6	crafty	13	108	381.1K	498.9K
7	twolf	18	191	762.9K	995.5K
8	eon	22	3.4K	1.2M	1.3M
9	gap	36	843	3.4M	4.4M
10	vortex	49	923	3.3M	4.2M
11	perlbmk	73	1.1K	9.3M	12.2M
12	gcc	135	2.2K	14.2M	18.4M
13	ffmpeg	1,001	74.2K	57.1M	76.4M
14	v8	1,201	260.4K	63.0M	73.5M
15	mysql	2,030	79.2K	68.8M	85.0M
16	wine	4,108	133.0K	90.2M	112.3M

Table 3. Performance compared to Pinpoint.

ID	Memory (GB)			Time (Seconds)		
	Fusion	Pinpoint	Speedup	Fusion	Pinpoint	Speedup
1	0.1	1.1	11×	4	19	5×
2	0.1	2.3	23×	4	172	43×
3	0.1	1.3	13×	3	30	10×
4	0.1	3.3	33×	49	233	5×
5	0.1	1.9	19×	3	145	48×
6	0.1	1.3	13×	2	23	12×
7	0.2	1.8	9×	41	95	2×
8	0.1	1.8	18×	2	21	11×
9	2.2	39.1	18×	53	2,033	38×
10	0.6	8.9	15×	164	1,769	11×
11	1.0	19.4	19×	227	2,524	11×
12	1.5	27.7	18×	339	2,615	8×
13	11.8	55.7	5×	689	5,899	9×
14	8.6	82.1	10×	748	7,672	10×
15	7.9	98.8	13×	1,250	9,057	7×
16	11.2	98.3	9×	772	8,893	12×

to show that SMT solving in Fusion can benefit from the modular program structure and, thus, is faster. Finally, we compared Fusion to Infer [24], a non-sparse but prominent and mature static analyzer from industry.

Benchmarks. Table 2 lists the subjects used in the evaluation, including the standard benchmark SPEC CINT2000 [30] (1 – 12) as well as four industrial-sized open-source projects with millions of lines of code (13 – 16). For reference, Table 2 also reports the number of functions and the size of the program dependence graphs.

Environment. All experiments were run on a server with eighty "Intel Xeon CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04. In the experiments, we use fifteen threads to run each static analyzer. Following previous works [46, 53], each call of the SMT solver is run with a limit of 10 seconds. An analysis of a program is run with the limit of 12 hours and 100GB of memory.

5.1 Comparing Fusion to Pinpoint

For a fair comparison, Fusion and Pinpoint are configured to find bugs on the same program dependence graph. Since they work with the same precision and the only difference is whether they employ the fused design, the bugs they report are the same. For instance, both of them reported 92, 293, 168, and 139 null exceptions for ffmpeg, v8, mysql, and wine, respectively. In what follows, we focus on discussing the experimental results of our key contribution to scalability.

To understand techniques such as qualifier elimination (QE) and formula simplification (FS), which have the potential to reduce the condition size and improve the scalability, we implemented three variants of Pinpoint: Pinpoint+QE, Pinpoint+LFS, and Pinpoint+HFS. We implemented QE using the "qe" tactic of Z3. LFS means lightweight formula simplification, which just performs local formula rewriting. We implemented it using the "simplify" tactic of Z3. HFS means heavyweight formula simplification, which simplifies a formula depending on the context where the formula exists. It is expensive because it needs to invoke the SMT solver several times during the simplification. We implemented it using the "ctx-solver-simplify" tactic of Z3.

In addition, we also tried to arm Pinpoint with the abstract refinement (AR) method [2]. This AR method does not immediately compute a full path condition to determine the path feasibility. Instead, it firstly computes and solves an intra-procedural condition and gradually extends the condition by adding conditions from callers and callees until the condition satisfiability can be decided.

In what follows, we detail the experimental results on checking null exceptions and briefly summarize the comparison results of the taint analysis.

Time and Memory. The results of comparing Fusion to Pinpoint are listed in Table 3. Compared to Fusion, Pinpoint consumes 5× to 33× memory and takes 2× to 48× time. The computational resources consumed by Fusion is even available in a common personal computer.[8] The better performance of Fusion confirms the value of our fused design, which allows us to avoid condition caching and excessive condition cloning in a sparse analysis.

To study whether QE, FS, and AR can reduce the condition size and improve the analysis scalability, we also ran Pinpoint+QE, Pinpoint+LFS, Pinpoint+HFS, and Pinpoint+AR on the benchmark programs. We observe that Pinpoint+QE only succeeded in analyzing mcf, the smallest project, but consumed 140× memory (14GB vs. 0.1GB) and took 77× time (308s vs. 4s) compared to Fusion. Pinpoint+QE failed to analyze all other projects because the memory was exhausted.

[8] We succeeded in reproducing the results of Fusion in a Macbook Pro 16" with 16GB RAM and 2.3GHz 8-core Intel Core i9 processor.

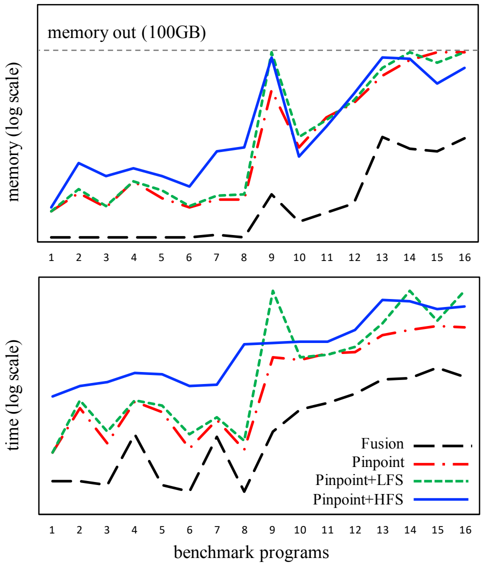


Figure 10. Fusion vs. Pinpoint and its variants.

Similarly, Pinpoint+AR failed to analyze all projects with more than 50KLoC due to timeout and only worked for other small projects with 14× time cost on average. We looked into the problems and found that this is because QE is of high complexity and may take a lot of time but notably enlarge the condition size; and AR frequently invokes the SMT solver, which significantly degrades the performance.

Pinpoint+LFS and Pinpoint+HFS also do not work well. The results are shown in Figure 10. In comparison to Pinpoint, they only help several benchmark projects, such as perlbmk (ID=11) and mysql (ID=15), to reduce the memory overhead a bit. Overall, LFS and HFS do not reduce memory overhead but make Pinpoint significantly slower due to the extra cost brought on by the formula simplification procedures.

SMT Solver in Fusion. The key component of Fusion is the dependence-graph-based SMT solver, which benefits from the structure of program dependence graph to avoid excessive condition cloning, thus being faster than a standalone and general-purpose SMT solver.

To demonstrate the efficiency of our SMT solution, we record the time cost of solving conditions for checking null exceptions. Meanwhile, we also use the default solver of Z3 to solve these conditions. In the experiment, we got 310,462 SMT instances, in which 60% are satisfiable, 40% are not, and 21% can be solved in the preprocessing phase of the solver. We believe such a large number of instances are sufficient to evaluate the performance of SMT solving. Figure 11 illustrates the solving time of the baseline approach and our solution on all the SMT instances. We can observe that most of the dots in the figure are under the diagonal, meaning that our SMT solution is more efficient. To summarize, for satisfiable instances, our solver is 3.0× faster than the default Z3 solver and, for unsatisfiable instances, our solver is 1.8× faster on average. Overall, our SMT solution is about 2.5× faster than the default Z3 solver.

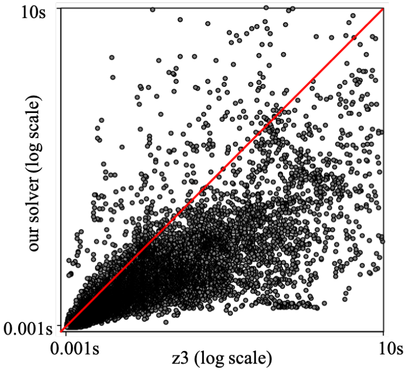


Figure 11. Time of SMT solving on all benchmarks.

Table 4. Taint analysis on the industrial-sized projects.

Issue	ID	Fusion		Pinpoint	
		Memory	Time	Memory	Time
CWE-23	13	10.1GB	519s	59.2GB (6×)	4,892s (9×)
	14	7.3GB	812s	90.1GB (12×)	6,999s (9×)
	15	8.2GB	1,192s	92.1GB (11×)	8,391s (7×)
	16	11.9GB	709s	Memory Out (>100GB)	
CWE-402	13	8.2GB	608s	69.0GB (8×)	6,009s (10×)
	14	8.9GB	721s	80.8GB (9×)	7,108s (10×)
	15	7.9GB	1,423s	88.2GB (11×)	9,924s (7×)
	16	10.7GB	806s	93.2GB (9×)	9,276s (12×)

Study of the Taint Analysis. We also ran the taint analyses discussed in § 4 over the benchmark projects. Owing to the page limits, we cannot present all the detailed results. Instead, we only present the time and memory cost of checking the industrial-sized projects in Table 4. The results are quite similar to that of checking null exceptions. Compared to Pinpoint, Fusion demonstrates 10× speedup but consumes only 11% of the memory on average.

5.2 Comparing Fusion to Infer

To understand the performance compared to other static analyzers, we also ran Fusion against Infer — an abduction based system from Facebook — to check null exceptions in the four industrial-sized projects. The results are demonstrated in Table 5, including the time and memory cost, as well as the number of reported bugs (#Report) and true/-false positives (#TP/#FP). On average, Fusion only consumes 16% of the memory and 31% of the time, but reports more real bugs with fewer false positives. For instance, Infer even fails to analyze the subject, wine, with 100GB of memory, while Fusion only takes 772 seconds and 11.2GB of memory. One reason for Infer's higher memory overhead is that it generates and caches many function summaries, which are

Table 5. Comparing Fusion to Infer.

ID	Program	Fusion		Infer	
		Memory	Time	Memory	Time
13	ffmpeg	11.8GB	689s	10.9GB (1×)	1,322s (2×)
14	v8	8.6GB	748s	92.1GB (11×)	3,829s (5×)
15	mysql	7.9GB	1,250s	43.2GB (5×)	3,402s (3×)
16	wine	11.2GB	772s	Memory Out (>100GB)	

ID	Program	Fusion			Infer		
		#Report	#TP	#FP	#Report	#TP	#FP
13	ffmpeg	92	52	40	132	45	87
14	v8	293	216	77	329	167	162
15	mysql	168	124	44	441	94	347
16	wine	139	98	41	Memory Out (>100GB)		
FP Rate		29.2%			66.1%		

definitely one thing we aim to avoid in Fusion. The relatively low recall and precision of Infer is due to its compromises of path-sensitivity, its limited capability of detecting cross-file bugs, and the innate approximation of abduction [8].

6 Related Work

Sparse Program Analysis. Sparse program analysis was greatly facilitated after the birth of the SSA form [16, 17], which explicitly encodes the def-use relations and allows the propagation of data flow facts along def-use chains. Reif and Lewis [41] proposed a sparse algorithm for constant propagation, which was then extended to conditional constant propagation using a sparse representation known as SSA-graph [51]. Hardekopf and Lin [27] proposed a semi-sparse algorithm for flow-sensitive pointer analysis and then extended it to a full sparse algorithm [28]. Madsen and Møller [37] proposed a special sparse analysis for JavaScript programs. Cherem et al. [11] employed sparse analysis to detect software bugs like memory leak, followed by a few works refining its recall and precision [45, 46, 48, 49]. All these sparse analyses are not inter-procedurally path-sensitive except Pinpoint [46], which follows a non-fused design.

Unlike the aforementioned works proposed for certain particular application scenarios, Oh et al. [39] generalized the idea of sparse analysis in the framework of abstract interpretation. Our work is also described in a general setting but aims to address a different problem, i.e., the scalability issue caused by inter-procedural path-sensitivity.

Sparse evaluation [13, 40] is a coarse-grained approach to sparse analysis, which aims to construct a compact control flow graph by removing statements with identity transfer functions. For an analysis that considers the semantics of all statements, these techniques cannot remove any statement and will be degenerated into a non-sparse analysis. Also, literature on sparse evaluation does not show how to achieve path-sensitivity and, thus, is different from our work.

Path-Sensitivity. Path-sensitivity is critical for the precision of static analysis [2, 34, 46, 53]. However, it has only a few studies on path-sensitive sparse analysis, which, as discussed before, do not follow a fused design [45–47].

For conventional data flow analysis, many methods have been proposed to mitigate the high overhead of computing path conditions. Yorsh et al [55] generate concise summaries with succinct path conditions for a special abstract domain. Their approach heavily depends on formula simplifications [1, 22, 36], which have been demonstrated to be expensive in our evaluation. Saturn [53] simplifies path conditions using binary decision diagrams [6] and only works with the precision of intra-procedural path-sensitivity. Babic and Hu [2] proposed a refinement-based method that expects to determine path feasibility with small or imprecise path conditions. As shown in our evaluation, such refinement-based methods only work for small projects, as the scalability suffers with the refinement of abstractions. Similar issues also have been shown in other refinement based works [3, 9, 12, 14, 31].

Other path-sensitive analyses include ESP [18], trace partitioning [38], elaborations [44], and many others. They employ various heuristics to control the trade-off between performing a join operation or a disjunction at the merge points on the control flow graph. SMPP [29] enumerates program paths and learns facts from a path to accelerate the analysis of other paths. Dillig et al. [21] focused on solving recursive path conditions and proposed a sound and complete path-sensitive analysis. Different from these techniques, we focus on a totally different problem, i.e., the explosive size of path conditions in a path-sensitive sparse analysis. We believe that these previous ideas are complementary to ours, and their combination has the potential for greater scalability.

7 Conclusion

This paper presents Fusion, a sparse analysis framework embodying a fused design that lowers the bar of deploying static analysis in practice. Our evaluation shows that Fusion is able to path-sensitively analyze millions of lines of code within just a few computational resources available in a common personal computer. Fusion has demonstrated a promising bug detection capability, and found over a hundred previously-unknown bugs in mature open-source software.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK-20202001) and NSFC61902329. Other authors are supported by the GRF16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, and the donations from Microsoft and Huawei. Peisen Yao and Rongxin Wu are the corresponding authors.

References

[1] Alessandro Armando and Silvio Ranise. 2003. Constraint contextual rewriting. *Journal of Symbolic Computation* 36, 1-2 (2003), 193–216. https://doi.org/10.1016/S0747-7171(03)00025-7

[2] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. IEEE, 211–220. https://doi.org/10.1145/1368088.1368118

[3] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 1–3. https://doi.org/10.1145/503272.503274

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[5] David Binkley. 1994. Interprocedural constant propagation using dependence graphs and a data-flow model. In *Proceedings of the 3rd International Conference on Compiler Construction (CC '94)*. Springer, 374–388. https://doi.org/10.1007/3-540-57877-3_25

[6] Randal E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* 100, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[7] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX, 209–224. https://doi.org/10.5555/1855741.1855756

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[9] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402. https://doi.org/10.1109/TSE.2004.22

[10] Jianhui Chen and Fei He. 2018. Control flow-guided SMT solving for program verification. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE '18)*. ACM, 351–361. https://doi.org/10.1145/3238147.3238218

[11] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789

[12] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: Compositional bounded model checking for real-world programs. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE '13)*. IEEE, 136–146. https://doi.org/10.1109/ASE.2013.6693074

[13] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, 55–66. https://doi.org/10.1145/99583.99594

[14] Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC '03)*. ACM, 368–371. https://doi.org/10.1145/775832.775928

[15] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 603–625. https://doi.org/10.1145/504709.504710

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, 25–35. https://doi.org/10.1145/75277.75280

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[18] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, 57–68. https://doi.org/10.1145/512529.512538

[19] James H. Davenport and Joos Heintz. 1988. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation* 5, 1-2 (1988), 29–35. https://doi.org/10.1016/S0747-7171(88)80004-X

[20] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[21] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 270–280. https://doi.org/10.1145/1375581.1375615

[22] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *Proceedings of the 17th International Static Analysis Symposium (SAS '10)*. Springer, 236–252. https://doi.org/10.1007/978-3-642-15769-1_15

[23] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 567–577. https://doi.org/10.1145/1993498.1993565

[24] Dino Distefano, Manuel Fahndrich, Francesco Logozzo, and Peter O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[26] Michael J. Fischer and Michael O. Rabin. 1998. Super-exponential complexity of Presburger arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 122–135. https://doi.org/10.1007/978-3-7091-9459-1_5

[27] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 226–238. https://doi.org/10.1145/1480881.1480911

[28] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO '11)*. IEEE, 289–298. https://doi.org/10.1109/CGO.2011.5764696

[29] William R Harris, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2010. Program analysis via satisfiability modulo path programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 71–82. https://doi.org/10.1145/1706299.1706309

[30] John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35. https://doi.org/10.1109/2.869367

[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL

'02). ACM, 58–70. https://doi.org/10.1145/503272.503279

[32] Susan Horwitz, Jan Prins, and Thomas Reps. 1988. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, 146–157. https://doi.org/10.1145/73560.73573

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75:1–75:12. https://doi.org/10.1109/CGO.2004.1281665

[34] Benjamin Livshits and Monica S. Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '03)*. ACM, 317–326. https://doi.org/10.1145/940071.940114

[35] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[36] Salvador Lucas. 1995. Fundamentals of context-sensitive rewriting. In *Proceedings of the 21st International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 405–412. https://doi.org/10.1007/3-540-60609-2_25

[37] Magnus Madsen and Anders Møller. 2014. Sparse dataflow analysis with pointers and reachability. In *Proceedings of the 21st International Static Analysis Symposium (SAS '14)*. Springer, 201–218. https://doi.org/10.1007/978-3-319-10936-7_13

[38] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming (ESOP '05)*. Springer, 5–20. https://doi.org/10.1007/978-3-540-31987-0_2

[39] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 229–238. https://doi.org/10.1145/2254064.2254092

[40] Ganesan Ramalingam. 2002. On sparse evaluation representations. *Theoretical Computer Science* 277, 1-2 (2002), 119–147. https://doi.org/10.1016/S0304-3975(00)00315-7

[41] John H. Reif and Harry R. Lewis. 1977. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, 104–118. https://doi.org/10.1145/512950.512961

[42] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

[43] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

[44] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static analysis in disjunctive numerical domains. In *Proceedings of the 13th International Static Analysis Symposium*. Springer, 3–17. https://doi.org/10.1007/11823230_2

[45] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 812–823. https://doi.org/10.1145/3377811.3380346

[46] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[47] Qingkai Shi and Charles Zhang. 2020. Pipelining bottom-up data flow analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 835–847. https://doi.org/10.1145/3377811.3380425

[48] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, 265–266. https://doi.org/10.1145/2892208.2892235

[49] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

[50] Peng Tu and David Padua. 1995. Efficient building and placing of gating functions. In *Proceedings of the 16th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, 47–55. https://doi.org/10.1145/223428.207115

[51] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136

[52] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[53] Yichen Xie and Alex Aiken. 2005. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, 351–363. https://doi.org/10.1145/1040305.1040334

[54] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*. ACM, 38–50. https://doi.org/10.1145/3395363.3397378

[55] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 221–234. https://doi.org/10.1145/1328897.1328467