Path-Sensitive Inference of Function Precedence Protocols

Murali Krishna Ramanathan Ananth Grama Suresh Jagannathan
{rmk, ayg, suresh}@cs.purdue.edu
Department of Computer Science, Purdue University

Abstract

Function precedence protocols define ordering relations among function calls in a program. In some instances, precedence protocols are well-understood (e.g., a call to `pthread_mutex_init` *must always be present on all program paths before a call to* `pthread_mutex_lock`*). Oftentimes, however, these protocols are neither well-documented, nor easily derived. As a result, protocol violations can lead to subtle errors that are difficult to identify and correct.*

In this paper, we present CHRONICLER*, a tool that applies scalable inter-procedural path-sensitive static analysis to automatically infer accurate function precedence protocols.* CHRONICLER *computes precedence relations based on a program's control-flow structure, integrates these relations into a repository, and analyzes them using sequence mining techniques to generate a collection of feasible precedence protocols. Deviations from these protocols found in the program are tagged as violations, and represent potential sources of bugs.*

We demonstrate CHRONICLER*'s effectiveness by deriving protocols for a collection of benchmarks ranging in size from 66K to 2M lines of code. Our results not only confirm the existence of bugs in these programs due to precedence protocol violations, but also highlight the importance of path sensitivity on accuracy and scalability.*

1 Introduction

Program specifications form an important aspect of the software development process. The lack of proper specifications has two significant negative consequences: (a) interfaces may be used incorrectly as programs evolve; and (b) confidence in the correctness of programs and absence of bugs is reduced. Indeed, absence of precise specifications often leads to incompletely validated software, and compromises software dependability and reliability.

Many of these errors occur because program implementations do not adhere to implicitly-assumed precedence protocols, which dictate how different program components may be ordered. For example, a call to function `pthread_mutex_init` must be present upstream on any program path from a call to `pthread_mutex_lock`, since the former initializes data structures associated with the mutex into an initially unlocked state.The precedence relation between `pthread_mutex_init` and `pthread_mutex_lock` forms part of a precedence protocol that define how these functions can be used in programs.

In some cases, precedence protocols are well-specified. Certain groups of library functions, for example, a call to `accept` should always be preceded by a call to `bind` and `socket`, a call to `pthread_mutex_lock` must always be preceded by a call to `pthread_mutex_init`, etc., have well-understood relationships with one another. When precedence protocols are known, a variety of techniques can be employed to check that they are faithfully obeyed in programs [7, 13, 24]. In general, however, the relationships that exist among most functions in a program (especially non-library ones), are known only to the designers of those functions, and are rarely documented precisely.

Existing research on specification inference [15, 3, 4, 9, 14, 23, 22, 12, 6, 16, 10, 25, 20, 17, 21] is based on the premise that commonly occurring patterns in analyzed programs are indicative of a likely specification. In this paper, we further qualify this notion by statically deriving procedure call patterns that reflect precedence relations. Like previous efforts, we also assume that most programs are well-written, and that call patterns which occur often are likely to be correct.

We define an inter-procedural path-sensitive static analysis that computes a collection of constraints whose solution defines potential precedence relations among procedure calls. These relations are of the form, "*a call to procedure q must always be preceded by a call to procedure p*". Path-sensitivity ensures that *all* paths leading to q have a call to procedure p; in contrast, path-insensitivity would only require that *some* path exists between p and q. However, naive exploration of all paths in a program is infeasible. To make our approach tractable, we memoize path information. To infer the precedence protocol for q, we construct ordered sequences of function calls that always precede q at each call-site of q. Let $\texttt{C} = \{c_1, c_2, ..., c_k\}$ be the set of call sites to function q. At any call-site $c_i \in \texttt{C}$, the memoized path associated with c_i forms an ordered sequence $s_q^{c_i}$ of function calls that always precede c_i. We sequence mine [2] the set of sequences, $\bigcup_{c_i \in \texttt{C}} s_q^{c_i}$ based on a user-defined confidence level to obtain the precedence protocol.

```
181 RI_FKey_check(PG_FUNCTION_ARGS)
182 {
199   ri_CheckTrigger(...);
210   pk_rel = heap_open(...);
248   if (tgnargs == 4)
249   {
        // match_type not checked
250     ri_BuildQueryKeyFull(...);
294   }
296   match_type = ri_DetermineMatchType(...);
298   if (match_type == RI_MATCH_TYPE_PARTIAL)
299     ereport(...);
303   ri_BuildQueryKeyFull(...);
437 }
```

Figure 1. Extract from `RI_FKey_check` **in postGreSQL-8.1.3.**

To motivate the problem, consider the code fragment in Figure 1. This fragment shows part of procedure `RI_FKey_check` from `postGreSQL`, version 8.1.3. Observe that the call to `ri_BuildQueryKeyFull` at line 303 is preceded by calls to `ri_DetermineMatchType`, `heap_open`, and `ri_CheckTrigger` in this order. This pattern occurs at several other locations in the program. However, in one specific instance of the call to `ri_BuildQueryKeyFull` at line 250, the rule is not satisfied, as there is no call to `ri_DetermineMatchType` preceding it. The absence of this call is significant; if the `match_type` is `RI_MATCH_TYPE_PARTIAL`, the call to `ri_BuildQueryKeyFull` is erroneous because the procedure does not handle arguments of this type. Path sensitive analysis is *critical* to deriving these inferences. We are unaware of other specification inference mechanisms that can generate precedence relations among function calls in this manner.

1.1 Preceded-by and Followed-by Relations

In this paper, we focus only on precedence properties ("a call to q must be *preceded* by a call to p"). Notably, our implementation does not consider constraints of the form "a call to p is *followed-by* a call to q". While useful in certain contexts, we observe that protocols of this form are less precise than precedence protocols, especially in the presence of non-local jumps, errors, and exceptions. A precedence relation captures the set of antecedent constraints (e.g., initialization invariants) that must be satisfied before a procedure call can be made; a follows relation captures the set of consequent actions (e.g., finalization invariants) that must occur after a procedure call is executed.

A precedence relation captures behavior that is guaranteed to occur if the protocol is properly obeyed (e.g., a call to `pthread_mutex_lock` must be preceded by a call to `pthread_mutex_init`). On the other hand, the fact that a forward relation is not satisfied does not necessarily imply that program behavior is incorrect. Consider the following snippet from some procedure `p`:

```
if((id = user_open(...)) == err) {
   print("error");
   return;
}
user_close(id);
```

Here, we cannot assert that every call to `user_open` will always be followed by a call to `user_close` because an error condition that occurs in the interim may lead to an abnormal exit from procedure `p`. On the other hand, if a call to `user_close` does take place, it is guaranteed that a preceding call to `user_open` would have occurred if the protocol was properly followed. This is a key insight that enables us to infer precise precedence protocols whose violation may signal the presence of a bug.

1.2 Technical Contributions

This paper makes the following technical contributions:

1. **Precedence Protocol Inference:** We present a new inter-procedural static analysis for inferring function precedence protocols. The constraints computed by the analysis capture causal (path-sensitive) dependencies.

2. **Experimentation:** We demonstrate the practicality of our techniques by applying CHRONICLER, a tool that incorporates the above techniques, to real-world C sources ranging from 66K to 2M lines of code.

3. **Assessment:** We provide a qualitative assessment of violations of precedence protocols that signify the presence of bugs, potential performance bottlenecks, and possible security-related errors.

1.3 Running Example

To provide further motivation for the goals of our work, consider the program fragments shown in Figure 2 that serve as a running example in the rest of the paper. We define eight procedures: `main`, `f`, `g`, `h`, `lwrap`, `uwrap`,

```
void main() {      void f() {        void g() {       void h() {     void lwrap() {     void uwrap() {
   if(cond1)          u_init();         if(cond)         if(cond3)      u_init();          u_access();
      f();            ...                 u_init();       lwrap();      ...                ...
   elseif(cond2)      ...               ...              else          }                  }
      g();            u_access();       u_access();        lwrap();
   else               ...               ...              ...
      h();         }                 }                  uwrap();
}                                                      }
```

Figure 2. Illustrative Example

u_init and u_access. The definitions of procedures u_init and u_access are not shown; u_init initializes data structures (e.g., by allocating memory, opening files, etc.) and u_access accesses these data structures (e.g., by writing into memory, reading files, etc.). By examining the interaction of these eight procedures, we can postulate a precedence protocol that every call to u_access must be preceded by a call to u_init. Observe that this rule is quite obvious in procedure f, not so obvious in h since procedure boundaries are crossed, and not guaranteed to be satisfied in g (when cond is false, there is no call to u_init). Our goal is to extract similar interesting protocols (of arbitrary complexity), and to locate regions in the source where these protocols are violated. Before describing our technique, we formally introduce terms and notation used in the rest of the paper.

2 Framework

2.1 Definitions and Notation

- **Precedes relation** ($a \leftarrow b$): A binary relation between procedures *a* and *b*, which specifies that a call to *b* is always preceded by a call to *a*. It is important to note that this relation does not imply immediacy; rather it implies that a call to *a* definitely occurs at some point upstream from a call to *b*. The precedes relation is insensitive to procedure boundaries, and thus the calls to *a* and *b* may be in different procedures.

- **Relation Chain**: An n-ary relation among procedures defined, such that if ($a \leftarrow b$) holds and ($b \leftarrow c$) holds, then ($a \leftarrow b \leftarrow c$) also holds.

- **Anchor**: In a relation chain ($a_1 \leftarrow a_2 \leftarrow a_3 \ldots$), $\forall i, j, i < j$, a_i is an *anchor* for a_j.

- **Definitive constraint** ($a \overset{p}{\leftarrow} b$): Within the context of any call to procedure p, *b* is *always* called and the *precedes* relation $a \leftarrow b$ holds; e.g., u_init $\overset{f}{\leftarrow}$ u_access. Relation chains generalize to constraints in an obvious manner.

- **Conditional constraint** ($a \overset{p}{\leftarrow} \widehat{b}$): For each call to procedure p, *b may* be called, and whenever *b* is invoked, the *precedes* relation $a \leftarrow b$ always holds. e.g., u_init $\overset{g}{\leftarrow} \widehat{\mathtt{g}}_C$. A conditional constraint chain that emanates from b is a composition of the definitive constraint chain emanating from a and the conditional constraint $a \overset{p}{\leftarrow} \widehat{b}$.

2.2 Generating Constraints

Every procedure p is associated with two *phantom* elements, p_C and p_R. The entry to a procedure p is always through p_C and the exit is always through p_R, unless control exits abnormally. An inferred precedes relation[1], $a \leftarrow b$ is classified as a definitive constraint $a \overset{p}{\leftarrow} b$ if and only if there exists a satisfying precedes relation chain $b \leftarrow \ldots \leftarrow p_R$. In other words, the constraint is definitive if in any call to the procedure p, b is *always* called (such constraints are subsequently used in the construction of a constraint summary for the procedure.). Otherwise, the precedes relation is classified as conditional $a \overset{p}{\leftarrow} \widehat{b}$.

We do not apply transitive closure on the precedes relation and only consider immediate predecessors in our framework. If we considered transitive closure, we would potentially infer a large number of spurious relations. For example, even though pthread_mutex_init $\leftarrow$ pthread_mutex_lock and pthread_mutex_lock $\leftarrow$ pthread_mutex_unlock are true, considering the relation pthread_mutex_init $\leftarrow$ pthread_mutex_unlock, while true, is unhelpful and does not reflect an interesting case. Instead, it is the relation chain, pthread_mutex_init $\leftarrow$ pthread_mutex_lock $\leftarrow$ pthread_mutex_unlock derived from the above two relations that yields a useful protocol. Thus, rather than using transitivity to build new relations from immediate precedences, we use relation chains instead. In our framework, for any element y (except p_C) within procedure p, we consider exactly one other element x in p for which $x \leftarrow y$ is true.

Table 1 presents the definitive and conditional prece-

[1]We defer the discussion on how the precedes relation is generated to Section 3.1.

Proc	Definitive	Conditional
$\mathtt{main}$	$\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \mathtt{main}_R$	$\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \widehat{\mathtt{f}}$ $\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \widehat{\mathtt{g}}$ $\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \widehat{\mathtt{h}}$
$\mathtt{f}$	$\mathtt{u\_access} \overset{\mathtt{f}}{\leftarrow} \mathtt{f}_R$ $\mathtt{u\_init} \overset{\mathtt{f}}{\leftarrow} \mathtt{u\_access}$ $\mathtt{f}_C \overset{\mathtt{f}}{\leftarrow} \mathtt{u\_init}$	
$\mathtt{g}$	$\mathtt{u\_access} \overset{\mathtt{g}}{\leftarrow} \mathtt{g}_R$ $\mathtt{g}_C \overset{\mathtt{g}}{\leftarrow} \mathtt{u\_access}$	$\mathtt{g}_C \overset{\mathtt{g}}{\leftarrow} \widehat{\mathtt{u\_init}}$
$\mathtt{h}$	$\mathtt{uwrap} \overset{\mathtt{h}}{\leftarrow} \mathtt{h}_R$ $\overline{\mathtt{lwrap}} \overset{\mathtt{h}}{\leftarrow} \mathtt{uwrap}$ $\mathtt{h}_C \overset{\mathtt{h}}{\leftarrow} \overline{\mathtt{lwrap}}$	$\mathtt{h}_C \overset{\mathtt{h}}{\leftarrow} \widehat{\mathtt{lwrap}}$ $\mathtt{h}_C \overset{\mathtt{h}}{\leftarrow} \widehat{\mathtt{lwrap}}$
$\mathtt{lwrap}$	$\mathtt{u\_init} \overset{\mathtt{lwrap}}{\leftarrow} \mathtt{lwrap}_R$ $\mathtt{lwrap}_C \overset{\mathtt{lwrap}}{\leftarrow} \mathtt{u\_init}$	
$\mathtt{uwrap}$	$\mathtt{u\_access} \overset{\mathtt{uwrap}}{\leftarrow} \mathtt{uwrap}_R$ $\mathtt{uwrap}_C \overset{\mathtt{uwrap}}{\leftarrow} \mathtt{u\_access}$	

Table 1. Constraint Repository for the source in Fig 2. $\overline{\mathtt{lwrap}}$ denotes `lwrap` occurs on both branches.

dence constraints for the example in Figure 2 for procedures `u_init` and `u_access`. We assume the definitive constraints $\mathtt{u\_init}_C \overset{\mathtt{u\_init}}{\leftarrow} \mathtt{u\_init}_R$ and $\mathtt{u\_access}_C \overset{\mathtt{u\_access}}{\leftarrow} \mathtt{u\_access}_R$ are present. Note that a conditional constraint need not always be present for every procedure (e.g., procedure `f`). Also observe that there may be the same conditional constraint may be duplicated (e.g., procedure `h`); these constraints correspond to calls of the same procedure executed along different branches of a conditional expression. The duplicate constraints are kept for increasing the confidence in any protocol – notice that the elements associated in the duplicate constraints refer to different call-sites of the same procedure.

To further improve precision, *constraint summaries* derived for one procedure can be used to refine constraints for procedures that call it. Suppose p is a procedure called from q. Instead of recomputing the constraints for p whenever it is called to build accurate constraint chains for q, we use p's constraint summary, as a representation of its invariants. Intuitively, a constraint summary for procedure p is a memoized characterization of a definite constraint chain for p_R. Thus, a constraint summary for p is obtained by computing a constraint chain over its definitive constraints, and is of the form $p_C ... \overset{p}{\leftarrow} ...p_R$. Thus, p's summary specifies the constraints that definitely hold whenever it is called. After generating the initial set of definitive constraints, constraint summaries ($\gamma(q)$) for each procedure (q) are computed using a fixed point calculation to take into account cyclic constraints that arise in the presence of recursion and loops. An example use of constraint summary is refining the constraint $\mathtt{u\_access} \overset{\mathtt{f}}{\leftarrow} \mathtt{f}_R$ to $\mathtt{u\_access}_C \overset{\mathtt{u\_access}}{\leftarrow} \mathtt{u\_access}_R \overset{\mathtt{f}}{\leftarrow} \mathtt{f}_R$.

2.3 Mining the Constraint Repository

Sequence mining is used to extract precedence information from constraint chains. If precedence protocols need to be identified for procedure p, we first identify constraint chains that emanate from each call-site of p. These comprise the set of sequences that is input to the sequence mining component. We present below the sequences associated with `u_access` from the refined constraint repository in our example:

- $\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \mathtt{f}_C \overset{\mathtt{f}}{\leftarrow} \mathtt{u\_init}_C \overset{\mathtt{u\_init}}{\leftarrow} \mathtt{u\_init}_R \overset{\mathtt{f}}{\leftarrow} \mathtt{u\_access}_C$
- $\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \mathtt{g}_C \overset{\mathtt{g}}{\leftarrow} \mathtt{u\_access}_C$
- $\mathtt{main}_C \overset{\mathtt{main}}{\leftarrow} \mathtt{h}_C \overset{\mathtt{h}}{\leftarrow} \mathtt{lwrap}_C \overset{\mathtt{lwrap}}{\leftarrow} \mathtt{u\_init}_C \overset{\mathtt{u\_init}}{\leftarrow} \mathtt{u\_init}_R \overset{\mathtt{lwrap}}{\leftarrow} \mathtt{lwrap}_R \overset{\mathtt{h}}{\leftarrow} \mathtt{uwrap}_C \overset{\mathtt{uwrap}}{\leftarrow} \mathtt{u\_access}_C$

A sequence for each constraint chain is easily obtained by listing all the anchors in order. Recall that an element a is an anchor of b if and only if $a \leftarrow ... \leftarrow b$ holds.

A sequence mining algorithm takes as input a set of sequences (*I*), user-defined confidence threshold, and outputs a set (*S*) of sequences that occur as subsequences in a minimum fraction (as specified by the confidence threshold) of input sequences. Observe that if a subsequence x is frequently occurring, all subsequences of x also occur at least as frequently as x. Therefore, we consider only maximal subsequences, i.e., it is the case that every sequence(s_i) in *S* is not a subsequence of any other sequence present in *S*.

For example, if the set of sequences is given by $\{(a \leftarrow b \leftarrow c \leftarrow e), (a \leftarrow d \leftarrow c \leftarrow e), (a \leftarrow c \leftarrow e), (a \leftarrow c \leftarrow d \leftarrow e \leftarrow f), (e \leftarrow f \leftarrow d \leftarrow c \leftarrow a)\}$, a sequence miner detects $(a \leftarrow c \leftarrow e)$ as a frequently occurring subsequence. We use the Apriori-all algorithm by Agrawal and Srikant [2], which is known to scale over a million sequences. Due to space limitations, we refer the reader to [2] for more details.

The use of a confidence level provides added flexibility to discover protocols that occur frequently, not just exclusively, and allows greater scalability of the implementation by permitting more aggressive abstraction of the call graph. Identifying frequently occurring protocols is also essential for bug finding. For example, if a discovered protocol occurs nine of 10 times, it is reasonable to assume that the protocol is indeed genuine, and the sole deviation is likely to be a bug. Protocols discovered with a 100% confidence are those that are guaranteed to be true, and for which no violations occur.

Consider the three calls to `u_access` in Figure 2. In two of these calls, namely calls that occur within procedures `f` and `uwrap`, we discover that a call to `u_init` must have preceded the call to `u_access`. However, the call to `u_access` that occurs in procedure `g` is not guaranteed to be preceded by `u_init` because of the unknown value of the guard `cond`. If we perform sequence mining with a confidence level of 0.66, we deduce that `u_access` is preceded by `u_init`, and that there is one violation of this protocol at procedure `g`. This violation signals a potential error.

3 Implementation

CHRONICLER takes as input the program source and a user-defined confidence level for determining when a constraint chain should form part of a procedure's protocol. We first generate the control-flow graph for each procedure in the program using [5]. We simplify the control-flow graph by pruning all nodes other than those corresponding to procedure calls. We reverse the direction of all edges in the control flow graph since we need to construct the precedes relation. The graphs obtained are fed into the relation builder and a cycle of relation, constraint, and constraint summary calculations is executed. The sequences obtained as a result of this process are then fed to a sequence miner implemented based on the ideas presented in [2]. The protocols output by the sequence miner and the associated violations are ranked by processing them according to the confidence, length and frequency of occurrence of the protocol. The output of the entire process is a ranked order of function precedence protocols and the program points in the source where these protocols are violated.

The time taken by CHRONICLER is essentially composed of the time taken for constraint generation and sequence mining. It is easy to show that the time taken for constraint generation for a procedure is sub-quadratic in the number of call-sites within the procedure. The time taken for sequential mining is dependent on the user-confidence level and the sequences generated; readers are referred to [2] for details.

3.1 Relation Builder

A relation builder builds relations among procedure calls. Recall that a precedes relation ($a \leftarrow b$) for procedures a and b states that a call to b is preceded by a call to a in the pruned control-flow graph. Thus, for any call to b, there can be exactly one call to a such that $a \leftarrow b$.

We employ a simple graph walking strategy to build the precedes relations on the simplified (and reversed) control-flow graph. To find the nearest anchor for any call to procedure b within the body of procedure p, a unit flow is initiated at the node corresponding to the call in p's control-flow graph. If the out-degree of the node is d, the flow on each edge emanating from that node is $\frac{1}{d}$. Each neighbor, on receiving a flow adds the incoming value to its flow value, defined as the flows from all of its incoming edges, and distributes the received value (as described above) to its neighbors, unless the sum of the flow received equals one. A node that receives a unit flow is considered an anchor for the node on whose behalf the flow was generated. Observe that our definition of the precedes relation is not captured using standard notions of dominance [18].

Figure 4 presents (reversed) control-flow graphs for the program fragments shown in Figure 3. Circles correspond to call-sites and the squares correspond to other nodes in the control flow graph. The square nodes are presented here only for ease of understanding. For these fragments, we determine `a` $\leftarrow$ `b` is satisfied, i.e., a call to procedure `b` is always preceded by a call to `a`. In Figure 4(a), `a` $\leftarrow$ `b` is obvious. In Figure 4(b), the unit flow initiated at `b` results in a total inflow of 0.5 at the square nodes and 1 at the call to `a`, again allowing us to deduce that `a` $\leftarrow$ `b`. In Figure 4(c), even though a call to `a` is present in a path on the branch, the inflow received at that node is 0.5 and therefore `a` $\leftarrow$ `b` does not hold. Figure 4(d) is similar to Figure 4(c). Observe, however, that determining whether the precedence relation `a` $\leftarrow$ `b` holds in this case depends upon the actions performed by procedures `c` and `d`. Consider the case when both these procedures in turn call procedure `e`. In this case, the correct precedence relation should be `e` $\leftarrow$ `b`.

In other words, suppose the constraint summaries for `c` and `d` are $\texttt{c}_C \leftarrow \texttt{e} \leftarrow \texttt{c}_R$ and $\texttt{d}_C \leftarrow \texttt{e} \leftarrow \texttt{d}_R$, respectively. In this case, the precedes relation `a` $\leftarrow$ `b` no longer holds (recall we do not consider the closure of the relation under transitivity), even though `a` is still an anchor of `b`.

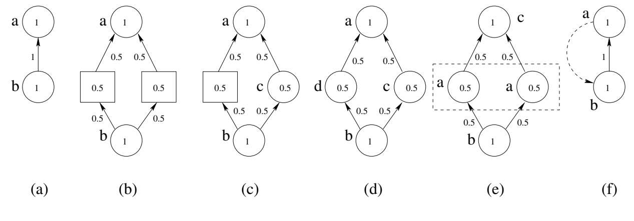


Figure 4. Graphs for program fragments shown in Figure 3.

In Figure 4(e), a unit flow from `b`'s call-site results in an inflow of 0.5 at each of the calls to `a` and unit inflow at node `c`. Our algorithm adds the inflow of all the nodes with the same label within a procedure, leading to a unit in-

```
void p() {     void q() {       void r() {      void s() {      void t() {    void u() {
   a();           a();             a();            a();            c();
   b();           if(cond)         if(cond)        if(cond)        if(cond)       while(a())
                    x++;             x++;            d();            a();           b();
                  else             else            else            else             ...
                    y++;             c();            c();            a();
                  b();             b();            b();            b();
}              }                }               }               }             }
```

Figure 3. Program Constructions.

flow for a virtual node corresponding to the two calls to `a`, again allowing us to determine that `a` $\leftarrow$ `b` holds. Observe that the inflows cannot be added if the corresponding nodes are not in different paths. We apply a conservative approximation and eliminate all loops in our implementation i.e., every loop is unrolled exactly once. In other words, every loop is treated as a conditional branch, except that nothing is present on the 'else' part of the branch. Because of this approximation, the relation `a` $\leftarrow$ `b` holds for the graph shown in Figure 4(f) as well. Note that the relation `b` $\leftarrow$ `a` is not generated because the call to `a` is not preceded by `b` always and therefore the edge from `a` to `b` in the (reversed) control flow graph will be removed. We present the algorithm in Figure 5.

```
procedure BUILDRELATION
    ▷ Input: G(V,E), directed, acyclic (reversed) CFG of p;
             V is topologically sorted, $p_C = |V|$, $p_R = 0$;
    ▷ Output: R = $\bigcup(i,j)$ where $i \leftarrow j$ and $i, j \in V$
1   for each node $i$ in V
2       for each neighbor $j$ of $i$
3           table$(i,j) \leftarrow \frac{1}{d(i)}$
4           for each $k > j$
5               table$(i,k) \leftarrow \frac{1}{d(i)}$ * table$(j,k)$
6       Find minimum $k$ in table s.t table$(i,k) = 1$
7       for each element $x$ between $i$ and $k$
8           Q $\leftarrow$ elements between $i$ and $k$ with $label$(x)
            such that for any $y, z \in$ Q, $y \leftarrow z$ = false
9           Create new node $n$ with $label$(x)
10          Modify E to show collapse of nodes in Q into $n$
11          Remove elements in Q from V
12          Topological sort of vertices between $i$ and $k$
13          table(n, *) $\leftarrow \Sigma_{m \in Q}$ table(m, *)
14      Find minimum $k$ in table s.t table$(i,k) = 1$
15      R $\leftarrow \bigcup(k,i)$
```

Figure 5. Building the precedes relation.

3.2 Constraint Identification

In this section, we discuss the methodology to identify definitive constraints. After building the basic precedes relation, the next step is to identify the constraints so that they can be effectively used in constructing the constraint summary for a procedure and can serve as input to the sequence miner. The input to this process of constraint identification is the relation pairs obtained from the relation builder. For procedure p, we identify the precedes-relation for p_R and label the corresponding relation as a definitive constraint. If $a \leftarrow p_R$ is the identified relation, we obtain the precedes relation for a and label it as a definitive constraint and iterate through this process until p_C is reached. Relations not labeled definitive are labeled conditional.

3.3 Constraint Summaries

Recall constraint summaries are used to improve the precision of the generated constraints. After identifying constraints, a constraint summary for a procedure p is constructed by calculating the definitive constraint chain with p_R as the source and p_C as the sink. Observe that the constraint summary of a procedure signifies that the calls in the chain associated with the constraint summary *always* happens.

Depending on the precision desired, the constraint summary of p can be used at a call-site for p in some other procedure q to improve the precision of q's constraint summary and other constraints associated with q. This is achieved by replacing the node corresponding to p in q with p's constraint summary in q. As new nodes are introduced into q, the relation builder needs to be reinvoked to handle changed scenarios which may be similar to the construction shown in Figure 4(d).

We can tradeoff precision for efficiency by reducing the number of iterations in the fixpoint calculation used to compute relation chains. A larger number of iterations will "fold" more constraints into a relation chain, and capture a larger number of precedence relations, but will lead to longer analysis times. In practice, we expect that complex cyclic dependency chains will not often arise, and that precedence protocols will not often span many procedure boundaries. Nonetheless, the relationship between the quality of computed dependencies and the number of iterations executed in the fixpoint used to build relation chains is a topic for further investigation.

4 Experiments

We validate CHRONICLER on selected benchmark sources, with a view to demonstrating its scalability and effectiveness. We also examine the impact of confidence parameters on the number and nature of inferred precedence protocols.

We extract precedence relations for five sources: `apache`, `gimp`, `linux`, `openssh` and `postgresql`. Specific details relating to the sources are provided in Table 2. The size of selected benchmarks varies from 66K to 2M lines of code. Since default configurations are used to compile these sources, we believe that the number of control flow nodes may be a more reliable indicator of effective source size than lines of code. The number of control flow nodes ranged from 66K to 1M. However, as discussed earlier, we can prune the nodes of the control flow graph that do not correspond to procedure call sites. In such a pruned control flow graph, the number of nodes varied from 6K - 108K. We also present the number of user-defined procedures examined in the table.

We implemented the constraint generator framework and sequence miner in C++. We perform our experiments on a Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on an Intel(R) Pentium(R) 4 CPU machine operating at 3.00GHz, with 1GB memory. We use a confidence level of 80% when mining the constraint chains. The precedence protocols detected, the number of violations associated with it along with the time taken for performing the analysis are presented in Table 2 .

4.1 Quantitative Assessment

Figure 6(a) presents the distribution of the protocols with respect to the confidence intervals for each of the benchmarks. Note that with a 5% increase in the confidence threshold level, the total number of protocols inferred could be easily cut by 50% for most of the benchmarks. Figure 6(b) presents the distribution of frequency of the precedence protocols (usually referred to as *support*). For any two protocols with the same confidence, the protocol that occurs with greater frequency is ranked higher; recall our intuition that protocol frequency should be positively correlated to correctness. Figure 6(c) presents the distribution of length of the precedence protocols, where length is defined in terms of constraint chain sizes. Note that while many protocols have a constraint length of two or less, there is a significant fraction with length five or more. The length of a protocol is loosely correlated with its complexity; longer length protocols are less likely to be easily verified manually, and bugs that arise because of violations of these protocols can be expected to be more difficult to identify.

We also consider the impact of path sensitivity on the accuracy of our results. We designed a path insensitive variant, roughly based on PR-Miner [15], where we associate, for every call-site of function f, all the functions that are called from the function that calls f and the functions called from those functions, etc.. For example, an association for function `h` in `main` in Figure 2 would include $\{$`f, g`$\}$ and all the functions subsequently called from those functions i.e., `u_init` and `u_access`. Therefore, the association for function `h` in this case is $\{$`f, g, u_init, u_access`$\}$. Contrast this with the path-sensitive variant (see Table 1) that preserves explicit precedence constraints. CHRONICLER's implementation guarantees that using path sensitivity does not generate any additional false negatives compared to the path insensitive implementation.

After computing the associations for each call-site for a given function f, we use a frequent item set mining tool, MAFIA [8], to generate the item sets that appear frequently based on a given confidence threshold. PR-Miner [15] adopts a similar approach except that it does not focus on generating a protocol for a specific function, generating protocols for the entire program instead. The results of the comparison between CHRONICLER and the path insensitive variant is given in Figure 7. As can be seen from the figure, path sensitivity greatly reduces the number of protocols generated. In addition, we verified that these protocols form a proper subset of the protocols inferred by the path-insensitive variant. While fewer protocols are generated using our approach, it is reasonable to ask whether we have eliminated valid protocols that would be detected by the path-insensitive variant. We show in the next section that our technique does not generate additional false negatives.

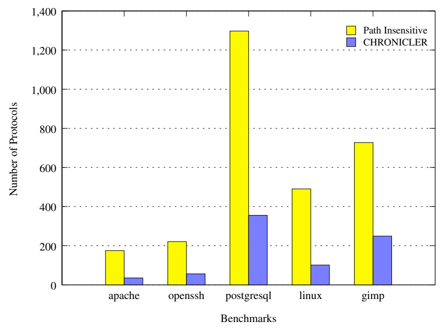
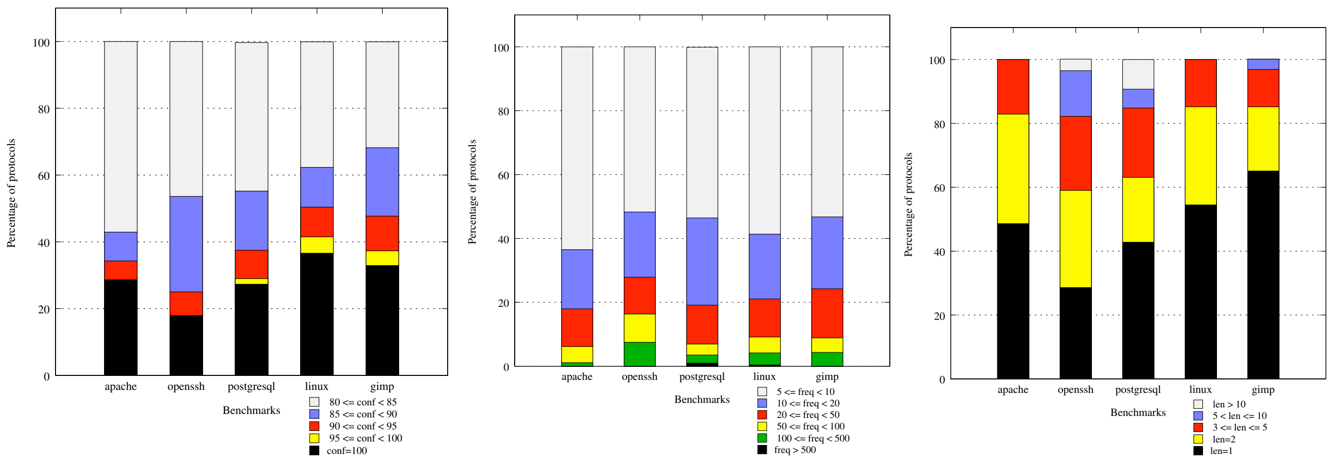


Figure 7. Comparison of CHRONICLER with a path insensitive variant.

Source	Version	LoC	CFG nodes	CFG nodes Pruned	Procedure count	Protocol count	Violation count	Constraint generation time	Sequence mining time
`apache`	2.2.0	272K	125K	9K	2067	35	16	85	4
`gimp`	2.2.9	662K	982K	108K	10192	249	133	621	27
`linux`	2.2.26	1.98M	330K	24K	7465	101	105	1270	138
`openssh`	4.1	65.9K	92K	12.6K	1242	56	31	123	7
`postgresql`	8.1.3	561K	625K	66K	8755	355	277	653	191

Table 2. Details about the benchmarks used in our experiments and the number of protocols discovered with an 80% confidence threshold. Time given in seconds.



(a) Confidence Distribution

(b) Frequency Distribution

(c) Length Distribution

Figure 6. Confidence, frequency and length parameters associated with the precedence protocols

4.2 Qualitative Assessment

While the above results address the scalability, complexity, and quantitative aspects of precedence analysis, we also consider some specific instances of protocol violations observed in the benchmarks. Though this is by no means a comprehensive study, the examples given below do provide interesting insights into the nature of the violations that can be detected using our system.

4.2.1 Case Study: Library calls in Apache

To study the quality of our results, we examined how effective our implementation was in discovering protocols associated with library calls made in `apache`, under a 100% confidence threshold. We correlate its effectiveness by comparing our results with the documentation found in the `man` pages of the corresponding library functions.

Out of the 87 library functions that are called at least more than once in `apache`, we found 26 functions that have some form of precedence requirement. Of these, 10 functions have relations that can be satisfied by any one of a set of functions of the same kind, e.g., a call to `write` can always preceded by a call to `open` or `socket`. While not a significant limitation, the current implementation only deals with precedence chains, and not arbitrary trees or graphs.

Based on the system's constraints there are potentially 16 protocols that could be inferred. CHRONICLER indeed discovered 16 protocols, of which 11 were confirmed to be valid (included in the set of 16 verified manually), and five were false positives, protocols that were not substantiated by existing documentation. These false positives all had low frequency count, and low ranking, and could be easily filtered.

Our implementation failed to detect five protocols. Of these five, CHRONICLER detected one potential bug. Every call to procedure `atexit` in the source was always preceded by `apr_app_initialize` except in one instance where it was only preceded by `apr_initialize` instead of the wrapper function `apr_app_initialize`. On Windows platforms, the documentation indicates that the wrapper call is necessary to supply appropriate command arguments and repair data formats. The path-insensitive approach also failed to detect the five protocols. We emphasize that path sensitivity is essential to derive this protocol violation.

4.2.2 Sample Violations

Hardware Bug: In Linux, we observe that calls to `__global_restore_flags` are preceded by `__global_cli` and `__global_save_flags`. The only violation of this protocol is present in procedure `init_amd` where `__global_restore_flags` is only preceded by `__global_save_flags`. Thus, the violation occurs in a platform-dependent section of the kernel.

We observe two relevant patterns in the source:

```
save_flags(flags);        __save_flags(flags);
cli();                    __cli();
...                       ...
restore_flags(flags);     __restore_flags(flags);
```

The fragment on the right is intended for non-symmetric multiprocessors. From the header file, we discover that `save_flags`, `cli`, and `restore_flags` are macros and are defined as `__global_save_flags`, `__global_cli`, and `__global_restore_flags`, respectively for symmetric multiprocessors. Otherwise, they are defined as `__save_flags`, `__cli`, and `__restore_flags`, respectively. Given the header file, the correct programming practice is to use the former code fragment. Nevertheless, as long as they are used together, problems do not arise. The potential bug is subtle, because in `init_amd`, `save_flags` is followed by `__cli` and then `restore_flags`. If this code is to be executed on an SMP, potentially degraded performance can result. This bug is difficult to observe since its occurrence is also dependent on the hardware used. We were able to discover this bug as it manifests when a complex function precedence protocol consisting of three procedures is violated.

Performance Bug: We also discover another violation in Linux that negatively impacts performance. A call to procedure `filemap_fdatawait` is not preceded by a call to procedure `filemap_fdatawrite` in `__sync_single_inode`. Procedure `filemap_fdatawrite` ensures that a block is dirty before writing the data by calling `do_writepages`. In the procedure where we detect the violation, `do_writepages` is directly called. While this violation is not necessarily erroneous, correct usage can improve performance since data writes are avoided if the page is not dirty.

Security Bug: Besides the violation described in Figure 1, CHRONICLER also discovered another protocol violation in PostGreSQL. The protocol in question requires any call to procedure `CreateComments` to be preceded by a call to `getUserId` to ensure that users have appropriate authorization before being allowed to create new comments. This protocol was violated in the procedure `CommentLargeObject`. Although this violation does not lead to a visible program error, it is an obvious security violation that permits unauthorized users to change documentation.

5 Related Work

Li and Zhou present PR-Miner [15], a tool that relies on association rule mining [1] to identify frequent program patterns. CHRONICLER fundamentally differs from PR-Miner as it ensures path-sensitivity; it is this difference that allows CHRONICLER to detect bugs such as the one presented in Figure 1, which would not be flagged by PR-Miner. Livshits and Zimmermann [16] present another mining-based technique by analyzing revision histories of programs for inferring invariants. In this paper, we also use a mining strategy, albeit with a different emphasis and methodology.

In a seminal work, Engler *et al.* [9] identify that bugs are a result of deviant behavior and statically analyze the program to detect bugs. They present an approach to detect relations between pairs of functions by exploring all possible paths. Gopalakrishna *et al.* extend this approach to detect protocols of arbitrary size in FaultMiner [12], a tool that performs sequence mining on sets of paths to generate temporal invariants. The brute-force path exploration used in these approaches limits their utility in practice to identify protocols of arbitrary size. In our experiments, we identify that on an average 50% of the protocols detected have size 3 or more (precedence length 2 or more) which cannot be detected by these approaches scalably. By folding constraints at join points and using memoization techniques for procedures, we are able to successfully apply our approach to large software systems.

There are several other salient approaches that address the problem of inferring specifications. Kremenek *et al.* [14] classify functions as claiming or returning ownership and infer specifications based on factor graphs. This semantic-based approach is complementary to program analysis-based systems like CHRONICLER. Weimer and Necula [22] present an automatic specification mining technique that uses information about exceptions and errors to identify temporal safety rules, and provide elaborate experiments to show the scalability and correctness of their approach. They compare their scheme to four existing miners [3, 4, 9, 23]. It is not clear how their approach can be extended to extract protocols of arbitrary complexity.

In [4], Ammons *et al.* perform specification mining by summarizing frequent interaction patterns as state machines that capture temporal and data dependencies when interacting with API's or abstract data types. Yang *et al.* [25] provide a scalable solution by using dynamic inference techniques that are effective in the presence of imperfect traces. Ernst *et al.* [10] present Daikon, a tool for dynamically detecting invariants in a program. These techniques are all based on a dynamic runtime analysis. Even though dynamic techniques can be more precise as inference is made based on observed program behavior, the dependence on test inputs to provide program coverage limits their utility. As with any static analysis technique, our approach is not dependent on test inputs for precision.

Function summaries have been used in a variety of application contexts (e.g., testing [11], verifying multi-threaded programs [19]). In [11], Godefroid presents an approach that uses function summaries to encode the test results in the form of pre- and post-conditions. In [19], Qadeer *et al.* present an approach to encode function summaries for multi-threaded programs. In this work, we propose using function summaries for effectively inferring precedence protocols using an inter-procedural static analysis.

6 Conclusion

This paper focuses on the problem of inferring function precedence protocols, and identifying when these protocols are violated. It presents a scalable and effective approach to extract protocols of arbitrary length. The proposed technique is path sensitive and has been validated on large real-world systems. We are able to detect several subtle protocols in our benchmarks, and identify non-trivial violations. Some of the interesting problems stemming from this work, which are topics of current investigation, include automatically identifying the correctness of an inferred protocol (currently the protocols are validated manually), integrating dataflow information to potentially improve precision, and associating confidence intervals with violations to provide additional refinement.

Acknowledgements: This work is supported in part by the National Science Foundation under grant STI 501-1398-1078. We also thank Cristian Ungureanu for useful initial discussions, Rajeev Gopalakrishna for providing the control flow graphs for our preliminary experiments, GrammaTech Inc. for providing `Codesurfer`, Xiangyu Zhang for insightful comments on an earlier version of this paper and the anonymous reviewers for their valuable suggestions.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 12–15 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, pages 3–14, 1995.

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[4] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, 2002.

[5] P. Anderson, T. Reps, and T.Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. on Software Engineering*, 29(8):721–733, August 2003.

[6] T. Ball, V. Levin, and F. Xie. Automatic creation of environment models via training. In *TACAS*, 2004.

[7] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software, LNCS 2057*, pages 103–122, May 2001.

[8] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu. Mafia: A performance study of mining maximal frequent itemsets. In *Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003.

[9] D. Engler, D. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[10] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[11] P. Godefroid. Compositional dynamic test coverage. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'07)*, Jan 2007.

[12] R. Gopalakrishna. Improving software assurance using lightweight static analysis. *PhD Thesis, Purdue University*, 2006.

[13] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[14] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Seventh USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006.

[15] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of ESEC-FSE*, pages 306–315, 2005.

[16] B. Livshits and T. Zimmermann. Dynamine: a framework for finding common bugs by mining software revision histories. In *Proceedings of ESEC-FSE*, 2005.

[17] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of PLDI '05*, pages 48–61, 2005.

[18] Steven Muchnick. *Advanced Compiler Design and Implementation*. Academic Press and Morgan Kaufmann Publishers, first edition, 1997.

[19] S. Qadeer, S.K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'04)*, Jan 2004.

[20] S. Reiss and M. Renieris. Encoding program executions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[21] N. Tansalark and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the Conference on Object-oriented programming systems, languages, and applications*, pages 413–430, 2006.

[22] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of Tools and Algorithms For The Construction And Analysis Of Systems (TACAS)*, pages 461–476, April 2005.

[23] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis, ISSTA*, 2002.

[24] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–363, 2005.

[25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE'06)*, May 2006.