Static Error Detection using Semantic Inconsistency Inference *

Isil Dillig Thomas Dillig Alex Aiken

Computer Science Department
Stanford University
{isil, tdillig, aiken}@cs.stanford.edu

Abstract

Inconsistency checking is a method for detecting software errors that relies only on examining multiple uses of a value. We propose that inconsistency inference is best understood as a variant of the older and better understood problem of type inference. Using this insight, we describe a precise and formal framework for discovering inconsistency errors. Unlike previous approaches to the problem, our technique for finding inconsistency errors is purely semantic and can deal with complex aliasing and path-sensitive conditions. We have built a null dereference analysis of C programs based on semantic inconsistency inference and have used it to find hundreds of previously unknown null dereference errors in widely used C programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Reliability, Verification, Experimentation

Keywords Static analysis, error detection, satisfiability, inconsistency

1. Introduction

Much recent work in static analysis focuses on *source-sink* properties: For safety policy S, if S is violated when a value constructed at location l_1 is consumed at location l_2, then is there a feasible path from l_1 to l_2? If the answer is "yes," then the program has a bug (violates policy S). Some typical specifications are:

- Does a null value assigned to a pointer or reference reach a pointer dereference?
- Does any closed file reach a file read?
- Does a tainted input reach a security critical operation?

To be concrete, consider the following C-like code:

```
foo(...) {
   if (Q) p = NULL;   (1)
   ...
   bar(p);
}

bar(x) {
   if (R) *x;         (2)
   ...
}
```

The null value assigned at (1) reaches the dereference at (2) if predicates `Q` and `R` can both be true, resulting in a program crash. Several model checkers incorporating predicate abstraction and refinement [2, 3] and type-based systems [10] target such specifications. These systems work by searching for a path from a source to a sink violating the specification.

There is a complementary approach to these problems. Instead of trying to prove that a source can reach a sink, we can look at a set of sinks that a value x reaches and see if they express *inconsistent beliefs* about x [6]. In the example above, assume we did not have the function `foo` available, but that the function `bar` is:

```
bar(x) {
    if (x != NULL) *x;   (2)
    ...
    *x;                  (3)
    ...
}
```

Something is clearly not quite right with this function. At best `bar` is never called with a null value, in which case the test at (2) is just unnecessary and might confuse readers of the code about the actual possible values of `x`. At worst `bar` has a latent crashing bug waiting to happen, as the unprotected dereference at line (3) must cause an error if `x` is null.

Previous work on inconsistency checking is informal in nature, and it is not clear how it relates to standard semantics-based approaches to software analysis [6], but it is clear that relying only on the uses of a value for clues about program errors is something different from what source-sink systems do. The purpose of this paper is to clarify what inconsistency checking is, how it is different from source-sink analysis, and to illustrate by example its potential in practice.

We propose that inconsistency checking is best thought of as a form of an older and better developed idea, type inference. Type inference systems already find type errors based only on the use of values; for example, in any functional language with type inference (e.g., ML or Haskell) the following code

$$\mathtt{x} + \mathtt{cons}(\mathtt{y}, \mathtt{x})$$

will be flagged as having a type error just because the two uses of `x` are type inconsistent (one as a number and the other as a list); note that the type declaration of `x` (the source) is not needed to

* This work was supported by grants from DARPA, NSF (CCF-0430378 and SA4899-10808PG-1), and equipment grants from Dell and Intel.

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

discover this error. From this starting point we make the following contributions:

- The insight that checking consistency of uses is a type inference problem shows a fundamental difference between type inference and source-sink systems, such as most model checkers. Type inference systems find inconsistency errors in open programs, such as libraries (e.g., the second instance of function `bar` above, considered without a caller `foo`) that cannot be found by source-sink analyzers simply because no source exists.
- Casting many inconsistency checking problems as type inference problems requires non-standard types. The core issue is when the values at two usage sites x and y are considered to be "the same", so that x and y are checked for consistency. A particularly problematic case is pointers; we propose that if two pointers point to the same values under the same conditions then those two pointers are really the same pointer (see Section 5.1).
- For path-sensitive analyses there is a difficulty of how to construct appropriate predicates when there is no one source-sink path to use as a source of counterexamples for refinement. We present a method based on computing *correlations* between program predicates and values of interest.
- We conduct an extensive experiment, analyzing over 8 million lines of C source (including the entire Linux kernel) for null dereference errors. We have implemented both source-sink checking and inconsistency checking and found over 600 previously unknown null dereferences, the overwhelming majority of which are found by inconsistency checking. While there are limitations to our experiment (in particular, our implemented analyzer is unsound, which may affect the ratio of source-sink to inconsistency errors detected), based on the results, we believe that inconsistency checking is valuable both because it works for open programs and because the discovered bugs are often local whereas understanding a source-sink path for the same bug appears daunting.

We begin our presentation with a small, paradigmatic language in which we develop our formal results (Section 2). We present both (intraprocedural) source-sink and inconsistency checking for this language (Section 3) and also extend our technique to an interprocedural analysis (Section 4). We then describe a null dereference analysis and necessary extensions for C programs (Section 5) and present our experimental results (Section 6).

2. Language and Inference System

This section describes a simple first order, call-by-value language we use for the formal development.

$$
\begin{array}{lll}
\textit{Program } P & ::= & F^{+} \\
\textit{Function } F & ::= & \texttt{def} f(x_1, \ldots, x_n) = s \\
\textit{Statement } S & ::= & x \leftarrow^{\rho} C_i \mid x \leftarrow^{\rho} y \mid \texttt{check}^{\rho}\ b \mid \\
 & & f(x_1, \ldots, x_n)^{\rho} \mid s_1;^{\rho}\ s_2 \mid \\
 & & \texttt{if}^{\rho}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
\textit{Condition } B & ::= & x = C_i
\end{array}
$$

The language has standard function definitions, assignments, statement sequences, and conditionals; the semantics is also standard and we omit a formal semantics for brevity. The only values in the language are nullary constructors (constants) $C_1, \ldots, C_n$. A condition $x = C_i$ is true if x has the value C_i. A statement $\texttt{check}^{\rho}\ x = C_i$ checks whether variable x is C_i. We use `check` statements to model requirements that a variable must have a certain value at a particular program point. In examples we sometimes need a no-op statement (e.g., to fill in a branch of an `if`); in such cases we write $\texttt{skip}^{\rho}$ to abbreviate the assignment $y \leftarrow^{\rho} y$. We also assume for simplicity that all variables that are not function arguments are assigned to before they are read, so we do not need to define how local variables are initialized.

The superscript ρ's on statements are *labels*. We assume all labels in a program are distinct, uniquely identifying statements. We often abuse our notation slightly by writing s^{ρ} to refer to the top-level label ρ of statement s.

The only sources (constructors) in this language are constants C_i and the only sinks (destructors) are the `check` statements. For example, the following program has a source-sink error: the source assigned at ρ_0 reaches the conflicting sink at ρ_4.

EXAMPLE 1.

```
(x ←^ρ0 C1;^ρ1
if^ρ2 (y = C2)
   then y ←^ρ3 C3
   else check^ρ4 x = C2);^ρ5
if^ρ6 (y = C1)
   then skip^ρ7
   else check^ρ8 x = C1
```

The language syntax allows us to define algorithms via structural induction, but it is also handy to be able to view a function definition as a control-flow graph. For each statement label ρ there are two *program points* ρ^{-} and ρ^{+} representing the points immediately before and after the statement executes, respectively. Definition 1 defines the possible order of evaluation of statements within a function.

DEFINITION 1 (Partial Order on Program Points). For a function $\texttt{def} f(x_1, \ldots, x_n) = s$, let $\prec_f$ be the smallest relation on program points in f satisfying for each sub-statement of s:

$$
\begin{array}{rcl}
x \leftarrow^{\rho} \ldots & \Rightarrow & \rho^{-} \prec_f \rho^{+} \\
\texttt{check}^{\rho}\ \ldots & \Rightarrow & \rho^{-} \prec_f \rho^{+} \\
s_1^{\rho_1};^{\rho_0}\ s_2^{\rho_2} & \Rightarrow & \left\{ \begin{array}{l} \rho_0^{-} \prec_f \rho_1^{-} \\ \rho_1^{+} \prec_f \rho_2^{-} \\ \rho_2^{+} \prec_f \rho_0^{+} \end{array} \right. \\
\texttt{if}^{\rho_0}\ b\ \texttt{then}\ s_1^{\rho_1}\ \texttt{else}\ s_2^{\rho_2} & \Rightarrow & \forall_{i=1,2} \left\{ \begin{array}{l} \rho_0^{-} \prec_f \rho_i^{-} \\ \rho_i^{+} \prec_f \rho_0^{+} \end{array} \right.
\end{array}
$$

Let $\prec_f^{*}$ be the transitive closure of $\prec_f$. A *path* from ρ_0 to ρ_n is a sequence of labels $\rho_0, \ldots, \rho_n$ in f such that

(1) $\rho_i^{-} \prec_f^{*} \rho_{i+1}^{-}$ for $0 \le i \le n-1$
(2) the sequence is maximal between the endpoints: inserting any additional label after ρ_0 and before ρ_n violates (1).

A path is *complete* if it cannot be extended either by adding new labels before the first label or after the last label; a complete path is a path through the entire function body. For instance, in Example 1, there is a path ρ_0, ρ_2, ρ_4 because $\rho_0^{-} \prec \rho_0^{+} \prec \rho_2^{-} \prec \rho_4^{-}$. This path can be extended in both directions to form a complete path $\rho_5, \rho_1, \rho_0, \rho_2, \rho_4, \rho_6, \rho_7$.

2.1 Guards

To allow for path-sensitivity in our static analysis, we construct *guards* that express program constraints. We use boolean satisfiability (SAT) as the underlying decision procedure for solving constraints; hence guards are represented as boolean formulas. In this section, we describe how to compute two kinds of guards:

- *statement guards* that describe the conditions under which a statement executes,
- *constructor guards* that describe the condition under which a variable x at a given program point evaluates to a constructor C_i. In addition, a constructor guard also encodes the source of the value C_i.

$$(1)\quad \frac{}{\Gamma, \gamma \vdash x \leftarrow^{\rho} C_i : \Gamma[x \leftarrow F]} \quad \text{where } F = \lambda(r,j).\text{if } (r,j) = (\rho, i) \text{ then } \gamma \text{ else } \mathit{false}$$

$$(2)\quad \frac{}{\Gamma, \gamma \vdash x \leftarrow^{\rho} y : \Gamma[x \leftarrow \lambda(r,j).\Gamma(y)(r,j) \wedge \gamma]}$$

$$(3)\quad \frac{}{\Gamma, \gamma \vdash \texttt{check}^{\rho}\ x = C_i : \Gamma}$$

$$(4)\quad \frac{}{\Gamma, \gamma \vdash f(x_1, \ldots, x_n)^{\rho} : \Gamma}$$

$$(5)\quad \frac{\Gamma_0, \gamma \vdash s_1 : \Gamma_1 \quad \Gamma_1, \gamma \vdash s_2 : \Gamma_2}{\Gamma_0, \gamma \vdash s_1;^{\rho} s_2 : \Gamma_2}$$

$$(6)\quad \frac{\pi = \bigvee_r \Gamma_0(x)(r,i) \quad \Gamma_0, \gamma \wedge \pi \vdash s_1 : \Gamma_1 \quad \Gamma_0, \gamma \wedge \neg\pi \vdash s_2 : \Gamma_2}{\Gamma_0, \gamma \vdash \texttt{if}^{\rho}\ x = C_i\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2}$$

$$(7)\quad \frac{\lambda x_i.D_{x_i}, \mathit{true} \vdash s : \Gamma}{\vdash \texttt{def} f(x_1, \ldots, x_n) = s}$$

Figure 1. Computing guards.

Constructor guards are functions of type

$$\texttt{CG} = (\texttt{Source} \times \texttt{Int}) \rightarrow \texttt{Guard}$$

The `Int` in the function signature corresponds to a constructor index, and the `Source` in function $\texttt{def f}(\texttt{x}_1, \ldots, \texttt{x}_n)$ is either a label ρ of an assignment statement $z \leftarrow^{\rho} C_i$ in f or one of the function arguments $\texttt{x}_1, \ldots, \texttt{x}_n$. Sources used in constructor guards track the origin of every value in a function in terms of function arguments or constructor assignments within that function. We use $r, r', r_1, \ldots$ to range over sources.

Consider an assignment $x \leftarrow^{\rho} C_i$ with statement guard γ. The constructor guard g_x for x after the assignment is $g_x(\rho, i) = \gamma$, where γ is the statement guard for ρ, and $g_x(r, j) = \mathit{false}$ for all $r \neq \rho$ and $j \neq i$. Thus, the constructor guard encodes that immediately after the assignment the value of x is C_i from source ρ if γ is satisfied, and no other value/source combinations are possible.

We require that the formulas in the range of a constructor guard be pairwise disjoint: if g is a constructor guard and $g(r, i) = \gamma_1$ and $g(r', j) = \gamma_2$, then $\gamma_1 \wedge \gamma_2 = \mathit{false}$ if $r \neq r'$ or $i \neq j$. This condition captures the idea that a value cannot simultaneously be two distinct constructors or come from two different sources. We can always enforce this condition by adding new unconstrained boolean variables to guards. For example, if there are only two constructors C_1 and C_2, then the constructor guard g with $g(r, 1) = \alpha$ and $g(r, 2) = \neg\alpha$ enforces disjointness; for more constructors we can use additional fresh variables. We write D_x for a fresh constructor guard associated with function argument x. By fresh, we mean that the formulas in the range of D_x share no variables with D_y for distinct variables x and y. Furthermore, $D_x(r, j) = \mathit{false}$ for all $r \neq x$; i.e., the only source of values in D_x is x.

Figure 1 gives inference rules for computing both statement guards and constructor guards resulting from executing a statement. An *environment* Γ : $\texttt{Var} \rightarrow \texttt{CG}$ is a map from program variables to constructor guards. For a statement s and initial environment Γ and statement guard γ, the system proves sentences of the form $\Gamma, \gamma \vdash s : \Gamma'$, where Γ' is the final environment after execution of s. Note that the inference system is purely structural; in any proof there is exactly one conclusion associated with statement s, which we can rewrite as:

$$\Gamma^{\rho^-}, \gamma^{\rho} \vdash s^{\rho} : \Gamma^{\rho^+}$$

In this way we can refer to the environments for the program points before and after ρ as well as the statement guard under which s^{ρ} is executed.

We briefly explain the rules in Figure 1. When a variable x is assigned a constructor C_i (rule (1)), x's constructor guard shows that it cannot have any value other than C_i from source ρ (guards for all other constructors and all other sources are *false*). Furthermore, x only has value C_i if the assignment executes (the guard γ on the assignment statement holds). The second form of assignment (rule (2)) says that the possible sources/values of x after the assignment are the possible sources/values of y before the assignment, but only if the assignment executes—the statement guard γ is added to the guard of every possible source/value pair.

A $\texttt{check}^{\rho}\ x = C_i$ statement (rule (3)) tests the predicate $(x = C_i)$ at run-time. These are the sinks in our language. The purpose of our analyses is to characterize when the run-time test can evaluate to *false*; this can model, for example, the implicit assertion that a pointer is non-null before it is dereferenced, or more generally that a value of a discriminated union type has the correct constructor (our choice of the term "constructor" is meant to suggest discriminated unions), or that a value is in the correct type-state before some operation is performed. As our interest is in when the test is *false* and not what happens as a result of the test, we define `check` statements to have no effect on the environment.

Function calls (rule (4)) also have no effect on the environment; because there are no visible side-effects of a function and no return value, function calls have no effect on the callee's state. Of course, this rule also gives us no information about `check` statements that may fail in the called function; thus, Figure 1 defines an intraprocedural analysis. We discuss extensions to interprocedural analysis in Section 4.

Rules (5) and (6) deal with compound statements. The rule for statement sequences (rule (5)) is standard. For an `if` statement (rule (6)) with statement guard γ, the guard π combines all the conditions under which $x = C_i$ from any source. The true branch is analyzed with statement guard $\gamma \wedge \pi$ and the false branch is analyzed with statement guard $\gamma \wedge \neg\pi$. The final result is a join $\Gamma_1 \sqcup \Gamma_2$ of the final environments of the two branches, defined as

$$(\Gamma_1 \sqcup \Gamma_2)(x)(r,i) = \Gamma_1(x)(r,i) \vee \Gamma_2(x)(r,i)$$

Finally, a function body (rule (7)) is analyzed in an environment where nothing is known about a function argument except that it evaluates to a single constructor at a given call site (recall that for each argument x, the guards in the range of the constructor guard for x are all disjoint).

Notice that statement guards and constructor guards are mutually dependent (e.g., rules (1) and (6)) and thus are computed simultaneously. The reason for this design decision is that the computation of statement guards is affected by side-effects of statements, which are in turn implicitly captured by constructor guards. Conversely, the condition under which a statement causes a particular side-effect to happen depends on the condition under which that statement executes; hence the computation of constructor guards makes use of statement guards. As an illustration of guard computation, consider the example:

EXAMPLE 2.

```
if^ρ0 (x = C1)
  then (
     x ←^ρ1 C2;^ρ6
     if^ρ2 (x = C1)
        then check^ρ3 x = C2
        else skip^ρ4 )
  else skip^ρ5
```

Suppose that we are interested in knowing the statement guard associated with the `check` statement at program point ρ_3. Suppose ρ_0 is the entry point of a function, and let α_1 and α_2 be formulas that represent the conditions under which function argument `x` evaluates to $\mathtt{C}_1$ and $\mathtt{C}_2$ at function entry respectively. After executing the assignment statement at program point ρ_1, the guard under which `x` evaluates to $\mathtt{C}_1$ is *false* by Rule (1) of Figure 1, and the guard under which `x` evaluates to $\mathtt{C}_2$ is α_1, which is the statement guard at this program point. The statement guard at program point ρ_3 is computed using Rule (6), where $\gamma = \alpha_1$ and $\pi = \mathit{false}$; hence the statement guard at ρ_3 is $\alpha_1 \wedge \mathit{false} = \mathit{false}$. Since the statement guard at ρ_3 is *false*, the path from the function entry ρ_0 to program point ρ_3 is not feasible.

As this example illustrates, the computation of statement guards directly allows the discovery of infeasible paths in a program.

DEFINITION 2 (Feasibility). Let $\rho_0, \ldots, \rho_n$ be a path. Then the path is *feasible* if $\mathit{SAT}(\bigwedge_{0 \le i \le n} \gamma^{\rho_i})$.

Returning to Example 1, the path of the source-sink error ρ_0, ρ_2, ρ_4 is feasible for an appropriate initial environment, but the path $\rho_0, \rho_2, \rho_3, \rho_6, \rho_7$ is not feasible in any environment. The following lemma captures some simple but very useful facts about feasible paths.

LEMMA 1. Assume $\Gamma, \gamma \vdash s : \Gamma'$ and let σ be any assignment that satisfies γ. Then there is a unique complete, feasible path including s such that σ satisfies all the statement guards on the path.

PROOF. The proof is by induction on the structure of s. The interesting case is when $s = (\mathtt{if}^{\rho}\ x = C_i\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2)$. From rule (6) of Figure 1, the final step of the derivation must be:

$$\frac{\begin{array}{c}\pi = \bigvee_r \Gamma(x)(r,i) \\ \Gamma, \gamma \wedge \pi \vdash s_1 : \Gamma_1 \\ \Gamma, \gamma \wedge \neg\pi \vdash s_2 : \Gamma_2\end{array}}{\Gamma, \gamma \vdash \mathtt{if}^{\rho}\ x = C_i\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2}$$

Now either $\sigma(\gamma \wedge \pi)$ is true or $\sigma(\gamma \wedge \neg\pi)$ is true. Assume that $\sigma(\gamma \wedge \pi)$ is true. Then

$$\Gamma, \gamma \wedge \pi \vdash s_1 : \Gamma_1$$

satisfies the induction hypothesis with assignment σ, and so there is a unique complete feasible path $\rho_1, \ldots, \rho_n$ for s_1 such that $\sigma(\gamma^{\rho_i})$ is true for all $1 \le i \le n$. Then $\rho, \rho_1, \ldots, \rho_n$ is the desired path for s. The case where $\sigma(\gamma \wedge \neg\pi)$ is true is symmetric. □

3. Error Detection

In this section we present techniques for identifying source-sink and inconsistency errors using the machinery developed in Section 2. Only intraprocedural techniques are discussed here; Section 4 extends the approach across function boundaries.

3.1 Source-Sink Errors

Source-sink errors arise when a value constructed at one program point reaches an unexpected destructor at a different program point. Most errors uncovered by model checking tools, and particularly model checkers based on counter-example driven refinement, are source-sink errors. This class of errors includes, for example, type-state properties, such as errors that arise from dereferencing a pointer that has been assigned to null, or using a tainted input in a security critical operation.

DEFINITION 3 (Source-Sink Error). Consider the sub-derivation for a `check` statement:

$$\Gamma^{\rho^-}, \gamma^{\rho} \vdash \mathtt{check}^{\rho}\ x = C_i : \Gamma^{\rho^+}$$

The check can fail because of a value from source ρ' if the statement is reachable when constructor C_j originating from ρ' is in the constructor guard of x for some $j \neq i$. More formally, a source-sink error arises if there is a label ρ' of an assignment statement $y \leftarrow^{\rho'} C_j$ such that

$$\mathit{SAT}(\gamma^{\rho} \wedge \bigvee_{j \neq i} \Gamma^{\rho^-}(x)(\rho', j))$$

The following lemma shows that there is always at least one feasible path corresponding to any source-sink error.

LEMMA 2. Every source-sink error is included on at least one complete feasible path.

PROOF. Let $\psi^{\rho}_{\rho'} = \gamma^{\rho} \wedge \bigvee_{j \neq i} \Gamma^{\rho^-}(x)(\rho', j)$, and let σ be any assignment satisfying $\psi^{\rho}_{\rho'}$. Since the formula $\psi^{\rho}_{\rho'}$ is satisfiable there is at least one such σ. By Lemma 1, σ defines a unique, complete feasible path. By expanding the definition of $\psi^{\rho}_{\rho'}$ and using the fact that rule (1) in Figure 1 includes the statement guard in the constructor guard after the assignment, we can show that $\psi^{\rho}_{\rho'}$ satisfies both statement guards $\gamma^{\rho'}$ and γ^{ρ}. Thus, both the assignment statement and the `check` are on the path. □

Consider once more the program in Example 1. Assignment statement ρ_0 gives x a constructor guard where C_1 from source ρ_0 has guard *true* (just because x is assigned C_1 at ρ_0). The constructor guard of x is not affected by the `check` statement at ρ_4. Since the check is whether $x = C_2$, one of the tests for a source-sink error is:

$$\mathit{SAT}(\gamma^{\rho_4} \wedge \bigvee_{j \neq 2} \Gamma^{\rho_4^-}(x)(\rho_0, j))$$

Because γ^{ρ_4} is satisfiable and $\Gamma^{\rho_4^-}(x)(\rho_0, 1)$ is *true*, we have shown a source-sink error in the program.

Note that Definition 3 requires that the source be the label of an assignment statement—we do not consider function arguments as sources in computing source-sink errors, because we do not know what actual values a function argument may have while analyzing only the function body. Source-sink errors may arise if a function is called with certain arguments that cause the `check` statement to fail. Interprocedural analysis is required in this case to find the matching source, if any, that actually causes the sink to fail; we address interprocedural source-sink errors in Section 4.

3.2 Inconsistency Errors

In this section, we define inconsistencies and describe a technique for semantically detecting inconsistency errors.

Consider the following motivating example:

EXAMPLE 3.

```
def f(a) =
  (x ←^ρ0 a;^ρ1
  if^ρ2 (x = C1)
     then check^ρ3 x = C1
     else y ←^ρ4 x);^ρ5
  check^ρ6 a = C1
```

In this example, `a` and `x` are aliases for the same value because of the assignment at ρ_0. At ρ_3, `x` is asserted to have the value C_1 and this statement is protected by the conditional at ρ_2. The variable `a` is also asserted to be C_1 at ρ_6, but without the protecting test. Thus, if there actually is an environment in which this function can be called where $\mathtt{a} \neq \mathtt{C}_1$, an error is sure to occur at ρ_6. The presence of the test at ρ_2 protecting the check at ρ_3 is evidence that some programmer believes there are such environments. Thus, without knowing anything about the rest of the program, it is

likely that there is something wrong in this function because of the inconsistent assumptions about a and x.

This example illustrates that inconsistency errors can involve aliasing if multiple names for the same value are used inconsistently. Finding inconsistency errors means identifying a set of uses of the same value that should be compared. If we are to take aliasing into account, we cannot rely on uses of the same variable name or (more generally) syntactically identical program expressions to identify the set of uses—a semantic test for "sameness" is needed.

More formally, we define a congruence relation $v_1 \cong v_2$ that captures when two quantities v_1 and v_2 should be checked for consistency. The exact definition of $\cong$ varies with the programming language. For our toy language, an appropriate definition is that two variables at given program points are congruent if they have the same values under the same guards at those points.

DEFINITION 4 (Congruence). Let v_1 and v_2 be two variables in the same function f, and let ρ_1^- and ρ_2^- be program points in f. Then $v_1^{\rho_1^-} \cong v_2^{\rho_2^-}$, meaning variable v_1 at program point ρ_1 is congruent to variable v_2 at program point ρ_2, if

$$\forall i.\ \bigvee_r \Gamma^{\rho_1^-}(v_1)(r,i) \equiv \bigvee_r \Gamma^{\rho_2^-}(v_2)(r,i)$$

Notice that we do not require that the sources of congruent variables be the same. Thus x and y can be congruent even if they are constructed completely independently; we return to this point shortly.

DEFINITION 5 (Inconsistency Error). Consider two check statements $\texttt{check}^{\rho_0}\ x = C_i$ and $\texttt{check}^{\rho_1}\ y = C_i$. There is an *inconsistency error* between the two statements if the variables are congruent and one check can fail while the other cannot. Formally, there is an inconsistency if the following three conditions are satisfied:

$$(1)\quad x^{\rho_0^-} \cong y^{\rho_1^-}$$
$$(2)\quad \neg SAT(\gamma^{\rho_0} \wedge \bigvee_r \bigvee_{j \neq i} \Gamma^{\rho_0^-}(x)(r,j))$$
$$(3)\quad SAT(\gamma^{\rho_1} \wedge \bigvee_r \bigvee_{j \neq i} (\Gamma^{\rho_1^-}(x)(r,j))$$

Condition (2) says that it is not the case that the statement guard at ρ_0 can hold and x has some value other than C_i. Condition (3) says that there is at least one solution where the statement guard at ρ_1 holds and y has some value other than C_i.

Returning to Example 3 above, at point ρ_6^- the variable a has constructor guard D_a (the original guards for a, as there are no assignments to a in the function) and at point ρ_3^- the variable x has the same guards because of the assignment at ρ_0. Thus $\texttt{a}^{\rho_6^-} \cong \texttt{x}^{\rho_3^-}$, satisfying condition (1). Now the statement guard at ρ_3 includes a conjunct $\Gamma^{\rho_3^-}(\texttt{x})(\texttt{a},1)$, which is disjoint with any guard $\Gamma^{\rho_3^-}(\texttt{x})(r,j)$ for $j \neq 1$ (recall Section 2.1). Hence, the check statement at ρ_3 cannot fail, and condition (2) is satisfied. Finally, the statement guard at ρ_6^- is just *true*, and so condition (3) is also satisfied.

As noted above, our definition of congruence does not require any dataflow relationship between the two variables—variables with different sources may still be congruent. Thus, unlike in Example 3, two congruent variables may not even have a common source. At first look, this definition of congruence seems too permissive in that it allows variables that apparently coincidentally share the same values to be compared. We argue that even when two variables don't share a common source, an inconsistency still exists. Consider the following example:

EXAMPLE 4.

```
def f(a) =
   if(a=C1)
      then x ← C1
      else x ← C2;
   if(a=C1)
      then y ← C1
      else y ← C2;
   check(x=C2);
   if(y=C2)
      then check(y=C2)
      else skip;
```

In this example, x and y have the same values under the same conditions, but not from the same sources. The conditional if(y = C2) indicates that some programmer believes there is some call site where a can be C1; otherwise, y would always be C1. But if this is the case, then x can also be C1, and there is at least one execution trace where check(x=C2) will fail. Hence, the above example should be classified as an inconsistency, justifying our definition of congruence.

Finally, note that while source-sink errors are characterized by a single feasible path, inconsistency errors are characterized by a feasible path (condition (3)) and the absence of any feasible path to a different program point (condition (2)). Thus, inconsistency inherently requires reasoning about the relationships among multiple paths, unlike source-sink error detection.

3.3 Intersection of Source-Sink and Inconsistency Errors

Our discussion so far highlights that source-sink and inconsistency error detection techniques are fundamentally different: First, detection of source-sink errors involves reasoning about a single program path, while the detection of inconsistencies can require reasoning about multiple paths. Second, source-sink error detection requires the source to be explicit in the source code, while inconsistency detection infers errors only from usage sites, i.e., sinks, and can therefore find errors even when the source comes from the environment.

Despite these differences, some errors can be seen both as source-sink and inconsistency errors; the following example is prototypical:

EXAMPLE 5.

```
if^ρ0(x = C1)
   then check^ρ1 x = C2
   else skip^ρ2
```

This example has an obvious error since the conditional $\texttt{if}(\texttt{x} = \texttt{C}_1)$ ensures that the check statement at program point ρ_1 fails. Despite the fact that there is no explicit source (i.e., a constructor assignment), the above example can be considered a source-sink error. Since x is known to be $\texttt{C}_1$ inside the true branch of the if statement, adding an extra assignment of the form $\texttt{x} \leftarrow^{\rho} \texttt{C}_1$ in the true branch preserves program semantics and introduces a feasible path between the source $\texttt{x} \leftarrow^{\rho} \texttt{C}_1$ and the sink $\texttt{check}^{\rho_1}\ \texttt{x} = \texttt{C}_2$.

On the other hand, we can also see this error as an inconsistency. Using the intuition that inconsistency detection is a generalization of type inference, we can introduce types POSSIBLY_C1 and NOT_C1. Informally, the example does not type-check because the test $\texttt{if}^{\rho_1}(\texttt{x} = \texttt{C}_2)$ adds a type constraint that x has type POSSIBLY_C1, while the unprotected check statement assigns the NOT_C1 type to x. More precisely, we can identify this error using Definition 5. Assuming that the language has only the constructors $\texttt{C}_1$ and $\texttt{C}_2$, adding the check statements $\texttt{check}^{\rho'}\ \texttt{x} = \texttt{C}_1$ and $\texttt{check}^{\rho''}\ \texttt{x} = \texttt{C}_2$ in the true and false branches of the if statement respectively preserves the semantics of the above program, yield-

ing the semantically equivalent code:

```
if^ρ0(x = C1)
   then (
        check^ρ'  x = C1;^ρ'''
        check^ρ1  x = C2 )
   else check^ρ''  x = C2
```

This directly exposes the inconsistency in the program according to Definition 5, because the check statement at ρ_1 can fail while the one at ρ'' cannot.

4. Interprocedural Error Detection

In this section we discuss interprocedural extensions to our approach for detecting both source-sink and inconsistency errors. Before presenting our interprocedural analysis we first revisit what we mean by inconsistency errors; unlike source-sink errors, the definition of inconsistency must be reconsidered in the interprocedural case. Consider the following example:

EXAMPLE 6.

```
def f(x) =
   if^ρ0(x = C1)
      then check^ρ1  x = C1
      else skip^ρ2  ;^ρ3
   g(x)^ρ4

def g(y) =
   check^ρ5  y = C1
```

This program clearly has an inconsistency error: The `check` at ρ_1 is protected by a test at ρ_0, but the `check` in `g` on the same value is unprotected. Now consider the following, slightly different, example:

EXAMPLE 7.

```
def f(x) =
   g(x)^ρ0  ;^ρ1
   check^ρ2  x = C1

def g(y) =
   if^ρ0(y = C1)
      then check^ρ1  y = C1
      else skip^ρ2
```

This example simply interchanges the protected and unprotected `check` statements: The `check` in the caller is now unprotected while the callee guards the `check`. Extending our intraprocedural definition of inconsistency errors in the obvious way leads us to conclude that this example also has an inconsistency error, but this definition of inconsistency results in large numbers of false positives on real programs. The issue is that `g` may have other callers besides `f`. That is, while `f` may be safe in relying on $\mathtt{x} = \mathtt{C}_1$, other callers of `g` may pass arguments other than C_1. Defensive programming of this sort is very common in practice. A typical example is a library that does extensive checking of arguments, while client code may be written with the knowledge that certain values cannot arise.[1]

In summary, Example 6 should be considered an inconsistency error, while Example 7 should not. Thus, when comparing two uses of a value between a caller and a callee, we only consider pairs of uses where the callee `check` can fail. This decision implies that we do not need to track `check` statements that are guaranteed to succeed outside of their containing function; the only interprocedural information we need is knowledge of when a `check` statement in a function can fail.

We use function summaries for interprocedural analysis: a summary is computed of the conditions under which a function f can fail, and this summary is then used at each call site of f to model f's behavior for the purpose of detecting source-sink and inconsistency errors. This approach is context-sensitive, since the summaries are applied separately at every call site. We describe how function summaries are defined and used in a basic form (Sections 4.1 and 4.3) and introduce a significant improvement (Section 4.2).

4.1 Function Summaries

A function summary describes the preconditions on the execution of a function that, if satisfied, may lead to errors. Computing sound and very precise preconditions is easy in our framework; the disjunction of all the failure conditions for every `check` statement in a function characterizes exactly the condition under which some `check` will fail. Unfortunately, propagating such precise information interprocedurally is prohibitively expensive; the formulas grow very rapidly as conditions are propagated through a series of function calls.

We take a different approach to function summaries that is designed to scale while still expressing all the possible conditions under which a `check` in a function may fail. The price we pay is a loss of precision in the general case; one can construct examples for which our summaries greatly overestimate the precondition for failure. However, our summaries do precisely summarize the failure precondition of the vast majority of functions we have observed in practice.

A *function summary* S has the same signature as a constructor guard: a map from sources (in this case just function arguments) and constructor indices to guards:

$$\mathtt{S} = (\mathtt{Source} \times \mathtt{Int}) \rightarrow \mathtt{Guard}$$

The interpretation of summaries is different, however. The idea is that if $S(a, k) = \pi$, then a call of f where formal parameter a is C_k can fail if the initial state of the call also satisfies predicate π. For example, in Example 6, $S_g(y, 1) = \mathit{false}$ and $S_g(y, i) = \mathit{true}$ for $i \neq 1$ captures that when the argument is C_i for any $i \neq 1$ function g may fail. In Example 7, $S_g(y, i) = \mathit{false}$ for all i expresses that the function can never fail.

DEFINITION 6 (Function Summary). Consider a function f where

$$\frac{\lambda x_i.D_{x_i}, \mathit{true} \vdash s^{\rho_0} : \Gamma}{\vdash \mathtt{def} f(x_1, \ldots, x_n) = s^{\rho_0}}$$

Then

$$(S_f(x_i, j) = \pi) \Rightarrow \Pi(x_i, j, \pi) \text{ where}$$

$\Pi(x_i, j, \pi) \equiv$
(1) $\forall(\mathtt{check}^{\rho_1}\ x = C_k)$ in f where $k \neq j$.
(2) if $\mathit{SAT}(\gamma^{\rho_1} \wedge \Gamma^{\rho_1^-}(x)(x_i, j))$ then
(3) $(\gamma^{\rho_1} \wedge \Gamma^{\rho_1^-}(x)(x_i, j)) \Rightarrow \pi$

In words, for each function argument x_i and constructor C_j, on line (1) we consider the set S of all statements $\mathtt{check}^{\rho_1}\ x = C_k$ such that the `check` fails if $x = C_j$ (i.e., the condition $k \neq j$). On line (2) we further restrict our focus to the subset S' of statements in S where the `check` can fail because the source of constructor C_j is argument x_i. On line (3), for every `check` in this smaller set S', we are looking for a necessary condition π that holds whenever one of

[1] A similar problem arises with our definition of inconsistency in the presence of function macros. Since macros are used in many different contexts, they are often written with defensive checks. In our implementation, code resulting from a macro expansion is tagged in the parse tree as coming from a macro and treated as an inlined function.

the checks in S' fails. As a result, π gives an over-approximation of the condition under which a `check` statement in `f` will fail if argument x_i is constructor C_j at some call site. In other words, if $SAT(\pi \wedge (x_i = C_j))$ for some call site, a `check` may fail in `f`.

It is easy to see that setting π to *true* always satisfies the conditions, so that $S_f(x_i, j) = \mathit{true}$ for all x_i and C_j is always a correct, if very imprecise, function summary. If $S_f(x_i, j) = \mathit{false}$ then no `check` in f can fail when $x_i = C_j$.

One simple strategy for computing function summaries is:

$$S_f(x_i, j) = \begin{cases} \mathit{false} & \text{if } \Pi(x_i, j, \mathit{false}) \\ \mathit{true} & \text{otherwise} \end{cases}$$

The reader may easily confirm that this algorithm yields $S_g(y, 2) = \mathit{true}$ and $S_g(y, 1) = \mathit{false}$ for Example 6. In Section 4.2 we consider how to compute guards π other than *true* and *false*. Now consider a more involved example:

EXAMPLE 8.

```
def foo(a1, a2) =
  if^ρ0 (a2 = C2)
    then x ←^ρ1 a2
    else x ←^ρ2 a1 ;^ρ3
  check^ρ4 x = C2
```

Assume that the only constructors are C_1 and C_2. Applying the test given in Definition 6 to $S_{\texttt{foo}}(a_1, 1)$, we have:

- The single `check` statement satisfies line (1) of Definition 6 with $k = 2$.
- For line (2), γ^{ρ_4} is *true* and $\Gamma^{\rho_4^-}(x)(a_1, 1)$ is satisfiable because of the assignment at ρ_2.
- For line (3), setting $\pi = \mathit{true}$ satisfies the implication.

4.2 Correlation Analysis

The summary generation strategy described in Section 4.1 has two principal strengths. First, it captures the common case where an error in the body of a function is triggered by the value of a single function argument. Second, if the only possibilities for π are *true* and *false*, then the size of summaries is guaranteed to be bounded by the product of the number of function arguments and the number of distinct constructors.

However, there are many realistic examples where this approach is not expressive enough, because there are times when programmers use two or more correlated arguments to a function; consider, for example, when one argument serves as a flag describing the state of another argument. The following example encodes such an idiom in our toy language:

EXAMPLE 9.

```
def f(a1, a2) =
  if^ρ0 (a2 = C1)
    then check^ρ1 a1 = C1
    else skip^ρ2
```

If the predicates of $\mathtt{S_f}$ are limited to *true* and *false*, then the best we can do in this example is $\mathtt{S_f}(\mathtt{a_1}, 2) = \mathit{true}$, which is rather coarse as `f`'s check does not unconditionally fail when $\mathtt{a_1} \neq \mathtt{C_1}$. A better summary would record that $\mathtt{S_f}(\mathtt{a_1}, 2) \equiv (\mathtt{a_2} = \mathtt{C_1})$, precisely capturing the necessary condition for failure when $\mathtt{a_1} = \mathtt{C_2}$. We perform a *correlation analysis* to discover such additional predicates:

DEFINITION 7 (Correlation Analysis). Consider a function definition $\mathtt{def\ f}(\mathtt{x_1}, \ldots, \mathtt{x_n}) = \mathtt{s}$. Let $\phi_{k,h}$ be a formula for the expression $(\mathtt{x_k} = \mathtt{C_h})$.

$$S_f(x_i, j) = \bigwedge \{\phi_{k,h} | \Pi(x_i, j, \phi_{k,h})\}$$

In Example 9, we have $\gamma^{\rho_1} \equiv (\mathtt{a_2} = \mathtt{C_1})$, and so $\mathtt{S_f}(\mathtt{a_1}, 2) \equiv (\mathtt{a_2} = \mathtt{C_1})$ using the algorithm in Definition 7. Similarly, using the correlation analysis for computing a more precise summary for Example 8, we obtain $\mathtt{S_{foo}}(\mathtt{a_1}, 1) \equiv (\mathtt{a_2} = \mathtt{C_1})$.

It is instructive to compare our approach to interprocedural path sensitivity with source-sink error detectors. While full interprocedural path sensitivity may be intractable for large programs, model checking techniques have shown that computing path sensitivity in a demand-driven fashion can avoid tracking unnecessary predicates and allow analyses to scale [3, 2, 5]. However, such model checkers rely on having a full path from source to sink to drive the process of discovering the needed predicates, information we do not have available both in an inconsistency analysis and a compositional interprocedural source-sink analysis. Correlation analysis allows us to find relevant predicates that play a role in interprocedural communication by computing necessary conditions for errors to occur. The price we pay is that we restrict the space of predicates considered to ensure scalability; for example, in our toy language we only consider the predicates $\phi_{k,h}$.

4.3 Summary Application

Consider a function definition

$$\mathtt{def\ f}(\mathtt{a_1}, \ldots, \mathtt{a_n}) = \mathtt{s}$$

and call site $\mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n})$ and a summary $\mathtt{S_f}$. We use the summary of `f` to model `f`'s behavior at the call site as follows. We define a new function $\mathtt{f_{summary}}(\mathtt{a_1}, \ldots, \mathtt{a_n}) = \mathtt{s'}$ where $\mathtt{s'} = \ldots; \mathtt{s_{ij}}; \ldots$ is a sequence of statements, one for every argument $\mathtt{a_i}$ and constructor $\mathtt{C_j}$. From Definition 7, $\mathtt{S_f}(\mathtt{a_i}, \mathtt{j})$ must have the form

$$\mathtt{S_f}(\mathtt{a_i}, \mathtt{j}) = \phi_{\mathtt{k_1},\mathtt{l_1}} \wedge \ldots \wedge \phi_{\mathtt{k_m},\mathtt{l_m}}$$

Abusing our syntax slightly, we define $\mathtt{s_{ij}}$ to be:

```
if^ρij (ai = Cj)
  then check((ak1 ≠ cl1) ∨ ... ∨ (akm ≠ clm))
  else skip;
```

At the call site we simply replace the statement $\mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n})$ by $\mathtt{s'}[\mathtt{x_1}/\mathtt{a_1}, \ldots, \mathtt{x_n}/\mathtt{a_n}]$. This approach, which inlines a "stub" function that approximates the error behavior of the original function, allows us to reuse the intraprocedural algorithms for detecting source-sink and inconsistency errors from Section 3 unchanged.

5. A Null Dereference Analysis

In this section, we apply our approach to the problem of detecting null dereference errors in C programs. We first present an encoding of the null dereference problem in our framework and then discuss extensions needed to analyze C.

To apply the techniques in Sections 2-4 to the problem of detecting unsafe null dereferences, we need only define the constructors and an appropriate congruence relation. Null dereference analysis is about understanding what pointers can be null, which in turn requires a reasonably precise model of all the possible values of all pointers in a program. Our C implementation incorporates a sound context-, flow- and partially path-sensitive points-to analysis for C [11]. Most points-to analyses compute a graph where the nodes V are the set of abstract locations and there is an edge $(v, v') \in E$ if location v may point to v'. The points-to analysis we use labels each points-to edge with a guard $(v, v')^g$, where g is a formula specifying under what conditions v points to v'. The value `NULL` is treated as a node in the graph, so $(v, \texttt{NULL})^g$ means that v may be a `NULL` pointer whenever guard g is satisfied.

For the congruence relation, given a guarded points-to graph (V, E), we say that $v_1, v_2 \in V$ are congruent, $v_1 \cong v_2$, if

$$\forall v_3 \in V.(((v_1, v_3)^{g_1} \in E \Leftrightarrow (v_2, v_3)^{g_2} \in E\ ) \wedge\ g_1 \equiv g_2)$$

That is, two pointers are equivalent if they are aliases of one another: they point to the same locations under the same conditions.

To model constructors, we classify all pointers as NULL or NOT-NULL (i.e., everything except NULL). Before each pointer dereference $*\mathtt{x}$ we insert a check:

$$\mathtt{check}^{\rho}\ \mathtt{x} = \mathtt{NOT\text{-}NULL}$$

The check succeeds only if the NULL guard in x's points-to graph is unsatisfiable at point ρ^{-}.

To illustrate how we detect null inconsistency errors in C, consider the following example:

EXAMPLE 10.

```
void foo(int* p, int* q, bool flag)
{
   P1.  flag = (p!= NULL);
   P2.  q = p;
   P3.  if (flag)
   P4.     *p = 8;
   P5.  *q = 4;
}
```

The assignment at P2 ensures p and q have the same guarded points-to relationships; thus $\mathtt{p} \cong \mathtt{q}$. The dereference of p at P4 cannot fail because the statement guard (the test on flag at P3) guarantees that p is non-null. However, the dereference of q at P5 can fail because the statement guard is just *true*. Thus, we detect a null inconsistency in foo.

5.1 Extensions for C

There are features in C that are not in the toy language we have used to present our techniques. We briefly discuss the most significant extensions that are required to support analysis of C programs.

The biggest technical difference between the toy language and C is that C functions can have externally visible side-effects. In particular, for a null dereference analysis, it is necessary to estimate the set of function side-effects making locations either null or not null. We address this problem by using a separate side-effect analysis to compute sources of null (both in the return value and as a result of function side-effects) as well as to track modifications to function arguments. However, this side-effect analysis is best effort and unsound; it tracks side-effects that must result in a location being assigned null, but it does not capture all assignments that just might result in a location being assigned null. In our opinion, this is the major source of unsoundness in our implementation.

The difficulty in estimating function side-effect information lies in resolving the tension between two competing goals. First, the quantity of side-effect information is potentially enormous; computing even simple use/mod information for every function (i.e., just the set of abstract locations the function reads or writes) in a large program is intractable if the result is represented naively, because the set of side-effects of a function includes all the side-effects of functions it can call either directly or indirectly. Thus, it is necessary to aggressively summarize interprocedural side-effect information to avoid consuming space quadratic (or worse) in the size of the program. Second, the resulting information must be precise enough to yield useful results, because even small imprecisions can lead to overwhelming numbers of false positives. We are not aware of any general results on efficiently computing interprocedural side-effect information; the problem appears to be unsolved. Previous null dereference analyzers have focused on intraprocedural checking (see Section 7).

Another separate issue is what predicates are used by the correlation analysis to compute function summaries. In Definition 7, we considered only predicates $\phi_{k,h}$ corresponding to conditions of the form $(\mathtt{x_k} = \mathtt{C_h})$. Unfortunately, in a real programming language, there are arbitrarily many predicates of this form. For example, if a function argument x is an integer, it is obvious that we cannot test $\mathtt{x} = \mathtt{c}$ for every possible integer constant. Our approach is to consider only the predicates that occur inside if statements in the computation of π.

An orthogonal issue is the modeling of loops and recursive functions. The system defined in Sections 2-4 can be used to analyze recursive functions in a sound manner by a standard iterative fixed point computation. In our implementation for C we analyze each function only once and do not attempt to compute fixed points, in part to limit the growth in interprocedural side-effect information.[2] We have observed that the function summary guards inferred by correlation analysis are almost always very simple; in fact, conjunctions of more than two simple atomic predicates are exceedingly rare, if in fact they ever occur (we have yet to notice one with more than two clauses). Thus, we believe that very simple restrictions on the size and form of function summary guards (along with conservative approximation if those limits are exceeded) would be sufficient to ensure that a fixed point computation terminates with useful (i.e., sufficiently precise) results.

Finally, as discussed above, our system builds upon a may-alias analysis for C. This underlying analysis is sound assuming the C program is memory safe (a standard assumption in may-alias analysis), a condition that is not checked by the alias analysis or our system.

6. Results

We have run our null dereference analysis on seven widely used open source projects and identified 616 null dereference issues with 149 false positive reports (an overall 19.5% false positive rate). These projects receive regular source code checking from multiple commercial bug-finding tools, and so we sought to learn whether these bugs had been previously reported. Developers for the Samba project confirmed that none of the Samba bugs had been previously found. For the other projects we did not receive such an explicit acknowledgment that the bugs were new; however, we judge from the fact that fixes were released quickly for many of the bugs shortly after our reports were filed that at least the majority of the bugs we found were previously unknown. The large majority of these bugs, 518, were found by our inconsistency analysis.

We ran our null dereference analysis on a compute cluster. Analyzing the Linux kernel with over 6 MLOC required about 4 hours using 30 CPU's, which was by far the longest time required for any of the projects. The smallest project we analyzed was OpenSSH, which took 2 minutes and 33 seconds to analyze on the same cluster. Our system makes many calls to a boolean SAT solver to test the satisfiability of the various predicates used in our analyses, and for Linux the number of SAT queries numbers in the millions. We impose a 60 second time limit for analyzing any individual function; if the analysis of a function times out, its function summary is incomplete.

Figure 2 summarizes our experimental results. The first column gives the number of lines of code for each project, the second column presents the total number of reports, which is classified in the following three columns into correct reports, false positives, and undecided reports (reports that we could not classify as either correct reports or as false positives, because the interpretation of these reports required a more global understanding of the code base than we had). The sixth column gives the false positive rate, which is calculated without including the undecided reports. The second group of three columns breaks down the correct reports by kind: the count of inconsistency errors excluding those also found by source-sink detection, the number of source-sink errors found

[2] Cycles of mutually recursive functions are analyzed once in an arbitrary order.

	LOC	Total	Correct	Undecided	False Pos	% False Pos	Inconsistent	Source-Sink	Both	% Interproc	% Alias
OpenSSL 0.9.8b	339319	55	47	2	6	11.3%	40	6	1	38.3%	34.0%
Samba 3.0.23b	515689	68	46	3	19	29.2%	40	4	2	34.8%	17.4%
OpenSSH 4.3p2	154660	9	8	0	1	11.1%	6	2	0	37.4%	0.0%
Pine 4.64	372458	150	119	3	28	19.0%	105	10	4	42.0%	6.7%
MPlayer 1.0pre8	761708	119	89	2	28	23.9%	71	16	2	41.6%	30.3%
Sendmail 8.13.8	364569	9	8	0	1	11.1%	7	1	0	62.5%	12.5%
Linux 2.6.17.1	6275017	373	299	8	66	18.1%	249	38	12	27.8%	12.0%
Total	8783420	783	616	18	149	19.5%	518	77	21	34.1%	15.4 %

Figure 2. Experimental Results

also as inconsistencies, and the number of errors identified by both. The last group of two columns show the percentages of correct reports that were interprocedural and that involved pointer aliasing, respectively. Many current bug finders ignore pointer aliasing and interprocedural analysis; at least for null dereference analysis, our results show that both features are important.

We used the following methodology in classifying the error reports. First, source-sink errors resulting from dereferences of return values of functions which can potentially return null were counted once per function, not once per call site. Return values of `malloc` wrappers that can return null are often used unsafely at many call sites, resulting in a misleadingly large number of correct reports if each such call site is counted as a bug. Second, we classified inconsistency reports as correct reports if there was actually an inconsistency, not if we could prove that the inconsistency would lead to a run-time crash. Lacking a detailed global understanding of these large projects, we could often not differentiate between redundant null checks and potential crashing bugs. In our correspondence with project developers, we were told that some of the inconsistency errors are due to redundant null checks. However, a large majority of developers deemed every inconsistency, including those believed to be redundant null checks, worth fixing. The majority view was that inconsistency errors represented misunderstandings of the inconsistent function's interface and should be fixed. A large number of error reports we classified as correct were confirmed by the developers; however not all project developers gave us feedback about the validity of error reports. In such cases, the numbers in Figure 2 represent our best effort to classify these errors.

Figure 2 shows that the large majority (87.5%) of the errors are inconsistency errors (including conditional misuse errors). Since most of these inconsistency errors were immediately fixed by developers, it is our belief that semantic inconsistency detection is able to identify real errors and important interface violations in real code. Figure 2 also reveals that roughly a third of the overall correct reports involve interprocedural dependencies, sometimes involving many function calls, especially in the case of source-sink errors. Our initial experiments with the tool also highlight the importance of selective path-sensitivity: A first version of the analysis without path-sensitivity resulted in a high false positive rate, while experiments with full path-sensitivity had unacceptably high time-out rates. However, using the correlation analysis, the time-out rate in our experiments stayed between 0.71% and 6.4% of all functions with an acceptable false positive rate.

Another interesting observation from Figure 2 is that a non-negligible number of errors (roughly one-third in OpenSSL and MPlayer) involve pointer aliasing. Pointer aliasing contributes to a significant source of null pointer errors, especially inconsistency errors, in two common programming patterns. The first pattern we observed is that generic `void*` pointers are often aliased by typed pointers and aliases with different types are used with inconsistent null pointer assumptions. The other pattern is that array elements are often assigned to "convenience" pointers, which denote current, head, or tail elements of a data structure. Programmers sometimes make different null pointer assumptions when they alternate, for example, between using `array[0]` and `head`.

The main source of false positives is imprecision in the pointer analysis we used, which collapses aggregate structures (e.g., arrays, lists) to a single abstract location. If a null pointer is assigned to any element of an aggregate data structure, it contaminates other elements of the same data structure, causing the analysis to raise false alarms whenever an element of such a contaminated data structure is dereferenced. Other contributing factors to false positives are some unmodeled constructs, such as inline assembly.

We conclude this section by presenting two sample errors reported by the analysis, which we believe to be representative of many of the error reports generated by the tool:

```
/* Linux, net/sctp/output.c, line 270 */

236 pmtu = ((packet->transport->asoc) ?
237 (packet->transport->asoc->pathmtu) :
238 (packet->transport->pathmtu));
...
269 if (sctp_chunk_is_data(chunk)) {
270   retval = sctp_packet_append_data(packet, chunk);
...
286 }

538 sctp_xmit_t sctp_packet_append_data
      (struct sctp_packet *packet,...)
540 {
...
543 struct sctp_transport *transport = packet->transport;
...
545 struct sctp_association *asoc = transport->asoc;
...
562 rwnd = asoc->peer.rwnd;
```

This example illustrates an interprocedural inconsistency error involving pointer aliasing, which might potentially cause a null dereference at line 562. On line 236, the pointer `packet->transport->asoc` is compared against null and `packet` is later passed to a function which first aliases `packet->transport` as `transport` and then aliases `transport->asoc` as `asoc`, which is finally dereferenced at line 562. Despite these aliasing relationships, the caller function assumes that `packet->transport->asoc` may be null, while the called function dereferences the same

pointer without ensuring it is non-null, causing the analysis to generate an inconsistency warning.

The next error illustrates an inconsistency error involving two mutually exclusive paths:

```
/* OpenSSL, e_chil.c line 1040 */
static int hwcrhk_rsa_mod_exp(BIGNUM *r, const BIGNUM *I,
                              RSA *rsa, BN_CTX *ctx)
967 {
985  if ((hptr = RSA_get_ex_data(rsa, hndidx_rsa))!= NULL)
987  {
990    if(!rsa->n){
994      goto err;
995    }
997    /* Prepare the params */
998    bn_expand2(r, rsa->n->top); /* Check for error !! */
       ...
1027 }
1028 else
1029 {
       ...
1039   /* Prepare the params */
1040   bn_expand2(r, rsa->n->top); /* Check for error !! */
       ...
1080 }
```

In the true branch of the `if` statement, the pointer `rsa->n` is checked for being null and subsequently dereferenced at line 998. On the other hand, the same pointer is dereferenced without a null check in the false branch of the same `if` statement at line 1040. The important point about this example is that detecting inconsistencies requires reasoning about multiple paths simultaneously.

7. Related Work

The various program analysis traditions appear to have equivalent power; for example, there is an equivalence between type systems and model checking [15]. However, these results are for closed programs. We observe that for open programs techniques that search only for a single source-sink path cannot express inconsistency errors requiring simultaneous reasoning about multiple distinct paths. We view semantic inconsistency checking as complementary to source-sink error detection; inconsistency checking can find bugs where there are multiple sinks but no sources, while source-sink checking can detect bugs between a single source and a single sink.

Our choice of the terms *constructor* and *destructor* is inspired by work on detecting uncaught exceptions in functional programs [18, 17] and soft typing [4, 1]. A core issue in both bodies of work is tracking which datatype constructors a program value may actually have at run-time. Null dereference analysis is a special case where there are only two constructors `NULL` and `NON-NULL`; our techniques could be adapted to give very precise analysis for these other applications as well.

FindBugs [12] is a widely used tool for Java that has paid particular attention to finding null dereference errors [13]. FindBugs pattern-matches on constructs that are common sources of certain error classes and performs some data-flow computation. As our implementation is for C, it is not possible to do a direct comparison. Nevertheless, it is clear that FindBugs would not find the many path-sensitive, interprocedural, and alias-dependent bugs our more semantic analyses uncover. One can also interpret our results as indicating that, at least for tools requiring no user annotations, one must move to computationally intensive models (incorporating at least path sensitivity) to do significantly better than tools like FindBugs without unusably high false positive rates.

Some approaches attack null dereferences using user annotations on function parameters and local checking of each function body. LCLint [7] uses an unsound procedure to check the safety of dereferences of parameters annotated as may-be-null. More recent annotation-based systems are much closer to being sound [9, 8]. Current annotation languages, which mark a single parameter as possibly null or definitely not null, are not expressive enough to capture the more complex path-sensitive and interprocedural relationships we observed in our experiments.

Another approach, exemplified by CCured [16], is to use a relatively inexpensive static analysis to verify the safety of many pointer dereferences statically and then to introduce dynamic checks to enforce the remaining dereferences at run-time. The unification-based type inference used in CCured would not find most of the bugs our tool detected, and while the program would at least fail in a well-defined way if the null dereference was triggered at run-time, it would still fail.

Engler et al. were the first to explicitly propose a method for finding null dereference errors based on inconsistency checking [6]. They argue that inconsistencies suggest programmer confusion and the presence of bugs, and they give some techniques for discovering inconsistencies. We observe that their notion of inconsistency is essentially the same as the idea underlying type inference systems, where inconsistent type constraints from multiple uses of a value result in a type error. Our inconsistency analysis adopts this more semantic point of view and we give purely semantic conditions for inconsistency checking, which allows our system to uncover subtler bugs involving, e.g., pointer aliasing.

Our approach to selective inter-procedural path-sensitivity is reminiscent of some selectively path-sensitive model-checking techniques. ESP, for example, only accurately tracks branches that affect relevant properties within that branch [5]. Unlike ESP, our approach is fully path-sensitive intraprocedurally, and more importantly, our analysis infers correlated predicates by computing implication relations between predicates and guards of relevant events. Model checking tools based on predicate abstraction and refinement [2, 3, 14] also achieve selective path-sensitivity by discovering relevant predicates. Such tools start with a coarse abstraction which is refined by tracking additional relevant predicates until a path is shown to be feasible or infeasible or until no new useful predicates can be discovered. As discussed in Section 4, our approach differs because inconsistency analysis does not have a source-sink path to use as a source of counterexamples.

8. Conclusion

We have proposed semantic inconsistency inference for finding errors and interface violations in large software systems. We have presented the results of experiments on a number of open source applications, showing that semantic inconsistency checking can uncover a large number of previously undiscovered errors.

References

[1] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[2] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the Symp. on Principles of Prog. Languages*, pages 1–3, January 2002.

[3] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In *Proc. of the Conf. on Fundamental Approaches to Software Engineering*, pages 2–18, 2005.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pages 278–292, 1991.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pages 57–68, 2002.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *Operating Systems Review*, 35(5):57–72, 2001.

[7] D. Evans. Static detection of dynamic memory errors. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pages 44–53, 1996.

[8] M. Faehndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. of the Conf. on Object-Oriented Programing, Systems, Languages and Applications*, pages 302–312, 2003.

[9] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pages 234–245, 2002.

[10] J. Foster, M. Faehndrich, and A. Aiken. A theory of type qualifiers. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pages 192–203, 1999.

[11] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the ACM International Symposium on Foundations of Software Engineering*, pages 69–80, 2006.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[13] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proc. of the Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, 2005.

[14] R. Jhala and K. McMillan. Interpolant-based transition relation approximation. In *Proc. of the International Conf. on Computer Aided Verification*, pages 39–51, 2005.

[15] M. Naik and J. Palsberg. A type system equivalent to a model checker. In *Proc. of the European Symp. on Prog.*, pages 374–388, 2005.

[16] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of the Symp. on Principles of Prog. Languages*, pages 128–139, 2002.

[17] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proc. of the Symp. on Principles of Prog. Languages*, pages 276–290, 1999.

[18] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proc. of the International Symp. on Static Analysis*, pages 98–113, 1997.