Static Specification Inference Using Predicate Mining

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

Abstract

The reliability and correctness of complex software systems can be significantly enhanced through well-defined specifications that dictate the use of various units of abstraction (e.g., modules, or procedures). Oftentimes, however, specifications are either missing, imprecise, or simply too complex to encode within a signature, necessitating specification inference. The process of inferring specifications from complex software systems forms the focus of this paper. We describe a static inference mechanism for identifying the preconditions that must hold whenever a procedure is called. These preconditions may reflect both dataflow properties (e.g., whenever p is called, variable x must be non-null) as well as control-flow properties (e.g., every call to p must be preceded by a call to q). We derive these preconditions using an inter-procedural path-sensitive dataflow analysis that gathers predicates at each program point. We apply mining techniques to these predicates to make specification inference robust to errors. This technique also allows us to derive higher-level specifications that abstract structural similarities among predicates (e.g., procedure p is called immediately after a conditional test that checks whether *some* variable *v* is non-null.)

We describe an implementation of these techniques, and validate the effectiveness of the approach on a number of large open-source benchmarks. Experimental results confirm that our mining algorithms are efficient, and that the specifications derived are both precise and useful – the implementation discovers several critical, yet previously, undocumented preconditions for well-tested libraries.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Documentation; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification—Statistical Methods

General Terms Algorithms, Documentation, Verification

Keywords specification inference, preconditions, predicate mining, program analysis

PLDI'07 June 11–13, 2007, San Diego, California, USA.

1. Introduction

Well-defined specifications can significantly enhance the reliability and correctness of complex software systems. When available, they can be used to verify correctness of libraries and device drivers [4, 7, 21, 36], enable modular reuse [29], and guide testing mechanisms toward bugs [13, 18]. When specifications are provided by the user, type systems [16, 17, 11], model checking [21, 7], typestate interpretation [24, 20], and other related static analyses [36] can be used to check whether implementations satisfy necessary invariants.

Often, specifications are easy to define (e.g., procedure p must always be called after data structure d is initialized), or are well-documented (e.g., `pthread_mutex_init` must be present on all program paths reaching a call to `pthread_mutex_lock`). In many cases, though, specifications are not known, and even when available, are often informal, imprecise, or incomplete. This is especially true for complex system software libraries. For example, consider the function `BN_is_prime` in the `openssl` library, a widely-used secure socket layer implementation. The function's signature is complex, taking five arguments, including a callback procedure. It returns true if its first argument, which is a pointer to a `bignum` object, is prime. Its documentation, however, makes no assertions about the expected structure of its arguments; for example, it does not specify the function's behavior if a null pointer value is supplied as the first argument. Clients must therefore either examine its implementation to determine the appropriate constraints on arguments, or proactively perform error checks before making the call. Both approaches have obvious drawbacks, and neither work if the client is not even aware that a potential issue exists.

One way to determine the appropriate conditions under which `BN_is_prime` can be safely called is to examine the collection of calls made to the function from other clients. The underlying hypothesis is that the confidence and specificity of a property inferred for `BN_is_prime`, is reflected in its satisfaction at various call sites. However, *manually* identifying call sites to `BN_is_prime` over a large number of client programs, and examining how the arguments are defined and used prior to the call, is generally infeasible. On the other hand, *automated* techniques for correlating invariants across different call sites to the same procedure must take into account the possibility that programs may contain bugs which can mask real invariants. For example, it may be the case that in some calls to `BN_is_prime`, the first argument is incorrectly *not* checked prior to the call. Failure to perform this check may not necessarily lead to an error if the argument values at these call sites serendipitously happen to be non-null. Automated techniques must also be able to distinguish between invariants that are significant enough to be included as part of the function's specification from those that, while possibly true, are irrelevant. For example, there may be several properties that hold at each call-site to `BN_is_prime` related to

global or temporary variables that are inconsequential to its specification.

In this paper, we consider the problem of statically inferring specifications *transparently* without requiring programmer annotations. Specifically, we consider the problem of generating specifications that define *preconditions* for procedures – predicates that must always hold when the procedure is called. We consider two important classes of preconditions: control-flow predicates that define precedence properties among procedures (e.g., a call to `fgets` is always preceded by a call to `fopen`), and data-flow predicates that capture dataflow properties associated with variables (e.g., whenever `fgets` is called, pointer `fp` must not be null).

We define an inter-procedural, path-sensitive static analysis that identifies a collection of constraints whose solution defines potential preconditions. If procedure p has precondition π, it means that π holds on all *calls* to p. To compute preconditions, our analysis collects a predicate set along each distinct path to each call-site. To manage the size of this set, intersections of predicate sets are constructed at join points where distinct paths merge. Predicates computed within a procedure are memoized and are used to compute preconditions that capture inter-procedural control and dataflow information. To compute the preconditions of a procedure p, we consider the intersection of the predicate sets at each call-site to p.

There are several significant design issues that need to be resolved for the derived preconditions to have any practical significance. We observe that using simple set intersection on predicates is too fragile to yield interesting specifications in general. This is because the predicates generated are insufficiently abstract (e.g., "at a call to procedure p, variable `x` bound in p is read, and the contents of locations `a` and `b` allocated in p are compared."). The intersection of any set S with a set containing just these predicates would be non-empty only if S contained *identical* predicates, reflecting the same operations on the same variables and locations. To relax this limitation, we examine techniques that allow us to define *structural similarities* among predicate sets. Such similarities enable preconditions that specify properties which are abstracted over variable names, references, and values. We are thus able to define preconditions that define more abstract properties such as "procedure q is called whenever *some* integer variable v is greater than zero, and the contents of *some* pair of locations l_1 and l_2 holding a value of type τ are equal."

While the use of intersection guarantees safety by ensuring that derived preconditions for a procedure hold at all call-sites, it is not a robust mechanism in the presence of errors. An error that causes a predicate to be omitted along some path leading to a call to procedure p would result in the predicate not being included as part of p's preconditions. To address this concern, we employ frequent itemset and sequence mining on the predicates computed at each call-site to p, and use the predicates that are most frequently occurring as the preconditions for p. Like other mining-based approaches, we assume that errors violating invariants occur *infrequently*, thus making mining a feasible strategy to filter such deviations from the generated specifications.

It is our approach to these issues, and the kinds of specifications generated as a result, that distinguish our work from previous efforts that have used mining techniques (both dynamic [4, 14, 35, 38] and static [23, 26, 28, 13]) to extract and validate program properties. While dynamic mining techniques can be used to generate specifications, the integrity of the specification depends upon the comprehensiveness of the input data. On the other hand, prior static approaches have not been well-integrated within a program analysis framework, and therefore are not effective in generating useful preconditions. Our primary contribution is this systematic integration of data flow analysis with scalable mining algorithms.

This paper makes the following additional technical contributions:

1. **Precondition Inference:** We present and formalize a new path-sensitive inter-procedural static analysis for inferring preconditions for procedures *transparently* with no programmer annotations, profiling, or instrumentation.
2. **Robust Specifications**: We describe the use of mining techniques to generate correct specifications even for programs that may have subtle bugs that lead to necessary invariants being erroneously omitted along certain paths. Mining also provides a way to compensate for imprecision introduced by the static analysis that would ordinarily result in omitting valid predicates from a precondition.
3. **Experimental Evaluation:** We demonstrate the practicality of our techniques to large open-source C programs, and provide a detailed quantitative and qualitative assessment of the effectiveness of our approach. Our results show that the analysis is (a) selective – the number of elements comprising derived preconditions tend to be small (less than five on average); (b) precise – the analysis derives approximately 78% of documented specifications to library calls made by `openssh`; and (c) useful – we discovered several bugs in the benchmarks that exist because of failure to adhere to derived specifications.

2. Motivating Example

We motivate our approach using a real-world example – deriving a specification for the `bind` system call in the Linux `socket` library. While the application of our approach is in deriving specifications for undocumented procedures, it is illustrative to demonstrate the technique for a procedure such as `bind`, which has a well-documented interface. The `bind` system call takes three parameters, *viz.,* a socket descriptor (type: `int`), the local address to which the socket needs to bind (type: `struct sockaddr *`), and the length of the address (type: `socklen_t`). For a stream socket to start receiving connections, it needs to be assigned to an address, which is achieved by using `bind`. Summarizing the documentation, the necessary conditions that must hold before `bind` can be called are:

1. A `socket` system call must have occurred.
2. The return value of `socket` must have been checked for validity.
3. The address (second parameter to `bind`) corresponds to a specific address family (e.g., AF_UNIX, AF_INET).

Ideally, our goal is to obtain the above information by tracking various calls to `bind` in the source. Figure 1 shows code fragments of two procedures (out of eight, total) that invoke the `bind` system call in `openssh-4.2p1`.

Figure 1(a) shows a code fragment from the file `sshd.c` where `bind` is invoked from `main`. Before the call to `bind`, as per the documented requirements, observe that there is a call to `socket` on line 1287. The returned value `listen_socket` is checked to ensure that it is a valid descriptor, and the address is set (lines 1075 and 1272). In fact, in a convoluted chain, the procedure `fill_default_server_options` in `main` invokes `add_listen_addr`, which in turn invokes `add_one_listen_addr` where the address that is eventually used in `bind` is set. Apart from

```
870 main(...)
883   struct addrinfo *ai;
918   initialize_server_options(&options);
1075  fill_default_server_options(&options);
1272  for (ai=options.listen_addrs; ai;ai=ai->ai_next) {
1273    if(ai->ai_family != AF_INET &&
                      ai->ai_family != AF_INET6)
1274       continue;
1275    if (num_listen_socks >= MAX_LISTEN_SOCKS)
          ...
1278    if ((ret = getnameinfo(...))) {
          ...
1287    listen_sock = socket(ai->ai_family,...);
1289    if (listen_sock < 0) {
          ...
1294    if (set_nonblock(listen_sock) == -1) {
          ...
1302    if (setsockopt(...) == -1)
1304       error(''setsockopt SO_REUSEADDR: ...'');
1309    if (bind(listen_sock, ai->ai_addr,
                           ai->ai_addrlen) < 0) {
          ...
```

(a) sshd.c

```
991 ssh_control_listener(void)
993   struct sockaddr_un addr;
997   if (options.control_path == NULL ||
998      options.control_master == SSHCTL_MASTER_NO)
999      return;
1003  memset(&addr, 0, sizeof(addr));
1004  addr.sun_family = AF_UNIX;
1005  addr_len = offsetof(...)
1008  if (strlcpy(addr.sun_path, options.control_path,
1009   sizeof(addr.sun_path)) >= sizeof(addr.sun_path))
1010     fatal(''ControlPath too long'');
1012  if ((control_fd = socket(PF_UNIX,
                              SOCK_STREAM, 0)) < 0)
1013     fatal(...);
1015  old_umask = umask(0177);
1016  if (bind(control_fd, (struct sockaddr*)&addr,
                                   addr_len) == -1) {
        ...
```

(b) ssh.c

Figure 1. Code fragments of two different call sites to **bind** in openssh-4.2p1.

sshd.c		ssh.c	
Variables	Attributes	Variables	Attributes
`(*ai).ai_addrlen`	{(**arg**(3), `bind`)}	`addr.sun_family`	{(:=, 1)}
`ai`	{(:=, `options.listen_addrs`),(≠, 0) }	`addr_len`	{(:=, **res**(`strlen`)), (**arg**(3), `bind`)}
`inetd_flag`	{(=, 0)}	`old_umask`	{(:=, **res**(`umask`))}
`listen_sock`	{(:=,**res**(`socket`)), (≥, 0), (**arg**(1), `bind`), (**arg**(1), `setsockopt`)}	`control_fd`	{(:=, **res**(`socket`)), (≥, 0), (**arg**(1), `bind`)}
`num_listen_socks`	{(<, 16)}	`options.control_master`	{(≠, 0)}
`ret`	{(:=, **res**(`getnameinfo`)), (=, 0)}	`options.control_path`	{(≠, 0)}

Table 1. A subset of predicates associated with the **bind** calls shown in Figure 1.

these known *requirements*, other operations dependent on the application context are also performed (e.g., the family of the address is checked in line 1273, the `num_listen_socks` is checked in line 1275, etc.). By observing just a single use of `bind` alone, we can generate some properties on the required operations before `bind` is called.

Table 1 shows the subset of properties generated for the corresponding `bind` call. For example, we observe a property where a variable `listen_sock` is assigned the return value of `socket`, has a value greater than or equal to 0 and is the first parameter in calls to `setsockopt` and `bind`. As explained above, these properties form some of the preconditions for calls to `bind`. However, not all properties generated before this `bind` call need to hold *always* before any other call to `bind`. For example, `ret` is assigned the return value of `getnameinfo` and is equal to 0 before the `bind` call. This property may be relevant in the context of calls to `bind` in `sshd.c`, but may not be relevant in calls made within other files. Unfortunately, simply examining this single call without any *a priori* knowledge of `bind`'s behavior would not permit us to discard this property from its specification.

To improve precision, we collect properties from other call sites to `bind`. Figure 1(b) presents one such call site in procedure `ssh_control_listener` in `ssh.c`. For this call, we obtain properties that include the known *requirements* (see lines 1004, 1005, 1012) and also shown in Table 1. We also obtain other irrelevant operations (e.g., the control path is checked at line 997, size of path checked in 1008, etc.). Based on the properties here and the properties previously obtained with respect to the `bind` call in Figure 1(a), an *intersection* of the derived properties can be computed. By repeated application of this process to each call to `bind` at other call-sites, we obtain the necessary operations that *must* be performed before every call to `bind`.

To summarize the example, observe that deriving the desired preconditions using *intersection* must account for the fact that (a) the names of relevant variables in the two files are not comparable (e.g., `listen_sock` in `sshd.c` and `control_fd` in `ssh.c`); (b) operations relevant to the bind call (e.g., `listen_sock` $\geq$ 0 in `sshd.c` and `((control_fd = ...)` $\geq$ `0)` in `ssh.c`) are interspersed with irrelevant operations; (c) the types of corresponding parameters to `bind` before casting are different (`struct sockaddr *` in `sshd.c` and `struct sockaddr_un *` in `ssh.c`); (d) there is no fixed order of calls to procedures setting the address family and the call to `socket` in the two files and (e) there can be different number of attributes associated with the corresponding variables across call-sites (e.g., `listen_sock` is used as a parameter in `setsockopt` whereas `control_fd` does not have any such attribute.).

3. Specification Language

We formalize our informal discussion above by defining a simple, call-by-value language equipped with first-class procedures and

references. Superscripts on expressions denote labels that are used in defining our analysis. The exact semantics for the language is standard and omitted here.

Informally, a `let`-expression binds x in e, $\lambda\, y.e'$ constructs a lexically-scoped first-class procedure, $y(z)$ denotes call-by-value application, $\texttt{ref}(y)$ constructs a first-class reference cell that holds the value denoted by y, and $\texttt{deref}(y)$ extracts the value of the cell bound to y. The expression $(\texttt{set}\ x := y^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}$ assigns the value of y to the cell bound to x, and continues with e. Bound and free variables are defined as usual. A program P is a closed expression, and $e^{\ell} \in P$ is true if e^{ℓ} is a subexpression of P.

In addition to the usual assumption that bound variables are distinct from free variables in different expressions, we also assume that all bound variables in a program are distinct. The *last* variable of an expression, which yields the expression's value, is defined as follows:

$$\begin{aligned} last(x^{\ell}) &= x^{\ell} \\ last((\texttt{let}\ x = t^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}) &= last(e^{\ell}) \\ last((\texttt{set}\ x := y^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}) &= last(e^{\ell}) \end{aligned}$$

Our analysis is defined in two steps. First, we compute a flow analysis for the program, F, that associates with every variable and label, a set of *abstract values*. An abstract value is either a constant, a label corresponding to the definition point of a procedure (*abstract procedure*) or reference (*abstract location*), or a primitive operation paired with the abstract values of its arguments. Thus, given variable x, $F(x)$ (or $F(\ell)$, if given label ℓ) defines the set of procedures, constants, references, and primitive operations that x (or the expression with label ℓ) can denote during execution of the program. We do not present details of the analysis here, but any monovariant flow analysis in the spirit of [30, 33] suffices for our purpose.

A judgment is a three-place relation on specification maps, flows, and expressions. Thus, the judgment $\Delta \models_F e^{\ell}$ is read "Assuming a flow analysis F, expression e^{ℓ} has the preconditions defined by $\Delta(\ell)$." Given a flow function F, and program P, we are interested in the least specification map Δ for which the judgment holds.

Specification inference is defined by a collection of inference rules (see Figure 2) that leverages the result of the flow analysis. Each rule is of the form:

$$\frac{c_1, \ldots, c_n}{\Delta \models_F e},$$

where the consequent defines a judgment whose validity depends upon the satisfiability of the constraints defined by the antecedent. The constraints impose restrictions on the structure of the specification map Δ, a map that identifies a set of preconditions with every program point.

A precondition π of an expression e defines an action or predicate that *must* hold prior to e's execution. Our analysis tracks a number of such actions; these actions are defined with respect to the abstract values computed for each expression in the program by the flow analysis. Thus, an action of the form $\mathsf{read}(\ell, \hat{v})$ asserts that a reference created at label ℓ holding the abstract value $\hat{v}$ is read; $\mathsf{write}(\ell, \hat{v})$ asserts a similar condition for reference assignment; and, $\mathsf{alloc}(\ell, \hat{v})$ holds if in an expression $\texttt{ref}(z)^{\ell} \in P$ and $F(z) = \hat{v}$. In the same vein, $\mathsf{bind}(x, \hat{v})$ is true whenever variable x is bound to e^{ℓ} and $F(\ell)$ is $\hat{v}$, and $\mathsf{cbind}(x, \hat{v})$ is used to express predicates that reflect if-splitting of flow values across conditionals; finally, $\mathsf{call}(\ell_1 \leftarrow \ell_2)$ is used to capture control-flow precedence relationships among procedure calls – it holds whenever a procedure with label ℓ_2 is invoked *after* an invocation of a procedure with label ℓ_1, with no intervening invocation of any other procedure.

The rules for expressions that bind constants and primitive operations are straightforward. The preconditions of the expression in the `let`-body within which the binding occurs includes the preconditions of the `let` expression, as well as a precondition that reflects the existence of the new binding. If a variable is bound in a `let`-expression to the result of a call to a primitive operation, the preconditions of the expression in the `let`-body must include this action; the values of the arguments to the primitive are approximated by the abstract values of the operation's arguments as determined by the flow analysis.

A reference binding induces a precondition on the `let`-body that includes both the binding as well as a predicate that captures the reference creation. Since references are first-class, a variable occurrence may be bound to many different references during its lifetime. In an expression of the form, $(\texttt{let}\ x = \texttt{deref}(y)^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}$, consider the set of references that y may be bound to (defined by $F(y)$). Each element in this set contains a label ℓ corresponding to a reference expression $\texttt{ref}(z)$ found in the program. The precondition for e^{ℓ} must therefore include predicates that reflect the potential read of each such location, and predicates that reflect the binding of x to the contents of these locations. Assignment expressions are defined similarly, with write predicates replacing reads as a consequence of the operation. The preconditions following a conditional include the intersection of the specification sets of the two branches; within these branches, an action that reflects the value of the Boolean guard is included as part of the precondition associated with the respective branches.

We now describe the rules dealing with procedure abstraction and call. The precondition associated with the procedure body is defined as the intersection of a collection of sets, each of which represents the specifications extant at a specific (distinct) call point to the procedure. Thus, the specifications defining the entry to a procedure reflect the common preconditions extant at every call point to the procedure. For example, the specification associated with the entry to the procedure body defines a predicate that relates the formal parameter to an abstract value. This value is constructed as the intersection of the abstract values (set of labels, constants, etc.) of the actual parameters to the procedure. Similarly, the intersection of the set of preconditions that exist at each such call defines the smallest set of predicates that is guaranteed to hold whenever the procedure is called.

A procedure call $y(z)$ is defined similarly. Its definition relies on an auxiliary procedure Λ that given the label of an expression $e^{\ell} \in P$ returns the label of the closest enclosing λ, if one exists, and the distinguished label ℓ_{main}, otherwise. If the set of procedures that y may be associated with is P_y (as determined by our flow analysis), then the intersection of the specifications extant upon *exit* from each procedure $p \in P_y$ defines the conditions present upon exit from the call guaranteed to hold for all procedures p that may be invoked at this call. Observe that these rules are slightly different from typical static analyses that would consider the definition of the procedure independently from its call-sites. This is because preconditions that hold at the entry to a procedure p depend upon the conditions extant at all call-sites to p; similarly, the invariants that hold upon completion of a call depend upon the invariants extant at the return point of all procedures that could be invoked at that call.

As currently defined, the preconditions associated with each program point are constructed by simple unions and intersections of abstract value sets computed by an inter-procedural dataflow analysis. It is straightforward to see that the predicates computed represent a conservative summary of the information present in the flow function.

SYNTAX:

$$
\begin{array}{rcl}
e \in \mathit{Exp} & ::= & x^{\ell} \mid \\
 & & (\texttt{let}\ x = t^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'} \mid \\
 & & (\texttt{set}\ x := y^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'} \\
t \in \mathit{Term} & ::= & c \mid \lambda x.e \mid x(y) \mid \\
 & & (\texttt{if}\ x\ \texttt{then}\ e_1^{\ell_t}\ \texttt{else}\ e_2^{\ell_f}) \mid \\
 & & \texttt{ref}(x) \mid \texttt{deref}(x) \mid \\
 & & \texttt{op}(x_1, \ldots, x_n)
\end{array}
$$

DOMAINS:

$$
\begin{array}{rcrl}
F & \in & \mathit{Flow} & = \mathit{Var} + \mathit{Label} \rightarrow \mathit{AVal} \\
\hat{v} & \in & \mathit{AVal} & = \mathcal{P}(\mathit{Label} + \mathit{Constant} + \\
 & & & \quad \mathit{Op}(\mathit{AVal} \times \ldots \times \mathit{AVal})) \\
\Delta & \in & \mathit{SpecMap} = & \mathit{Label} \rightarrow \mathcal{P}(\mathit{Pred}) \\
\Lambda & \in & \mathit{ProcMap} = & \mathit{Label} \rightarrow \mathit{Label} \\
\pi & \in & \mathit{Pred} & = \mathsf{read}(\mathit{Label}, \mathit{AVal}) + \mathsf{write}(\mathit{Label}, \mathit{AVal}) + \\
 & & & \quad \mathsf{alloc}(\mathit{Label}, \mathit{AVal}) + \mathsf{bind}(\mathit{Var}, \mathit{AVal}) + \\
 & & & \quad \mathsf{cbind}(\mathit{Var}, \mathit{AVal}) + \mathsf{call}(\mathit{Label} \leftarrow \mathit{Label})
\end{array}
$$

$$\frac{\Delta(\ell') \cup \{\mathsf{bind}(x, \{c\})\} \subseteq \Delta(\ell)}{\Delta \models_F (\texttt{let}\ x = c^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{\Delta(\ell') \cup \{\mathsf{bind}(x, \mathsf{op}(F(x_1), \ldots, F(x_n)))\} \subseteq \Delta(\ell)}{\Delta \models_F (\texttt{let}\ x = \mathsf{op}(x_1, \ldots, x_n)^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{\Delta(\ell') \cup \{\mathsf{alloc}(\ell_1, F(y)), \mathsf{bind}(x, \ell_1)\} \subseteq \Delta(\ell)}{\Delta \models_F (\texttt{let}\ x = \texttt{ref}(y)^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{S = \{\mathsf{read}(\ell_i, F(\ell_i)), \mathsf{bind}(x, F(\ell_i)) \mid \ell_i \in F(y) \wedge \texttt{ref}(z)^{\ell_i} \in P\} \quad \Delta(\ell') \cup \{S\} \subseteq \Delta(\ell)}{\Delta \models_F (\texttt{let}\ x = \texttt{deref}(y)^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{S = \{\mathsf{write}(\ell_i, F(y)) \mid \ell_i \in F(x), \texttt{ref}(z)^{\ell_i} \in P\} \quad \Delta(\ell') \cup \{S\} \subseteq \Delta(\ell)}{\Delta \models_F (\texttt{set}\ x := y^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{\begin{array}{c}\Delta(\ell') \cup \{\mathsf{cbind}(y, \mathsf{true})\} \subseteq \Delta(\ell_t) \\ \Delta(\ell') \cup \{\mathsf{cbind}(y, \mathsf{false})\} \subseteq \Delta(\ell_f) \\ \Delta\ell' \cup (\Delta(\ell_t) \cap \Delta(\ell_f)) \cup \{\mathsf{bind}(x, F(\ell_1))\} \subseteq \Delta(\ell)\end{array}}{\Delta \models_F (\texttt{let}\ x = (\texttt{if}\ y\ \texttt{then}\ e_1^{\ell_t}\ \texttt{else}\ e_2^{\ell_f})^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{\begin{array}{c}\bigcap\{\Delta(\ell_i) \mid (\texttt{let}\ z_i = x_i(y_i)\ \texttt{in}\ e_i)^{\ell_i}, \ell_1 \in F(x)\} \subseteq \Delta(\ell_1) \\ \{\mathsf{bind}(w, \hat{v}) \mid (\texttt{let}\ z_i = x_i(y_i)\ \texttt{in}\ e_i)^{\ell_i}, \ell_1 \in F(x), \hat{v} = \cap(F(y_i))\} \subseteq \Delta(\ell_b) \\ \Delta(\ell') \cup \{\mathsf{bind}(x, \ell_1)\} \subseteq \Delta(\ell)\end{array}}{\Delta \models_F (\texttt{let}\ x = (\lambda w.e_b^{\ell_b})^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

$$\frac{\begin{array}{c}S = \{\ell_j \mid \ell_i \in F(y) \wedge (\lambda w_i.e_i)^{\ell_i} \in P \wedge e_j^{\ell_j} = \mathit{last}(e_i)\} \\ \Delta(\ell') \cup \{\mathsf{call}(\Lambda(\ell_1) \leftarrow \ell_i) \mid \ell_i \in F(y) \wedge (\lambda w_i.e_i)^{\ell_i} \in P\} \cup \{\bigcap\{\Delta(\ell_j) \mid \ell_j \in S\} \cup \{\mathsf{bind}(x, S)\} \subseteq \Delta(\ell)\end{array}}{\Delta \models_F (\texttt{let}\ x = y(z)^{\ell_1}\ \texttt{in}\ e^{\ell})^{\ell'}}$$

Figure 2. Specification inference via flow analysis.

```
let r = λ z. ref(z)^ℓ1
    ...
    g1 = λ c1. let y1 = λ w. ref(w)^ℓ2
                   y2 = r(c1)
                   y3 = y1(y2)
                   y4 = deref(y3)
                   y5 = deref(y2)
         in ... op1(y2, y3)
            ... op2(y4, y3)
            ... set y2 := c1 in f(...)
```

```
g2 = λ c2. let x1 = ref(c2)^ℓ3
               x2 = λ w'. ... ref(w')^ℓ4 ...
               x3 = ref(c3)^ℓ5
               x4 = x2(c4)
     in ... op2(x1, x4)
        ... op1(x3, x1)
        ... set x3 := c3 in ... f(...)
```

Figure 3. A program fragment illustrating the need for structural matching of predicates. Syntactic sugar is used to simplify the examples.

There are two interesting issues to note about the analysis. First, a predicate is recorded as part of a precondition at a program point *only if* the predicate occurs on all paths to that point. Consider a module whose designer expects certain preconditions to hold when procedures defined within the module are called. Our analysis would certainly infer these preconditions for correctly written programs, but fail to identify the desired specification in the presence of errors that result in the omission of some of these legitimate predicates. The ability of the analysis to derive meaningful specifications in the presence of errors is consequently poor. There is an obvious conundrum here, given that the inferred specifications are derived from a program source that potentially contains bugs, and can thus potentially compromise the integrity of the specifications themselves.

Second, the intersection of precondition sets fails to consider structural equivalence among predicates. In particular, our specification language does not permit predicates to be abstracted over an arbitrary set of locations, names, or constants. To illustrate this, consider the program fragments shown in Figure 3. We are interested in the specification that should be inferred for the entry to procedure `f` based on the preconditions extant at its two call-sites in $\texttt{g}_1$ and $\texttt{g}_2$. Suppose $\texttt{g}_1$ and $\texttt{g}_2$ are called from the following expression:

```
if pred then g1(c) else g2(c)
```

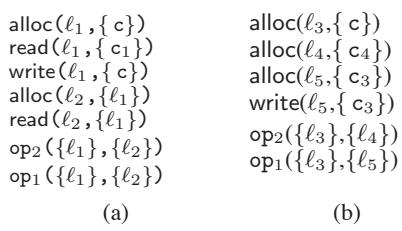


Figure 4. A subset of the preconditions that hold prior to the call to procedure `f` in procedure g_1 (a) and procedure g_2 (b).

At the calls to `f` in procedures g_1 and g_2, there are a number of preconditions that hold. Ignoring predicates that describe variable bindings, the most interesting are those related to abstract locations ℓ_1 and ℓ_2 (see Figure 4(a)) allocated and accessed by procedure g_1 and abstract locations ℓ_3,ℓ_4, and ℓ_5 accessed by procedure g_2 (see Figure 4(b)).

Based on the structure of the rules, we would conclude that no interesting preconditions exist that are common to both calls since the sets of locations manipulated by the two procedures are disjoint. This is clearly overly conservative.

For example, it is the case that prior to both calls (i) two locations are allocated and used in operation op_2 (ℓ_1 and ℓ_2 in procedure g_1, and ℓ_3 and ℓ_4 in procedure g_2), and (ii) the contents of one of these locations (ℓ_1 in g_1 and ℓ_3 in g_2) holds the constant `c`. By "unifying" ℓ_1 and ℓ_3, and ℓ_2 and ℓ_4, we derive the preconditions for `f`: "there exist a pair of locations (call them *a* and *b*) such that *a* and *b* are used as arguments in an operation op_2, and hold the constant `c`".

Surprisingly, by considering an alternative mapping of locations in the two calls, we can deduce another equally valid specification. Prior to both calls it is also the case that (i) two locations are allocated (ℓ_1 and ℓ_2 in g_1, and ℓ_3 and ℓ_5 in g_2) and used in operation op_1; and (ii) one location is written with a constant (ℓ_1 in g_1 and ℓ_5 in g_2).

To extract commonalities such as those among sets of predicates extant at the two calls requires us to match locations, names, and constants across these different sets. As the example illustrates, there are potentially many such matches that can be constructed. Of course, some commonalities could be extracted by examining the body of `f`, but this would compromise scalability and modularity. Other commonalities can be derived by examining `f`'s signature, the types of values stored in these locations, etc. We exploit some of these heuristics in our implementation.

As we show in the next section, simply enumerating the set of all possible matches over the predicate sets used to define preconditions is infeasible. We therefore consider an alternate strategy to identify matches among the precondition sets computed at different call-sites (or among procedures called at the same call-site) inspired by data mining techniques. As we shall discuss, these approaches sacrifice optimality for scalability and efficiency; our experimental results reveal that they yield surprisingly valuable specifications even in the presence of complex control- and data-flow, even in the presence of bugs that result in invariants being omitted along certain program paths.

4. Extracting specifications

We use mining as a tool for deriving *common* properties across multiple call sites instead of ensuring that properties hold across *each* call site. The reasons for adopting such a strategy are two-fold. First, even when programs are well-tested, they are not necessarily free from errors. Hence, by imposing the strong requirement that a property must hold at each call-site in order to be a precondition candidate, we may omit preconditions that otherwise might have been detected. Second, by identifying frequently occurring properties, we can detect call-sites where the preconditions do not hold. If the property is indeed a valid precondition, its absence at certain call-sites *may* point to an error. To motivate our problem further, we consider two examples taken from our benchmark suite.

Consider the code fragment in Figure 5(a). 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, which suggests that this might be a feasible control predicate precondition. However, in one specific instance of the call to `ri_BuildQueryKeyFull` at line 250, the rule is not satisfied, since 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.

Figure 5(b) shows the code fragment of the procedure `add_listen_addr` found in file `serverconf.c` from `openssh` 4.2p1. This procedure is called from the procedure `fill_default_server_options`, which in turn precedes a call to `bind`. In the body of the procedure, there are several calls to `add_one_listen_addr`, which is responsible for setting an address family, eventually supplied as the second argument to `bind`. There are several ways in which a call to `add_one_listen_addr` can take place; notably, when `port == 0` and `options->num_ports` is less than one, the call does not happen. It so happens that lines 403 and 404 reveal that this situation cannot arise, and thus the loop must be executed at least once whenever `port` is zero. Unfortunately, in the absence of a theorem prover or model checker [15] that can assert `options->num_ports` is always greater than zero at line 407 because of the operations performed at lines 403 and 404, we must conclude that it is not always the case that `add_one_listen_addr` is called from `add_listen_addr`, and thus, the second argument to `bind` need not always be set to a specific address family. Even if there are no bugs in the program, limitations of the analysis in determining a precise set of feasible paths can be overcome using mining techniques. By mining the set of predicates computed along different paths to `bind` calls, we discover that it is only along one path (namely the infeasible one described above) that the second argument to bind is not set to an address family. By setting confidence thresholds appropriately, the absence of this predicate would not be considered a critical omission, and the predicate asserting that the second argument to bind is always set would be recorded as part of `bind`'s preconditions.

4.1 Mining Strategies

Recall that our analysis collects control-flow and dataflow predicates. The elements in a set of dataflow predicates have no ordering relationship among one another. Control-flow predicates, on the other hand, do reflect a specific ordering: each element represents a procedure call, and the order of calls defines a precedence relation. We use frequent itemset mining to derive preconditions for dataflow predicate sets, and subsequence mining to derive preconditions for control-flow predicates.

```
181 RI_FKey_check(PG_FUNCTION_ARGS)
182 {
199   ri_CheckTrigger(...);
210   pk_rel = heap_open(...);
248   if (tgnargs == 4)
249   {
250      ri_BuildQueryKeyFull(...);
294   }
296   match_type = ri_DetermineMatchType(...);
298   if (match_type == RI_MATCH_TYPE_PARTIAL)
299      ereport(...);
303   ri_BuildQueryKeyFull(...);
437 }
```

(a) PostgreSQL-8.1.3

```
399 add_listen_addr(ServerOptions *options,
                   char *addr, u_short port)
400 {
403   if (options->num_ports == 0)
404     options->ports[options->num_ports++]
                                  = SSH_DEFAULT_PORT;
407   if (port == 0)
408    for (i = 0; i < options->num_ports; i++)
409     add_one_listen_addr(options,
                            addr, options->ports[i]);
410   else
411     add_one_listen_addr(options, addr, port);
412 }
```

(b) openssh-4.2p1

Figure 5. Code fragments illustrating the application of mining techniques.

```
void c1() {
 if(packets > 0)
   pack_flag = true;
   size = MAX_SIZE;
   buf = allocbuf(size);
   readbuf(buf, size);
   ...
   ...
   ...
   ...
}
```

(a) c1

```
void c2() {
 if(packets > 0)
   size = MIN_SIZE;
   buf = allocbuf(size);
   if(buf ≠ NULL)
     while(l = lock(buf));
     readbuf(buf, size);
   ...
   ...
   ...
}
```

(b) c2

```
void c3() {
 if(packets > 0)
   i = 0;
   pack_flag = true;
   size = MIN_SIZE;
   buf = allocbuf(size);
   if(buf ≠ NULL)
     while(l = lock(buf));
     readbuf(buf, size);
  ...
}
```

(c) c3

```
void c4() {
 if(packets > 0)
   i = 0;
   size = MAX_SIZE;
   buf = allocbuf(size);
   while(l = lock(buf));
   readbuf(buf, size);
   ...
   ...
   ...
}
```

(d) c4

Figure 6. Illustrative example.

4.1.1 Frequent Itemset Mining

To obtain a specification on predicates where ordering is not critical, we use maximal frequent itemset mining [9]. In this technique, there is assumed to be a set of *transactions*; each transaction contains a collection of elements. The elements that occur in at least n transactions, where n is a confidence threshold specified by the user, is a frequent itemset. For our application, a transaction is a call-site and the set of predicates that hold at the call-site form the elements of the transaction.

We illustrate the mining process using the code fragments shown in Figure 6. We observe that there are four different call-sites to function `readbuf` and each call-site is preceded by a number of operations. For ease of understanding, we use the same names for the associated variables across call-sites. Based on the operations preceding each call to `readbuf`, a number of properties are gleaned and are shown in Table 2. Observe that there are four[1] transactions, equal to the number of call-sites of `readbuf` in the example. For example, observe that there are six items for transaction `c3`. Each item is composed of multiple attributes (e.g., the item associated with variable `l` has two attributes *viz.,* `l` is assigned the return value of `lock(buf)` and is equal to 0 before the call to `readbuf`.). When the frequent items are extracted at confidence 75%, we obtain the following specification:

`packets`:	{>,0}
`size`:	{(**arg**(1), `allocbuf`), (**arg**(2), `readbuf`)}
`l`:	{(=,0), (:=,**res**(`lock`))}
`buf`:	{(**arg**(1), `lock`), (**arg**(1), `readbuf`), (≠,0), (:=,**res**(`allocbuf`))}

[1] Each transaction encodes properties on all possible paths to the call-site.

Depending upon on the level of precision required by the user, the above mining technique can be easily translated into the more restrictive intersection technique, by simply fixing n to be the total number of call-sites (confidence = 100%).

4.1.2 Sequence Mining

For control-flow predicates, frequent itemset mining does not suffice since the order of elements in the transaction is not considered. For deriving precedence relations [31], we use sequence mining [2]. A sequence mining algorithm takes as input a set of sequences (*I*), a 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 s is frequently occurring, all subsequences of s also occur at least as frequently as s. Therefore, we consider only maximal subsequences, i.e., it must be 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 $\{(a \leftarrow b \leftarrow c \leftarrow e), (a \leftarrow d \leftarrow c \leftarrow e), (d \leftarrow a \leftarrow c \leftarrow e), (a \leftarrow c \leftarrow d \leftarrow e \leftarrow f), (e \leftarrow f \leftarrow d \leftarrow c \leftarrow a)\}$, a sequence miner detects $(a \leftarrow c \leftarrow e)$ as a frequently occurring subsequence. Observe that, the same set of transactions without ordering in frequent itemset mining would generate the set {`a, c, d, e`}. For our application, a transaction corresponds to a call site and the sequence within a transaction corresponds to the sequence of procedure calls that

Variables	Transactions			
	c1	c2	c3	c4
packets	{(>, 0)}	{(>, 0)}	{(>, 0)}	{(>, 0)}
pack_flag	{(:=, true)}		{(:=,true)}	
size	{(:=, MAX_SIZE), (**arg**(1), allocbuf), (**arg**(2), readbuf)}	{(:=, MIN_SIZE), (**arg**(1), allocbuf), (**arg**(2), readbuf)}	{(:=, MIN_SIZE), (**arg**(1), allocbuf), (**arg**(2), readbuf)}	{(:=, MAX_SIZE), (**arg**(1), allocbuf), (**arg**(2), readbuf)}
buf	{(:=, **res**(allocbuf)), (**arg**(1), readbuf)}	{(:=,**res**(allocbuf)), (≠, 0), (**arg**(1), lock), (**arg**(1), readbuf)}	{(:=,**res**(allocbuf)), (≠, 0), (**arg**(1), lock), (**arg**(1), readbuf)}	{(:=, **res**(allocbuf)), (≠, 0), (**arg**(1), lock), (**arg**(1), readbuf)}
l		{(=, 0), (:=,**res**(lock))}	{(=, 0), (:=, **res**(lock))}	{(=, 0), (:=, **res**(lock))}
i			{(:= 0)}	{(:= 0)}

Table 2. Transactions associated with calls to **readbuf** shown in Figure 6.

occurred before the call site. Our implementation uses the Apriori-all algorithm by Agrawal and Srikant [2], which is known to scale to over a million sequences. For the example shown in Figure 6, we generate the specification allocbuf ← lock ← readbuf.

4.2 The Structural Similarity Problem

In Figure 6, the names of the variables are the same across multiple call-sites whereas this does not hold in real programs (as noted earlier in this section). In other words, as discussed earlier, predicates computed along different paths may share structural, if not syntactic similarities. In order to capture such similarities, a technique to determine the locations, names, values, etc. that can be abstracted uniformly among different sets is necessary.

Consider every predicate expression as being mapped to a set of locations. Thus, assume a set of location sets, $\{L_1 = \{\ell_{11}, \ell_{12}, ...\ell_{1m_1}\}, L_2 = \{\ell_{21}, \ell_{22}, ...\ell_{2m_2}\}, ... L_k = \{\ell_{k1}, \ell_{k2}, ...\ell_{1m_k}\}\}$, where L_i corresponds to locations associated with predicates that reach call-site i of procedure P. Now, for every element in L_i, we wish to find a corresponding element in every other L_j such that the cumulative matching of the attribute sets for such a mapping is *maximal*. Given three sets A, B and C, we say A and B match maximally, if and only if $\mid A \cap B \mid$ is greater than $\mid A \cap C \mid$ or $\mid B \cap C \mid$.

THEOREM 1. *The maximal matching problem as stated above is NP-hard.*

Proof By reduction from maximal bipartite $(k, *)$-clique in a bipartite graph. [37, 19].

Fortunately, there are a number of heuristics that can be employed to map locations based on semantic information available within programs. We describe below heuristics that match the attribute sets across multiple call-sites that we have used in our implementation.

- **Type:** Attribute sets can be divided based on the type of the variable. e.g., two variables, x and y with attributes [x: {(:=, true)}] and [y: {(>, 20)}] can never be matched.
- **Parameter:** If variables are supplied as arguments to the same parameter for a given procedure at different call-sites, their attributes can be matched. Note, however, that while using positional parameter information for the purposes of matching may be a useful heuristic, other variables that are not used as parameters, but nevertheless are significant as preconditions, need to be detected as well (e.g., matching attribute sets associated with l in Figure 6).
- **Result:** Variables that are assigned the return values of the same function can have their attribute sets matched.

```
let a = λ arg_a. let ptr = ref(...) in p(...)
    in ...

let mkptr = λ z. let ptr = ref(...) in ...
    b = λ arg_b. ... mkptr(...) ... p(...)
    in ...

let mkptr = λz. let ptr = ref(...) in ...
    e = λ arg_e. ... mkptr(...)
    c = λ arg_c. ... e (...) ... p (...)
    in ...
```

Figure 7. Example showing the need for FPA evaluation.

5. Implementation Design

Our implementation takes as input the program source and a user-defined confidence level for determining when a property should form part of a precondition. We first generate the control-flow graph for each procedure using an efficient program analysis tool [6]. The direction of all edges in the control flow graph are reversed, since we need to construct preconditions. The graphs obtained are fed into the intra-procedural analysis framework, which builds the initial set of predicates. This data is processed by the inter-procedural analysis framework iteratively until a fixed-point is reached.

There are two categories of fixed-point iterations that are essential for generating predicates that cross multiple function boundaries. One iteration (FPA – Fixed Point Iteration A) corresponds to the set of tasks accomplished when a procedure is invoked and which are at a lower level of the invocation tree and the other iteration (FPB – Fixed Point Iteration B) corresponds to the set of operations performed before a call to the procedure. We discuss this issue in detail.

Figure 7 presents an example for FPA. From the example, it is clear that calls to procedures a, b, and c always allocate a pointer variable and then call procedure p. Furthermore, procedure mkptr always allocates the pointer variable. To reduce redundant computation of these properties, we maintain a memoization table for each of the procedures in the source, and update the information iteratively until fixed point is reached.

Observe that while the above fixed point iteration accumulates facts in one direction (down the call graph invocation path towards the leaves), there is a necessity for fixed point computation in the reverse direction as well (towards the root in the call graph). Consider the example shown in Figure 8. It is clear that before p is invoked, apart from the pointer being allocated, cond is always true. However, to obtain this information, a fixed point iteration (FPB) in the direction towards the root of the call graph needs to be performed.

```
let x = λ arg_x. if cond then a(...) else ...
    in ...

let y = λ arg_y. if cond then b(...) else c(...)
    in ...

let call_y = λ arg_k. if cond then y(...)
    in ...
```

Figure 8. Example showing the need for FPB evaluation.

procedure BUILDPREDICATES
- ▷ **Input**: `G(V,E)`, directed, acyclic (reversed) CFG of α; `V` is topologically sorted;
- ▷ **Output**: `true` if `dflow` or `cflow` changes from previous iteration for any node in `V`; `false` otherwise;
- ▷ **Auxiliary Information**:
  - `LCS`: longest common subsequence of multiple strings;
  - `data_predicates`(i): data predicates generated at i;
  - `concat`(i,j,k): concatenates strings i, j, k;
  - CALLSITE(i): `true` if i is a callsite;
  - RETURN(i): `true` if i is the exit node from procedure α;

```
1  for each node i = 1 to |V|
2     for all neighbors j of i
3        in_data_flow(i) ← ∩ dflow(j)
4        in_control_flow(i) ← LCS( cflow(j))
5     dflow(i) ← in_data_flow(i) ∪ data_predicates(i)
6     if CALLSITE(i) is true then
7        dflow(i) ← dflow(i) ∪ data_signature[ func(i)]
8        cflow(i) ← concat( cflow(i), func(i),
9                          control_signature[ func(i)])
10    if RETURN(i) is true then
11       data_signature[ α] ← dflow(i)
12       control_signature[ α] ← cflow(i)
```

Figure 9. Algorithm for building predicates.

Figure 9 presents pseudo-code describing details on building the control and data flow predicates, apart from computing procedure summaries (memoization tables) used in FPA. The algorithm follows closely from the analysis formalized in Figure 2. At the end of FPA, a set of predicates (`data_precond` and `control_precond`) for all the procedures in the program are obtained. Figure 10 presents the pseudo-code that performs the mining process that forms part of FPB iteration. There are two mining implementations that we use in our approach – a frequent item-set miner [9] on data flow predicates, where ordering is not necessary, and a sequence miner [2] for control flow predicates. At the end of FPB, the preconditions for the procedures are obtained.

6. Experiments

We validate our ideas on selected benchmark sources, with a view to demonstrating its scalability and effectiveness. We extract preconditions for seven sources: `apache`, `linux`, `openssh`, `osip`, `postgreSQL`, `procmail` and `zebra`. Specific details relating to the sources are provided in Table 3. The size of selected benchmarks varies from 9K to 1.98MLoc. Since default configurations are used to compile these sources, we believe that the number of control flow nodes represents a more reliable indicator of effective source size than lines of code. The number of control flow nodes ranges from 16K to 958K. We also present the number of procedures examined in the table.

We implemented our tool in C++ and perform 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,

procedure CONCATPREDICATES
- ▷ **Input**: α: a procedure in the program;
  `C` = $\{c_1, c_2, \ldots c_n\}$ is the set of call sites of α;
  `E` = $\{e_1, e_2, \ldots e_n\}$ is the set of enclosing procedures for respective call sites;
- ▷ **Output**: `true` if `dflow_precond` or `cflow_precond` changes from previous iteration; `false` otherwise;
- ▷ **Auxiliary Information**:
  `in_control_flow`(c_i): see Figure 9
  `concat`(i,j,k): concatenates strings i, j, k;

```
1  for each node c_i
2     dflow_t(c_i) ← dflow(c_i) ∪ dflow_precond[e_i]
3     cflow_t(c_i) ← concat( cflow_precond[e_i],
                             in_control_flow(c_i), -)
4  Input dflow_t for all c_i into the frequent itemset miner
5  dflow_precond[α] ← result of Step 4
6  Input cflow_t for all c_i into the sequence miner
7  cflow_precond[α] ← result of Step 6
```

Figure 10. Mining preconditions.

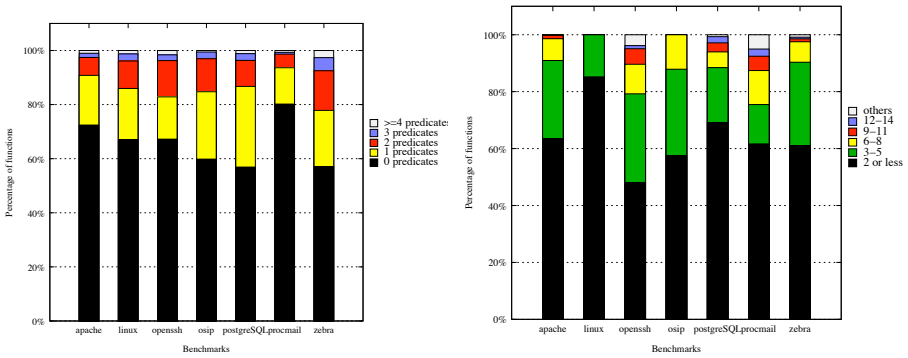
with 1GB memory. The time taken for performing the analysis is presented in Table 3.

6.1 Quantitative Assessment

We derive two kinds of predicates – data-flow and control-flow. For data-flow predicates, we derive assignments to variables and logical relations between variables with other variables and constants. Control-flow predicate specify the procedures that are called before the associate procedure is called. The total number of preconditions generated for procedures mined at 70% confidence is presented in Table 3. Our choice of mining at 70% is somewhat arbitrary, chosen to be resilient to latent errors in the benchmark, without comprising accuracy of the results. The predicate size distribution (the number of predicates found within a precondition) for the generated preconditions is given in Figure 11. For generated data-flow preconditions, the size of the predicate set is less than three for a majority of the procedures. For example, observe that approximately 95% of the procedures in `postgreSQL` have fewer than two predicates in their preconditions. In the case of control-flow predicates, we observe the predicate set size to be less than five for a majority of the procedures. Thus, the output of the tool is tractable for further examination and analysis by users.

To further quantify the effect of the confidence threshold on the preconditions derived, we performed experiments on `apache` over different thresholds. Figure 12 presents the results on the change in the number of preconditions with change in confidence. As expected, we observe that the number of predicates derived reduces with increase in the confidence threshold, although the change is not dramatic. For example, there are 60 additional procedures for which no preconditions are derived when the confidence level changes from 60% to 100%. This is expected because increase in confidence, leads to more aggressive pruning of predicates.

Experiments we conducted that did not include the structural matching heuristics resulted in uninteresting (and fewer) preconditions. This is expected as the attribute sets across call-sites are improperly matched. We also found that parameter matching was by far the most useful heuristic to employ since most real programs employ a coding style that encodes significant semantic information through the flow of parameters and results into and out of functions.



(a) Dataflow predicate distribution (b) Control flow predicate distribution

Figure 11. Predicate distributions.

Source	Version	LoC	CFG nodes	Procedure count	Total number of specifications		Analysis time(s)
					Data-flow	Control-flow	
`apache`	2.2.3	273K	102K	2079	556	330	157
`linux`	2.2.26	1.98M	958K	7465	5862	101	1258
`openssh`	4.2p1	68K	88K	1281	625	202	120
`osip`	3.0.1	24K	34K	666	213	51	46
`postgreSQL`	8.1.3	618K	548K	8568	3348	615	1007
`procmail`	3.22	9K	16K	298	84	105	26
`zebra`	0.95a	183K	145K	3342	1397	608	162

Table 3. Benchmark Information.

6.2 Qualitative Assessment

To study the quality of our results, we examine the effectiveness of our technique in discovering protocols associated with library calls made in `openssh`. We mine the predicates at a 100% confidence threshold. We correlate the effectiveness of the analysis by comparing our results manually with the documentation found in the `man` pages of the corresponding library functions.

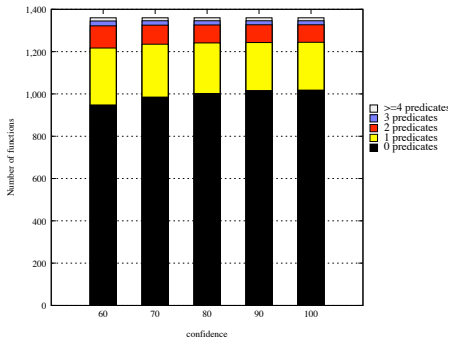


Figure 12. Reduction in number of predicates in `apache` with increase in confidence threshold

Out of the 242 library procedures that are invoked in `openssh`, we derive preconditions correctly for 199 of them (77.13% accuracy). Moreover, we were able to derive preconditions for an additional nine procedures that were not documented. We observe 12 false-positives (our tool derives preconditions where none exist) and 31 *false negatives* (our tool did not derive the preconditions the documented specification claims should hold). These false negatives are potential sources of bugs. False positives occur primarily because there is an insufficient number of use cases for mining to effectively prune away irrelevant predicates. For example, in a few cases, a predicate is included in the precondition to a procedure call because the procedure was invoked only twice and in both cases, the calling context contained similar irrelevant invariants.

Besides being potential sources of bugs, false negatives can also manifest because of limitations in the implementation. In addition to the obvious approximations introduced by our heuristics (recall that a precise solution to structural matching, the heart of the precondition inference problem, is NP-hard), there are two other limitations in our approach addressed briefly below:

- **Absence of theorem proving:** A precondition for the procedure `BN_mod_word` is that the second parameter must not be equal to zero. We do not observe any explicit sanity check on the second parameter in the program source. On further inspection, we notice that there is a chain of assignments leading to the second parameter, where we may be able to prove that the second parameter will be non-zero on any invocation. Automatically formulating this conclusion is possible with the aid of a theorem prover. The integration of theorem proving to the existing infrastructure to handle these predicates is part of our ongoing research.
- **Closed world assumptions:** Sometimes preconditions are formulated with respect to environment variables whose values are directly manifest in the program source. Since our implementation analyzes the program source in a closed-world setting, it is unable to accurately derive preconditions for those procedures whose predicates depend upon values of environment variables.

6.2.1 Bug Detection

We discuss several bugs detected in `openssh-4.2p1`. In the code fragment shown below, neither variables `p` or `q` are checked

for being non-null. Subsequent use of these values in procedure `prime_test` results in a segmentation fault. The computational complexity of the Miller-Rabin primality testing performed in `prime_test` makes it difficult to generate comprehensive test suites that would detect this bug. We exercised this fault by making the system run out of buffer space and using the test case (`ssh-keygen -T <outfile> -f <infile>`), the program crashes in the then latest release `openssh-4.4p1`. Based on our report, these bugs are now fixed in `openssh-4.5p1`. We also observe a similar bug associated with invocation of `BN_new` and subsequent absence of sanity check in the procedure `gen_candidates`.

```
473     p = BN_new();
474     q = BN_new();
475     ctx = BN_CTX_new();
```

Observe that return value of `BN_CTX_new` is also not checked as being non-null. Even though this does not result in a crash, this violation potentially leads to a significant degradation in the performance of the library call `BN_is_prime` used in `prime_test`, as documented in the man pages.

There are several other instances of similar errors in the program. The existing documentation for library procedure `initgroups`, for example, claims that the first parameter to this procedure must always be non-null. However, our analysis does not generate predicates to this effect because this check is not performed. Similarly, before invoking procedure `RSA_size`, the field `n` of its parameter must be non-null. Even though the parameter is checked for being non-null, `n` itself is not. A similar bug exists in the invocation of `DH_size`. Any one of these bugs can be exercised with appropriate inputs, and could lead to a server crash.

7. Related Work

There has been significant research towards automatically validating program properties, and detecting program errors when programs are annotated with partial specifications describing desired invariants [3, 16, 11, 21, 7, 24, 20, 36, 8, 22, 15]. Our approach differs fundamentally from these other efforts insofar we assume *no* input from the programmer on the specifications that need to be validated.

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. Subsequently, Ammons *et al*. present an approach [5] to debug derived specifications using concept analysis. Ernst *et al*. [14] present Daikon, a tool for dynamically detecting invariants in a program. Yang *et al*. [38] present scalable dynamic inference techniques that also work effectively with imperfect traces. While these techniques can indeed be used to derive preconditions, they critically rely on test input providing comprehensive coverage. In this regard, they differ in obvious ways from our approach.

Li and Zhou present PR-Miner [26], a tool that relies on mining [1] to identify frequently occurring program patterns. Our work differs significantly from theirs because we integrate mining within a path-sensitive dataflow framework. Livshits and Zimmermann [28] present a tool which uses mining to analyze revision histories of programs. Li *et al*. [25] also use data mining techniques to detect copy-paste bugs in large software systems. Mandelin *et al*. [29] present a technique for synthesizing jungloid code fragments automatically based on the input and output types that describe the code. Their approach is useful for reusing existing code. Because none of these techniques tightly integrate dataflow and control-flow information with the mining engine, it would be difficult to leverage them for deriving useful preconditions. It is precisely this synthesis that is the distinguishing contribution of this work.

There are several other related approaches that address the problem of mining specifications. An automatic specification mining technique that uses information about exceptions and errors to identify temporal safety rules is presented in [34]. Engler *et al*. [13] use mining to detect relations between pairs of functions, and Kremenek *et al*. [23] significantly generalizes these earlier ideas. However, [23] is domain specific, and requires either machine learning or user specifications to generate initial annotation probabilities, and employs naming conventions for identifying interesting procedures to improve accuracy. As a result, their approach would be ineffective in deriving the specifications for the example programs in Figures 1, 5, or Section 6.2.

To summarize, unlike these other efforts, our approach requires no annotations or guidance from programmers, leverages no presumed semantics of library or primitive functions, is not restricted to limited program contexts (e.g., examining only pairs of functions [13], leveraging program semantics [34], or using domain-specific knowledge [23]), and can detect arbitrarily large and complex preconditions (e.g., Figure 5).

Apart from mining based approaches, many other interesting techniques have been devised for bug detection in software systems [18, 39, 27]. Rinard *et al*. [32] present an approach on failure oblivious computing that enables servers to run even in the presence of memory errors. Castro *et al*. [10] present an approach where a data flow graph is generated and ensures that the data flow integrity is preserved at run time. Because our work focuses on an entirely new dimension, namely statically extracting preconditions from program source transparently, it is conceivable that it could be used in conjunction with these other approaches to operate with even greater precision and scale.

8. Conclusion

This paper focuses on the problem of deriving specifications using predicate mining and describes a static inference mechanism for detecting the preconditions that must be valid whenever a procedure is invoked. We derive these preconditions using an interprocedural path-sensitive dataflow analysis that gathers predicates at each program point. We apply mining techniques to these predicates to make specification inference robust in the presence of errors. We demonstrate the practicality of our techniques by applying it to large open-source C programs. Quantitative and qualitative analysis of the preconditions generated by our system validate its effectiveness.

Acknowledgements

This work is supported in part by the National Science Foundation under grant STI 501-1398-1078. We thank Kathleen Fisher and the anonymous reviewers for their insightful comments. We also thank GrammaTech Inc. for providing `Codesurfer` used to generate the control flow and data flow graphs required for our experiments.

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, 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 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, 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] G. Ammons, D. Mandelin, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 182–195, 2003.

[6] 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.

[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] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.

[9] 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.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th Usenix Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[11] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–95, 2005.

[12] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *KDD '00: Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 9–17, 2000.

[13] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

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

[15] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ESEC-FSE '05: 10th European Software Engineering Conference and 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 227–236, 2005.

[16] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[17] M. Furr and J. Foster. Checking type safety of foreign function calls. In *PLDI '05: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–72, 2005.

[18] P. Godefroid, N. Klarslund, and K. Sen. Dart: Directed automated random testing. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Il, 2005.

[19] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28:140 – 174, 2003.

[20] T. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. *SIGSOFT Softw. Eng. Notes*, 30(5):31–40, 2005.

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

[22] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 38–47, 2005.

[23] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI '06: Proceedings of the 7th Usenix Symposium on Operating Systems Design and Implementation*, pages 161–176, 2006.

[24] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI '05: Proceedings of 6th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 430–447, 2005.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI '04: Proceedings of the 6th Usenix Symposium on Operating Systems Design and Implementation*, pages 289–302, 2004.

[26] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC-FSE '05: 10th European Software Engineering Conference and 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 306–315, 2005.

[27] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, Chicago, Illinois, 2005.

[28] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *ESEC-FSE '05: 10th European Software Engineering Conference and 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 296–305, 2005.

[29] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 48–61, 2005.

[30] J. Palsberg. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst.*, 17(1):47–62, 1995.

[31] M.K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007.

[32] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W.S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI '04: Proceedings of the 6th Usenix Symposium on Operating Systems Design and Implementation*, 2004.

[33] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[34] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS '05: Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, April 2005.

[35] 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.

[36] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 351–363, 2005.

[37] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *KDD '04: Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 344–353, 2004.

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

[39] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI '04: Proceedings of the 6th Usenix Symposium on Operating Systems Design and Implementation*, pages 273–288, San Francisco, CA, 2004.