Path-Sensitive Sparse Analysis without Path Conditions

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

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

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

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.

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
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. Our key insight is that program dependence graph and path condition are allotropes, which means 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, 28×–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.

---

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.

---

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.
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. That is, since the variables, \( a \) and \( b \), are unconstrained, 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.

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.

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

<table>
<thead>
<tr>
<th></th>
<th>computing</th>
<th>solving</th>
<th>caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>conventional</td>
<td>( O(kn + m) )</td>
<td>( O(kn + m) )</td>
<td>( O(kn + m) )</td>
</tr>
<tr>
<td>fusion</td>
<td>( O(n + m) )</td>
<td>( O(n + m) )</td>
<td>-</td>
</tr>
</tbody>
</table>

\( n \): size of bar. \( m \): size of foo. foo calls bar \( k \) times.

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

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., \( f(v) = \{ v = (v); \cdots \} \). The if-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 \( \phi \)-assignment, e.g., \( v_1 = \cdots ; \) if \( (c = \cdots ; \) \( v_2 = \cdots ; \) \( v_1 = \phi(v_1, v_2) \). In our language, we use \( v_3 = \text{ite}(c, v_2, v_1) \) to replace the \( \phi \)-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 and we can merge multiple definitions via a \( \phi \)-assignment, e.g., \( v_1 = \cdots ; \) if \( (c = \cdots ; \) \( v_2 = \cdots ; \) \( v_1 = \phi(v_1, v_2) \). In our language, we use \( v_3 = \text{ite}(c, v_2, v_1) \) to replace the \( \phi \)-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.

\[
\begin{align*}
\text{Program } P & := F + \\
\text{Function } F & := f(v_1, v_2, \cdots) = [ S_1 ] \\
& \quad | f(v_1, v_2, \cdots) = \emptyset \\
\text{Statement } S & := v_1 = \langle v_1 \rangle \\
& \quad | v_1 = v_2 \\
& \quad | v_1 = v_2 \otimes v_3 \\
& \quad | v_1 = \text{ite}(v_2, v_3, v_4) \\
& \quad | v_1 = f(v_2, v_3, \cdots) \\
& \quad | \text{return } v_1 = v_2 \\
& \quad | \text{if } (v_1 = v_2) \{ S_1; \} \\
& \quad | S_1; S_2 \\
\end{align*}
\]

**Figure 4.** Language of target programs.

\[
\begin{align*}
\text{data dependence path on the program semantics, e.g., path conditions in our context.}
\end{align*}
\]

*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, \( \pi \) stands for a data dependence path on the program dependence graph; \( i_n \) and \( o_n \) stand for the sets of data flow facts before and after a statement; and \( t_r \) 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.

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{\texttt{sparse}}{(s_0, s_1, \ldots, s_n)}
\State $\Pi = \emptyset$;
\For{$i = 0 \ldots n$}
\State $\text{outs}_s = \text{tr}_{\pi}(\text{in}_{s_i})$;
\If{$\text{outs}_s \neq \emptyset$ and $(s_i, s_j) \in E_d$}
\State $\text{in}_{s_j} = \text{in}_{s_j} \cup \text{outs}_s$;
\EndIf
\EndFor
\State $\text{smt\_solve}(\land_{\pi \in \Pi} \phi_{\pi})$;
\EndProcedure
\end{algorithmic}
\end{algorithm}

Algorithm 2: Inter-procedural sparse analysis.

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{\texttt{interprocedural\_sparse}}{(s_0, s_1, \ldots, s_n)}
\State $S_1 = \{(\pi_1, tr_{\pi_1}, \phi_{\pi_1}), (\pi_2, tr_{\pi_2}, \phi_{\pi_2}), \ldots\}$;
\State $S_2 = \{\phi_{\text{ret}_1}, \phi_{\text{ret}_2}, \ldots\}$;
\State $\Pi = \emptyset$;
\For{$i = 0 \ldots n$}
\If{$\exists (\pi = (s_i, \ldots, s_j), tr_{\pi}, \phi_{\pi}) \in S_1$}
\State $\text{outs}_s = tr_{\pi}(\text{in}_{s_i})$;
\If{$\text{outs}_s \neq \emptyset$ and $(s_j, s_k) \in E_d$}
\State $\text{in}_{s_k} = \text{in}_{s_k} \cup \text{outs}_s$;
\EndIf
\Else
\State $\Pi = \Pi \cup \{(s_i, s_j, s_k)\}$;
\EndIf
\Else
\State /* intra-procedural part, which is the same as Line 4–11 in Algorithm 1. */
\State $\text{smt\_solve}(\land_{\pi \in \Pi} \phi_{\pi})$;
\EndIf
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

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

means that it propagates the data flow facts along the data dependence edges, thus skipping unnecessary control flows.

To path-sensitively check if the sensitive information contained in the variables, $a$ and $b$, will be propagated to the final statement, $\text{send}(c, d)$, we actually need to check if the two data dependence paths, $\pi_1 = (a, c, \text{send}(c, d))$ and $\pi_2 = (b, d, \text{send}(c, d))$, on the program dependence graph are simultaneously feasible, i.e., if the conjunction of their path conditions $\phi_{\pi_1} \land \phi_{\pi_2}$ 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 $(\pi, tr_{\pi}, \phi_{\pi})$ or a condition $\phi_{\text{ret}}$ (Lines 2–3, Algorithm 2). The former summarizes a data dependence path $\pi = (s_0, s_1, \ldots, s_n)$, where $tr_{\pi} = tr_{s_0} \circ \cdots \circ tr_{s_n}$ is the summarized transfer function and $\phi_{\pi}$ is the path condition. The latter summarizes the condition of a return value, just like the return-value condition $z = y \lor y = 2x$ 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 $(\pi, tr_{\pi}, \phi_{\pi})$ will be instantiated (Lines 6–14, Algorithm 2), leading to a path condition where $\phi_{\pi}$ is cloned (Line 12, Algorithm 2). On the other hand, as demonstrated in Figure 1, a path condition $\phi_{\pi}$ itself may also include many clones of the return-value condition $\phi_{\text{ret}}$ 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.

**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
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 \( \Pi \) 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 \( \Pi \) 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 \( \Pi \) of data dependence paths. The second step translates the slice \( G[\Pi] \) to the path condition \( \phi_\Pi \).

For slicing, Rule (1) processes the if-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 \( v_3 = \text{ite}(c, v_1, v_4) \), 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 \( G[\Pi] \) 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 \( \Pi = \{ (p=p), (q=p), (r=q) \} \). 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 \( \Pi \).

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

\[
e := \text{true} \mid \text{false} \mid v \mid e_1 \oplus e_2 \mid \text{ite}(e_1, e_2, e_3).
\]

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 \( [\, ]_d \)). Rule (6) translates all data dependence relations in the slice to its first-order logic counterpart (denoted as \( [\, ]_d \)). 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 \( f \equiv \text{true} \) and \( c \equiv \text{true} \). 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 \( f \equiv \text{true} \land c \equiv \text{true} \land b \equiv a > 20 \land c \equiv b \land d \equiv a+2 \land e \equiv d > 90 \land f \equiv e \).

**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:

\[
(v_1, v_2) \in E_d[\Pi] \quad (v_2 = (v_3)) \quad \text{if} \quad f \neq \emptyset
\]

\[
[v_2 = (v_3)]_d = v_1 \equiv v_2
\]

\[
[v_2 = f(\cdots)]_d = v_1 \equiv v_2
\]

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.
As shown in Algorithm 3, an SMT solver first applies a series of equisatisfiable formula transformations\(^5\) 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 hit-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 \( \Pi \) 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 3 has the same complexity as the conventional solver.

\textbf{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 either program dependence graph or path conditions are unconstrained.

\(5\) 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.
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 cloning condition 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 be cloned 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 \((\text{f} \text{ and } \text{y})\), are deleted after inter-procedural constant propagation, a classic preprocessing method [5, 43]. Since the callee function \text{qux} is called only once after preprocessing, we do not need to clone the intra-procedural condition \(\phi_{\text{intra\_qux}}, \text{i.e., } y \equiv 2 \times x \land z \equiv y, \) 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 \(O(h(\text{size}(\land f \in F, \phi_{\text{intra\_f}})))\) where \(F\) is the set of all functions and their clones, and \(h\) 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 \(O(2_f \in F, h(\text{size}(\land f \in F, \phi_{\text{intra\_f}})))) + O(\text{size}(\land f \in F, \phi_{\text{intra\_f}}))\) where \(F'\) 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
We have implemented Fusion on top of the LLVM compiler infrastructure using Pin.

The program dependence graph is built to verify the correctness of a program, we made a few reasonable 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)\(^6\) and transmission of private resources (CWE-402). 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 \texttt{input-gets(...)}. The former makes private data to attackers and is modeled as a data dependence path from sensitive data to I/O operations, e.g., from \texttt{password-getpass(...)}. 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 \texttt{password-getpass(...)}. The former makes private data to attackers and is modeled as a data dependence path from sensitive data to I/O operations, e.g., from \texttt{password-getpass(...)}. 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 \texttt{password-getpass(...)}

### 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.

**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 contributon — 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

---

\(^{6}\)https://cwe.mitre.org/data/definitions/23.html

\(^{7}\)https://cwe.mitre.org/data/definitions/402.html

\(^{7}\)https://fusion-scan.github.io
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 Pinpoint, Fusion 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. 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.

---

**Table 2. Subjects for evaluation.**

<table>
<thead>
<tr>
<th>ID</th>
<th>Program</th>
<th>Size (KLoC)</th>
<th># Functions</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mcf</td>
<td>2</td>
<td>26</td>
<td>22.8K</td>
<td>28.9K</td>
</tr>
<tr>
<td>2</td>
<td>bzip2</td>
<td>3</td>
<td>74</td>
<td>95.8K</td>
<td>120.4K</td>
</tr>
<tr>
<td>3</td>
<td>gzip</td>
<td>6</td>
<td>89</td>
<td>165.3K</td>
<td>221.5K</td>
</tr>
<tr>
<td>4</td>
<td>parser</td>
<td>8</td>
<td>324</td>
<td>824.2K</td>
<td>1,114.1K</td>
</tr>
<tr>
<td>5</td>
<td>vpr</td>
<td>11</td>
<td>272</td>
<td>376.3K</td>
<td>478.0K</td>
</tr>
<tr>
<td>6</td>
<td>crafty</td>
<td>13</td>
<td>108</td>
<td>381.1K</td>
<td>498.9K</td>
</tr>
<tr>
<td>7</td>
<td>twolf</td>
<td>18</td>
<td>191</td>
<td>762.9K</td>
<td>995.5K</td>
</tr>
<tr>
<td>8</td>
<td>eon</td>
<td>22</td>
<td>3.4K</td>
<td>1.2M</td>
<td>1.3M</td>
</tr>
<tr>
<td>9</td>
<td>gap</td>
<td>36</td>
<td>843</td>
<td>3.4M</td>
<td>4.4M</td>
</tr>
<tr>
<td>10</td>
<td>vortex</td>
<td>49</td>
<td>923</td>
<td>3.3M</td>
<td>4.2M</td>
</tr>
<tr>
<td>11</td>
<td>perlbmk</td>
<td>73</td>
<td>1.1K</td>
<td>9.3M</td>
<td>12.2M</td>
</tr>
<tr>
<td>12</td>
<td>gcc</td>
<td>135</td>
<td>2.2K</td>
<td>14.2M</td>
<td>18.4M</td>
</tr>
<tr>
<td>13</td>
<td>ffmpeg</td>
<td>1,001</td>
<td>74.2K</td>
<td>57.1M</td>
<td>76.4M</td>
</tr>
<tr>
<td>14</td>
<td>v8</td>
<td>1,201</td>
<td>260.4K</td>
<td>63.0M</td>
<td>73.5M</td>
</tr>
<tr>
<td>15</td>
<td>mysql</td>
<td>2,030</td>
<td>79.2K</td>
<td>68.8M</td>
<td>76.4M</td>
</tr>
<tr>
<td>16</td>
<td>wine</td>
<td>4,108</td>
<td>133.0K</td>
<td>90.2M</td>
<td>112.3M</td>
</tr>
</tbody>
</table>

**Table 3. Performance compared to Pinpoint.**

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

---

[5] 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.
Similarly, Pinpoint+AR failed to analyze all projects with more than 50KLoC due to timeout and only worked for other small projects with 14\times 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 perlbench (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 and, for unsatisfiable instances, our solver is 1.8\times faster on average. Overall, our SMT solution is about 2.5\times faster than the default Z3 solver.

**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\times speedup but consumes only 11% of the memory on average.

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

<table>
<thead>
<tr>
<th>Issue</th>
<th>ID</th>
<th>Fusion</th>
<th>Pinpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Memory</td>
<td>Time</td>
</tr>
<tr>
<td>CWE-23</td>
<td>13</td>
<td>10.1GB</td>
<td>519s</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>7.3GB</td>
<td>812s</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>8.2GB</td>
<td>1,192s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.9GB</td>
<td>709s</td>
</tr>
<tr>
<td>CWE-402</td>
<td>13</td>
<td>8.2GB</td>
<td>608s</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8.9GB</td>
<td>721s</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>7.9GB</td>
<td>1,423s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>10.7GB</td>
<td>806s</td>
</tr>
</tbody>
</table>

Marked green yellow.  

Z3 solver and, for unsatisfiable instances, our solver is 1.8\times faster on average. Overall, our SMT solution is about 2.5\times faster than the default Z3 solver.

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
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


