

A FLOW-, PATH-, AND CONTEXT-SENSITIVE NULL  
DEREFERENCE ANALYSIS FOR C PROGRAMS

A THESIS  
SUBMITTED TO THE COMPUTER SCIENCE DEPARTMENT  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
COMPUTER SCIENCE WITH HONORS

Isil Ozgener (Dillig)

January 2007

© Copyright by Isil Ozgener (Dillig) 2007  
All Rights Reserved

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as an undergraduate honors thesis.

---

Alex Aiken Principal Adviser

Approved for the University Committee on Undergraduate Studies.

# Acknowledgments

I would like to thank my adviser Alex Aiken and all the members of the SATURN project group for their valuable contributions to this this work. In addition, I would like to thank Mary McDevitt from the Technical Communications Program for proof-reading this thesis.

# Contents

<b>Acknowledgments</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 The SATURN Framework</b>	<b>5</b>
2.1 Description of the Language . . . . .	5
2.2 Expressing path-sensitivity . . . . .	6
2.3 Modeling memory . . . . .	6
2.4 Transforming Expressions to Traces . . . . .	8
2.5 Summary-Based Approach . . . . .	10
<b>3 The Intra-Procedural Null Analysis</b>	<b>12</b>
3.1 Null Guard and Guarded Memory Locations . . . . .	12
3.2 Detecting Errors Caused by the Dereference of a Pointer Set to Null .	14
3.3 Detecting Errors Caused by the Misuse of Conditionals . . . . .	16
3.4 Detecting Inconsistencies . . . . .	17
3.4.1 Guarded Location Sets . . . . .	18
3.4.2 Defining an Inconsistency . . . . .	20
<b>4 The Inter-procedural Null Analysis</b>	<b>24</b>
4.1 Interface Objects . . . . .	24
4.2 Summary Representation . . . . .	25
4.3 Dereference Summary . . . . .	26

4.4	Return Summary . . . . .	29
4.5	Alias Summary . . . . .	30
4.6	Side Effect Summary . . . . .	31
<b>5</b>	<b>Results</b>	<b>39</b>
<b>6</b>	<b>Future work</b>	<b>42</b>
	<b>Bibliography</b>	<b>43</b>

# Chapter 1

## Introduction

*Null dereference errors* are common bugs in real software that result in unacceptable failures and program crashes. Most `null` dereference errors are triggered only under complicated execution paths that may be hard to trigger even with carefully designed test cases. Using dynamic instrumentation to discover many kinds of memory errors, such as `null` dereference errors, fails to detect many serious bugs in real, complex software because of the inherent difficulty of covering all possible execution paths by running a given program with a finite number of input parameters. In contrast to dynamic methods, static analysis techniques symbolically simulate source code to discover interesting properties about program behavior under every possible execution path, rather than just the finite set of paths triggered by test input. The `null` checker that is described in this thesis uses a framework for static analysis to simulate program execution in a flow-, path-, and context-sensitive way to detect `null` dereference errors in real code.

Unfortunately, `null` dereference errors are not easily modeled by some of the common abstractions heavily used in many static analyzers, such as finite state machine properties. For example, while a lock that is acquired by a given thread needs to be eventually released on all execution paths, a `null` pointer does not need to be re-assigned to a non-`null` value on all execution paths in the program, making it difficult

to identify transition rules governing the state of a pointer variable. Furthermore, almost all interesting `null` dereference errors arise from the interaction between different functions (often written by different programmers), and simple state-machine strategies fail to scale when analyzing thousands of functions in a real code base. Even using simplifying strategies such as program slicing that retain only relevant parts of a program do not help scalability very much since pointer manipulations are extremely frequent, and the set of relevant program properties tends to blow up exponentially as information is propagated across hundreds of function boundaries.

In order to deal with the issue of scalability in large and complex software, the research community has focused on two different approaches. One of these approaches has been to rely heavily on programmer annotations to explicitly declaring invariants on pointer types across function boundaries [4]. Typical annotations required by this approach are what aliasing relations may be introduced by a called function, what parameters of a function or fields of a global data structure may or may not be `null`, and how the return value of a function relates to its parameters. This information can then be used check for contradictions between the function's simulated behavior and the programmer-annotated invariants. For example, annotating a parameter as `null` indicates that pointer may be `null`, and dereferencing such a pointer inside a function body without checking that it is not `null` violates an invariant, which is likely to cause an error under some possible execution path. While annotations allow static checkers to be more scalable and track more accurate information, they are not easily adopted by many real-world programmers who tend to view annotations as cumbersome. Furthermore, even though it is possible to verify the absence of `null` pointer errors modulo the programmer-annotated invariants, it is also possible that some of the programmer annotations are wrong. For example, if a programmer adds an annotation stating that a certain argument of a function is never `null` when this is not actually the case, the analysis would incorrectly conclude that the program is free of `null` dereference errors.

Another approach in the research community for dealing with the complexity introduced by function boundaries is to largely ignore them, focusing on intra-procedural information and looking for inconsistencies that are likely to indicate programmer confusion or carelessness. One approach collects a set of programmer beliefs, such as the belief that a pointer may be `null`, then identifies contradictions among the beliefs in this set [3]. For example, in the following code fragment, comparing a pointer against `null` in line 5 indicates the programmer’s belief that the pointer variable `tty` may be `null`:

```
/* 2.4.7: drivers/char/mxser.c */
1. int mxser_write (struct tty_struct *tty, ...)
2. {
3.     struct mxser_struct *info = tty->driver_data;
4.     unsigned long flags;
5.     if(!tty || !info->xmit_buf)
6.         return(0);
7. }
```

Although the expression `if(!tty)` in line 5 expresses the programmer’s belief that the pointer variable `tty` may be `null`, the dereference of the variable `tty` at line 2 without a comparison against `null` reflects the programmer’s conflicting belief that `tty` can never be `null`. By identifying such contradictory assumptions about pointer variables, the checker can conclude that either the check is redundant or that there is a potential error in the code that needs to be fixed. This general technique of identifying inconsistencies between programmer’s assumptions within one function is very effective in detecting many real errors in large code bases and scales to millions of lines of code, such as the Linux Kernel and OpenBSD.

Our approach to detecting `null` dereference errors does not require annotations or any other form of programmer assistance. Our analysis is inter-procedural, finds many serious bugs in supposedly-well-tested code bases, and scales to millions of lines

of C code. Unlike many previous works, our analysis also takes aliasing relationships between pointers into account and is able to detect `null` dereference errors even when detecting complicated aliasing relations are involved. Our analysis also identifies inconsistent assumptions between programmer beliefs, but unlike previous approaches, it does not rely on syntactic matching on expressions. The `null` dereference analysis is built in the SATURN framework, which uses a SAT-based approach to analyze programs [5].

# Chapter 2

## The SATURN Framework

### 2.1 Description of the Language

SATURN is a static analysis framework that accurately models the C programming language. SATURN translates C programs into a representation that makes it easier for checkers built on top of the framework to track properties of interest [5]. In the SATURN representation, a program consists of function bodies, global variables and types, where a function body consists of commands and a set of program variables, which include local and global variables, as well as function parameters [1]. Statements  $s$  can be stores, calls, and dynamic memory allocations; a function body is formed by composing commands through sequences and non-deterministic branches.

$$s := s_0; s_1 \mid if \ ? \ s_0 \ s_1 \mid e_0 \Leftarrow e_1 \mid call \ f \mid e \Leftarrow new(\tau) \quad (2.1)$$

Expressions  $e$  are constants, program variables, and casts, and binary or unary operator expressions.

$$e := const \mid var \mid cast(e) \mid unop \ e \mid e_1 \ binop \ e_2 \quad (2.2)$$

Each expression is labeled with a unique integer identifying where the expression occurs; this integer is called a *program point*  $\rho$ . Note that syntactically equivalent expressions, such as `*ptr` occurring in different contexts are distinguished by their  $\rho$

values.

We also define conditionals  $c$  in SATURN as follows:

$$c := true \mid false \mid neg\ c \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid e_1\ comp\ e_2 \quad (2.3)$$

where  $comp \in \{=, <, \leq, >, \geq, \neq\}$ .

SATURN converts every conditional to an equivalent boolean formula, which is referred to as a *guard*  $g$  in the framework. Guards represent path-sensitive conditions under which a statement executes. Consider the following simple example:

```

stmt1;          /* g1 = true */
if (b<0) stmt2; /* g2 = b < 0 */
else stmt3;     /* g3 = b > 0 */
stmt4;         /* g4 = true */

```

## 2.2 Expressing path-sensitivity

To translate between conditionals and their corresponding boolean formulas, we define the lift operation.  $lift(c)$  is a boolean formula representing the condition  $c$ . SATURN represents every variable and scalar as a vector of 32 boolean variables, one for each bit on a 32-bit architecture. Representing conditions as precise boolean formulas is very important in the SATURN framework, since the path-sensitive information encoded by guards at every program point can be utilized by any analysis to make satisfiability queries about program properties of interest under any possible execution path.

## 2.3 Modeling memory

Memory locations are described by globally unique *traces*. A trace describes the access path to global or local variables, function parameters, newly allocated data sites, return values of other functions, and constants. We build traces according to the

recursive definition given below:

```

trace_root := arg{arg_num}
            | glob{name}
            | local{name}
            | new_loc{id}
            | return{fn_name}
            | constant{bitvector}

```

```

trace := trace_root | drf{trace} | field{trace, field_name}

```

A memory location associated with a function parameter has a root trace `arg{num}` where `num` is an integer, identifying which formal parameter of the function this parameter is. The memory location associated with global and variables are labeled with root traces `glob{name}` and `local{name}` respectively, where `name` is the name of the variable in the program. The root trace `new_loc{id}` represents freshly allocated memory with a unique `id` representing a particular memory allocation site in the program. Calls to allocation functions such as `malloc` or `calloc` yield memory cells with root trace `new_loc`. The root trace `return{fn_name}` identifies the return value from a function called `fn_name`. Scalar values are represented with the root trace `constant`. Recall that SATURN represents scalars as 32 boolean variables (called *bitvectors*); thus, a bitvector bound to a constant trace identifies the value of this constant. A `null` pointer would also have a root trace `const` with the associated zero bitvector.

An important invariant maintained by the framework is that every distinct memory cell has a unique trace associated with it. Representing memory cells as traces allows the framework to conveniently describe memory locations by how they can be accessed through program variables of interest. As will be clear shortly, traces are invaluable for modeling heap variables and play a key role in the `null` dereference analysis.

Finally, we conclude this section with a few simple examples of traces. Consider the `struct` declaration:

```
struct state
{
    int num;
    char* buf;
    struct state* next;
}
```

and the function definition

```
int foo(struct state* s)
{
    (1) int a = s->next->num;
    (2) s->buf = malloc(a*sizeof(char))
    (3) return a;
}
```

At the exit point of `foo`, the trace of the integer variable `a` is `fld{drf{fld{drf{arg0}, next}}, num}`, and the trace of variable `s->buf` is `newloc{1}` because it is assigned to a freshly allocated memory location in line (2).

## 2.4 Transforming Expressions to Traces

Static analyzers checking memory safety properties must track memory locations that a pointer expression can point to. Any alias analysis, `null` dereference analysis, or buffer-overflow analysis must know of the set of memory locations that can be accessed through a variable in order to perform any useful work. Pointer aliasing information can be represented as a guarded points-to graph, where the target of every pointer variable yields a certain memory location under a guard, stating the path-sensitive conditions under which this points-to relationship holds [1].

Since we track memory locations in terms of traces that are described in 2.3, it is necessary that we are able to evaluate the target trace of every expression at any given program point. We introduce a *points-to* function:

$$points\_to(\varepsilon, \rho, g) = \tau$$

which retrieves the set of traces  $\tau$  that expression  $\varepsilon$  can point to at program point  $\rho$  under the statement guard  $g$ .

To clarify the semantics of the *points-to* function, consider the following example:

```
void foo(int* ptr1, int* ptr2, bool flag)
{
(P1)  int* a, * b;
(P2)  a = ptr1;
(P3)  if(flag) ptr2=ptr1;
(P4)  b = ptr2;
}
```

The result of the *points-to* function on variables `ptr1` and `ptr2` at selected program points are:

```
Ex1: points_to(ptr1, P2, true) = {drf{arg1}}
Ex2: points_to(ptr2, P4, flag) = {drf{arg1}}
Ex3: points_to(ptr2, P4, ¬flag) = {drf{arg2}}
```

Example 1 is straightforward; examples 2 and 3 are more interesting. Because of the conditional assignment of `ptr1` to `ptr2` at P3, the memory location that is referenced through `ptr2` at P2 is its original target (i.e.  $drf\{arg_2\}$ ) under the condition that the guard `flag` does not hold. Otherwise, the target trace of expression `ptr2` is the entry target trace of `ptr1`, which is  $drf\{arg_1\}$ .

Let us also note that the cardinality of the set obtained from the *points\_to* operation need not always be one. For example, at the exit point P4 with statement guard **true**, we have:

$$\text{points\_to}(\text{ptr2}, P4, \text{true}) = \{drf\{arg_1\}, drf\{arg_2\}\}.$$

Since it is also important to be able to retrieve the condition under which an expression points to a particular memory location at a given program point, we define the *points\_to\_guard* as:

$$\text{points\_to\_guard}(\varepsilon, \rho, l) = g \tag{2.4}$$

such that expression  $\varepsilon$  at program point  $\rho$  will point to memory location  $l$  under the resulting guard  $g$ . For example, in the example above:

Ex1:  $\text{points\_to\_guard}(\text{ptr1}, P2, drf\{arg_1\}) = \text{true}$

Ex2:  $\text{points\_to\_guard}(\text{ptr2}, P4, drf\{arg_1\}) = \text{flag}$

Ex3:  $\text{points\_to\_guard}(\text{ptr2}, P4, drf\{arg_2\}) = \neg \text{flag}$

An important point to keep in mind about the *points\_to\_guard* operation is the following: For any memory location  $l$  such that  $l \notin \text{points\_to}(\varepsilon, \rho, \text{true})$ ,  $\text{points\_to\_guard}(\varepsilon, \rho, l) = \text{false}$ . The application of the *points\_to\_guard* function to a pointer that has never been explicitly set to **null** will always yield *false*. For instance, in the previous example,  $\text{points\_to\_guard}(\text{ptr1}, \rho_1, \text{null}) = \text{false}$  since **null** is not included in the may-point-to set for **ptr1** in **foo**.

## 2.5 Summary-Based Approach

SATURN adopts a summary-based approach to support inter-procedural analysis. Checkers built in the framework abstract relevant program properties of interest (for example, the aliasing relationships introduced by a function) and incorporate this

behavior into a function summary associated with the function. At call sites, the summary of the callee is queried, and the summary representing the behavior of the called function is used to account for the observable side effects of the callee. Every checker defines its own abstractions to incorporate different properties of interest.

The summary-based approach to inter-procedural analysis has several advantages. Unlike function inlining, a summary abstracts only relevant properties and does not typically lead to an exponential blow-up. Once a summary for a function has been computed, we can reuse this summary for all calls to this function. In addition, context-sensitivity can be achieved by using a summary-based approach for inter-procedural analysis.

Since analyzing a function  $f$  involves querying summaries of the functions  $f$  calls, we must analyze callees before callers. If there are cyclic dependencies among functions, we re-analyze mutually recursive functions until the summaries stabilize. As a final note, SATURN does not make any distinction between loop and function bodies: Loops are treated as tail-recursive functions in the framework, and summaries are generated for loops as well as for functions.

# Chapter 3

## The Intra-Procedural Null Analysis

In this chapter, we describe the rules that identify whether a pointer known to be `null` is dereferenced under some possible execution path. We also discuss how these rules apply to concrete examples.

### 3.1 Null Guard and Guarded Memory Locations

In this section, we first define what a *null guard* is. Consider a pointer dereference expression,  $\varepsilon$ , that occurs at program point  $\rho$  under statement guard  $\psi$  such that  $\text{points\_to\_guard}(\varepsilon, \rho, \text{drf}\{\text{null}\}) = \sigma$ . In other words, the pointer expression  $\varepsilon$  evaluates to trace `null` under the *null guard*  $\sigma$ . To disambiguate the *null guard* from the statement guard, we denote the *null guard*  $\sigma$  and the statement guard as  $\psi$ . It is important to note that in the general case the *null guard* and the expression guard need not in any way be related to each other. For example, consider:

```
void foo(int* p, bool flag)
{
(1)   if(flag) p = 0;
(2)   *p = 5;
}
```

The `null` guard for the dereference expression at line (2) is the boolean formula representing the condition `flag = true`; however the expression guard for the same expression is `true` since the expression `*p` executes unconditionally.

Note that if the `null` guard for an expression  $\varepsilon$  is `false`, this does not mean that the  $\varepsilon$  can never evaluate to `null`. Since our analysis is bottom-up rather than top-down, `null` pointers that are passed by callers are not known to be `null` when the called function is first analyzed. Thus, the *points\_to\_guard* function will return a value that is satisfiable only when a pointer has been explicitly set to `null` within that function or inside one of its callees.

Another very important concept for the `null` dereference analysis is the notion of *guarded location*. A memory location is said to be *guarded* iff the statement guard implies that this expression can never evaluate to `null`. We define the *guarded* predicate as:

$$\textit{guarded}(\varepsilon, \psi) = \begin{cases} \textit{true} & \textit{if } \neg\textit{SAT}(\psi \wedge (\neg(\textit{lift}(\varepsilon))))^1 \\ \textit{false} & \textit{otherwise} \end{cases} \quad (3.1)$$

Consider the code fragment:

```
void bar(int* p, int* q)
{
(1) p = q;
(2) if(p) *q = 5;
(3) *q = 8;
}
```

Line (1) in `bar` introduces an aliasing relationship between `p` and `q`. The statement guard `p != null` for the assignment `*q = 5` at line (2) implies that neither `p` nor `q`

---

<sup>1</sup>The *lift* operation translates an expression into 32 boolean variables. The condition  $\neg(\textit{lift}(\varepsilon))$  is logically equivalent to  $\varepsilon == \textit{null}$ .

can be `null`; so both `p` and `q` are *guarded locations* for the store into `q` in line (2). On the other hand, since the statement guard is true at line (3), `q` is not guarded for the dereference in line (3).

## 3.2 Detecting Errors Caused by the Dereference of a Pointer Set to Null

This section explains the rules the analysis uses to determine the dereference of a pointer that has been explicitly set to `null` within the analyzed function or in one of its callees, and if there exists a possible execution path that allows the `null` pointer to flow to the dereference site.

Consider a dereference expression  $\varepsilon$  that occurs at  $\rho$  under statement guard  $\psi$ , and let the `null` guard for the pointer that is dereferenced be  $\sigma$ . Recall that  $\sigma$  is `false` unless there is an explicit `null` assignment to this pointer in either the currently analyzed function or in one of its callees. A `null` dereference results if:

$$\boxed{\text{SAT}(\sigma \wedge \psi) \wedge \neg \text{guarded}(\varepsilon, \psi)} \quad (3.2)$$

The first part of this rule states that the `null` guard and the statement guard must not be mutually exclusive, because otherwise there is no possible execution path where a `null` pointer can flow to this dereference site. In addition, for a `null` dereference error to occur, the statement guard should not imply that the dereferenced pointer is never `null`. Note that although these two requirements sound similar, they in fact express different ideas. To understand the difference, consider the example:

```
void no_error(int* p)
{
(1)  p = NULL;
(2)  ...
(3)  if(p) *p = 0;
```

```
}

```

Although the dereference in line (3) is clearly dead code, this code piece nonetheless illustrates the difference between the two requirements for a `null` dereference error. Since the pointer `p` is unconditionally set to `null` in line (1), its `null` guard is `true`. At the dereference point in line (3), the statement guard is the condition `p!=NULL`. The first condition is satisfied, since  $SAT(p \wedge true)$ , but the second condition is not since the statement guard implies that `p` cannot be `null`. Thus, we do not report a `null` dereference error.

Let us consider a few more examples illustrating this rule.

Example 1:

```
void ex1(int* p, bool flag)
{
(1) if(flag) p = NULL;
(2) *p = 5;
}

```

Here, the `null` guard for `p` is `flag` and the statement guard in line (2) is `true`. Since  $SAT(flag \wedge true)$  and since `true` does not imply anything, rule 3.2 applies, and we report a `null` dereference error.

Example 2:

```
void ex2(int* p, bool flag, int x)
{
(1) if(flag) p=0;
(2) if(!flag && x>0) *p=x;
}

```

In this example, the `null` guard for `p` is `flag`, and the statement guard for `*p = x` in line (2) is `!flag && x > 0`. Since  $\neg SAT(flag \wedge (!flag \wedge x > 0))$ , we correctly conclude that there is no `null` dereference error.

### 3.3 Detecting Errors Caused by the Misuse of Conditionals

The rule presented above identifies the dereference of pointers explicitly set to `null`, but does not detect much more common errors resulting from misusing conditionals. Consider the following example:

```
void fatal(int* p)
{
(1) if(!p) *p = 8;
}
```

If the true branch of the conditional expression at line (1) is taken, the variable `p` is known to be `null` although `p` has not explicitly been set to `null`. Thus, the dereference of `p` in line (3) is guaranteed to cause a `null` dereference error if this branch is ever taken. Unfortunately, the rule described in Section 3.1 cannot detect this simple mistake because the `null` guard for the variable expression `p` is always `false`. Since  $c \Rightarrow \neg false$ , i.e.,  $c \Rightarrow true$ , holds for all conditionals  $c$ , the rule presented in the previous section is not sufficient to deal with logical errors of this kind. This observation motivates a second rule for detecting `null` dereference errors.

Errors resulting from logical misuse of conditionals are detected easily by checking if the statement guard  $\psi$  implies the pointer  $p$  that is dereferenced is `null`. We identify these errors by the following rule:

Report error if:	$\neg SAT(\psi \wedge lift(p))^2$	(3.3)
------------------	-----------------------------------	-------

---

<sup>2</sup> $lift(p)$  translates `p` to 32 boolean variables;  $lift(p)$  is logically equivalent to  $p \neq \text{null}$

Even though it seems unlikely that real, widely-used project code would involve these kinds of simple errors, our results reveal this assumption surprisingly unjustified. For example, we found the following `NULL` dereference error in OpenSSL 3.9p1 source code:

```
#define check_store(s, fncode, fname, fnerrcode)
(1) if(s == NULL || s->meth)
(2) {
(3)   STOREerr((fncode), ERR_R_PASSED_NULL_PARAMETER);
(4)   return 0;
(5) }
(6) if(s->meth->fname == NULL)
(7) {
(8)   STOREerr((fncode), (fnerrcode));
(9)   return 0;
(10)}
```

This macro returns an error code if `s->meth` is not `NULL`, but if it is `NULL`, `s->meth` is dereferenced in line (6), causing a real `NULL` dereference error. Our experiments show that amusing examples of this kind are quite common even in supposedly well-tested code bases. Moreover, the above macro is called at more than twenty places in OpenSSL!

### 3.4 Detecting Inconsistencies

As mentioned in Chapter 1, finding inconsistencies in code is a useful way to discover actual bugs. The notion of inconsistency described here is motivated by the concept introduced by Engler et al [3]. Intuitively, a programmer is being inconsistent about the assumptions made about a pointer variable if the possibility that a

pointer may be `null` is acknowledged at one program point, but the same pointer is unconditionally dereferenced at another program point. Consider the code:

```
void bar (int* p)
{
(1) if(p) *p = 8;
(2) ...
(3) *p = 6;
}
```

In line (1), the programmer acknowledges the possibility that `p` may be `null` by checking it before dereferencing it; but this check is omitted before the dereference in line (3). Either the check in line (1) is defensive or there is a real `null` dereference error in line (3). In either case, such inconsistencies reveal confusion about the assumptions made about the pointer and should be signaled as warnings.

In this section, we describe how we detect such inconsistency errors in the SATURN framework without relying on any kind of syntactic matching. First, we present some definitions that allow us to formally define what we mean by an inconsistency; we then discuss how we actually implement the inconsistency checker in the SATURN framework.

### 3.4.1 Guarded Location Sets

We first define *Guarded Location Set* (*GLS*), a concept central to our notion of inconsistency. Every pointer variable in the program has an associated *GLS* that represents its guarded points-to relations. A *guarded location* is a pair  $\langle l, g \rangle$  where  $l$  is a memory location and  $g$  is a guard. The guarded location set of a variable expression  $v$  at program point  $\rho$  is a set of guarded locations:

$$GLS(v, \rho) = \{ \langle l_1, g_1 \rangle, \langle l_2, g_2 \rangle, \dots, \langle l_n, g_n \rangle \}$$

such that variable expression  $v$  at program point  $\rho$  points to memory location  $l_i$  if guard  $g_i$  is satisfied.

In our representation, memory locations are modeled as traces (recall 2.3). Since we maintain the invariant that every memory location has a unique associated trace, we use the terms memory location and trace interchangeably.

To clarify the notion of guarded location sets, we proceed with a simple example.

```
void bar (int* p, int flag)
{
(1) if(flag) p = malloc(sizeof(int));
(2) else p=q;
}
```

We show the guarded location set of  $p$  at the entry and exit points of the function `bar`:

$$GLS(p, entry) = \{ \langle drf\{arg_1\}, true \rangle \} \quad (\text{Entry})$$

$$GLS(p, exit) = \{ \langle drf\{new\_loc\}, flag \rangle, \langle drf\{arg_2\}, \neg flag \rangle \} \quad (\text{Exit})$$

An important point about our definition of guarded locations is that the guard does not encode the full path sensitive information. Rather, the guard encodes only the path sensitivity related to the points-to relation. To clarify the distinction, consider the following example:

```
void foo (int* p, int* q, bool assign_flag, bool deref_flag)
{
(1) if(assign_flag) p = q;
(2) if(deref_flag) *p = 0;
```

}

In the true branch of the conditional at line (2), (i.e. at the expression `*p` at program point P2), the *GLS* of `p` is:

$$GLS(p, P2) = \{ \langle drf\{arg_1\}, \neg assign\_flag \rangle, \langle drf\{arg_2\}, assign\_flag \rangle \}$$

The guards in the *GLS* do not encode information about the current statement guard, which in this case has the information that `deref_flag` is true. The *GLS* guards only encode those conditionals that affect the points-to relations. To distinguish between statement guards and the guards in the guarded location sets, we call the latter *memory location guards*. It is important to keep in mind throughout this chapter that expression guards and memory location guards are distinct. This distinction is important to our method for detecting inconsistencies.

Another final point worth mentioning about guarded location sets is their relevance to aliasing information they represent. If two pointer variables  $v_1$  and  $v_2$  have identical guarded location sets, involving at least one non-null location, they must necessarily alias each other.

### 3.4.2 Defining an Inconsistency

In this section, we refine our intuitive notion of inconsistency into a formal description. Consider two expressions  $\varepsilon_1$  and  $\varepsilon_2$  at program points  $\rho_1$  and  $\rho_2$  with expression guards  $\psi_1$  and  $\psi_2$ , respectively. An inconsistency error exists if all of the following conditions are satisfied:

- |  |       |
|--|-------|
| <ol style="list-style-type: none"> <li>1. <math>GLS(\varepsilon_1, \rho_1) = GLS(\varepsilon_2, \rho_2)</math></li> <li>2. <math>guarded(\varepsilon_1, \psi_1)</math></li> <li>3. <math>\neg guarded(\varepsilon_2, \psi_2)</math></li> </ol> | (3.4) |
|--|-------|

This set of rules conveys the following idea: Given two expressions with identical

points-to relations, it is possible for one of them to dereference `null` under its expression guard while it is impossible for the other to dereference `null` under its expression guard. Consider the following simple example:

```
void bar(int* p)
{
(P1) *p = 6;
(P2) if(p) *p = 8;
}
```

We explain how the rules presented above detect the consistency in this example. First, note that the *GLS* of `p` at both P1 and P2 is the set  $\langle drf\{arg_1\}, true \rangle$ . Next, at P1:

$$guarded(p, true) = false$$

since the statement guard `true` does not imply that `p` cannot be `null`, but at P2, we have:

$$guarded(p, p) = true.$$

The above example illustrates why it is advantageous not to incorporate full path sensitivity into the memory location guards. If we had forced memory location guards to be a more restrictive form of the statement guard and encode full execution history, the two occurrences of the variable `p` in the example given above would have two distinct guarded location sets, and we could not have discovered that the two variable expressions identify the same value.

Another advantage of detecting inconsistency errors using the rules given above is that we can automatically identify inconsistency errors that involving pointer aliases. Consider a slight modification of the same example:

```
void bar(int* p, int* q)
{
(P1) q = p;
(P2) *q = 6;
(P3) if(q) *p = 8;
}
```

Just as in the previous example, this code is inconsistent about the assumptions made about memory locations accessible through `p`, even though detecting this inconsistency requires the ability to track aliasing relations between pointers. Because of our guarded location set based approach to detecting inconsistency errors, discovering this inconsistency is no harder than discovering the inconsistency in the previous example. On the other hand, an approach that detects inconsistencies by syntactically matching on condition expressions would fail to realize that `p` and `q` are aliases of each other and would not report an inconsistency error.

In order to illustrate further advantages of detecting inconsistency errors this way, we consider a final example:

```
void foo(int* p, int* q, bool flag)
{
(1)  flag = (p!=NULL);
(2)  q = p;
(3)  if(flag) *p=8;
(4)  * q = 4;
}
```

As this example illustrates, a programmer can acknowledge that a pointer can be `null` in various different ways. Rather than using the more common idiom `if(p)` to acknowledge that `p` can be `null`, the programmer sets the boolean `flag` to true only if `p!=NULL`. Furthermore, this example introduces an aliasing relationship between `p`

and `q`. Despite this non-conventional way of checking that `p` is not `null` in line (3), our analysis is still able to detect the inconsistent assumptions made about variables `p` and `q`.

# Chapter 4

## The Inter-procedural Null Analysis

So far, our discussion about the `null` dereference analysis ignored inter-procedural communication across function boundaries. In this chapter, we discuss techniques used by our checker to perform an inter-procedural analysis.

As discussed earlier, the SATURN framework uses a summary-based approach to inter-procedural analysis. Each checker abstracts relevant properties of interest to be included in its function summaries. At call sites, the summary associated with the called function is retrieved and the relevant behavior of the function is incorporated in the context of the caller.

### 4.1 Interface Objects

Before we discuss the summary representation for the `null` dereference analysis, we first introduce *interface objects (IO)*. Interface objects are values shared between callers and callees. Global variables, formal parameters of functions, and return values of functions are all used to share information across function boundaries; we call these *primary interface objects (PIO)*. Local variables, on the other hand, are local to one function and are not interface objects. Note that primary interface objects are not the only channels of communication between functions. Any fields or any memory locations accessible through primary interface objects are also interface objects. This

discussion motivates the following definition:

$$PIO := global \mid param_i \mid ret\_val \quad (4.1)$$

$$IO := PIO \mid *IO \mid IO.f \quad (4.2)$$

In the grammar for  $PIO$ ,  $param_i$  denotes the  $i$ th parameter of a function prototype, and  $ret\_val$  denotes the return value of a function.. The recursive definition of  $IO$  defines an interface object to be any chain of pointers with a primary interface object as its root.

An important issue in discussing interface objects is how we translate between the different name spaces of callers and callees. For example, the first parameter of the called function `foo` may correspond to a local variable or a different parameter of the calling function `bar`. We will not discuss this translation mechanism in this chapter and assume that we are able to correctly convert interface objects to native objects in the caller's name space. A brief discussion of how this translation is done in SATURN is outlined in [9].

## 4.2 Summary Representation

As far as the `null` dereference analysis is concerned, the program properties of interest that need to be communicated across function boundaries are: which interface objects are dereferenced inside a called function (*dereference summary*); whether the return value of a function may or may not be `null` (*return summary*); the aliasing introduced by the called function (*alias summary*); and finally the side effects of the called function (*side effect summary*). We discuss these summaries in detail in the rest of this chapter.

It is useful to differentiate between aliasing and side effects in our discussion. An aliasing relationship is introduced by a called function if two interface objects that did not have the same points-to relations on function entry have identical points-to

relations on function exit. Side effects, for our purposes, do not involve aliasing. A side effect is introduced by a function if an interface object has a different points-to relation on function exit than the one it had on function entry, and furthermore, there is no *may-alias relation* between this interface object and any of the other interface objects [6]. Common examples of side effects are fresh memory allocations of a global variable or parameter, and assignments of interface objects to `null`.

### 4.3 Dereference Summary

Since we do not use annotations in the current implementation of the `null` checker or pass information top-down about which pointers may be `null`, we assume that any interface object that is a pointer has the possibility of being `null` on function entry. Unless the statement guard implies that this pointer value cannot be `null`, we conservatively assume that any dereference expression may result in a `null` dereference error. Thus, pointers that are dereferenced in a function must be included in this function's summary so that its behavior can be reconsidered within a particular calling context.

Consider a dereference expression  $\varepsilon$  involving an interface object at program point  $\rho$  under the statement guard  $\psi$ . This dereference expression is a *potential null dereference* if  $\text{guarded}(\varepsilon, \psi)$  does not hold.

We also need a *taint* operation on interface objects, which returns a boolean formula for the conditions under which a potential `null` dereference of that interface object occurs. To evaluate the result of the taint operation, we introduce the following rules:

**Taint Algorithm**

1. At function entry, do:

$$\forall \phi \text{ taint}(\phi) := \text{false}$$

where  $\phi$  denotes an interface object.

2. For every potential `null` dereference of an interface object  $\phi$  under the statement guard  $\psi$ , do:

$$\text{taint}(\phi) := \text{taint}(\phi) \vee \psi$$

At function exit,  $\text{taint}(\phi)$  yields a boolean expression that represents the conditions under which an interface object  $\phi$  is dereferenced. To decide whether an interface object should be included in the function summary, we use the following heuristic:

**Dereference Summary Generation**

*If  $UNSAT(\neg \text{taint}(\phi))$ ,  $\phi \in \text{deref\_summary}$*

This rule states that only pointers that are unconditionally dereferenced are included in the function summary. Note that unconditional dereference of an interface object does not imply that the statement guard is *true* at the dereference site. Consider the example:

```
void bar(int* p, bool flag)
{
(1) if(flag) *p=2;
(2) else *p=5;
}
```

Because `p` is dereferenced on both branches,  $\text{taint}(\text{arg}_1) = \text{true}$  at function exit; thus, the interface object  $\text{arg}_1$  is unconditionally dereferenced, although the statement guard is `true` at neither dereference site.

The heuristic given above is unsound. For example, any `null` pointer that is conditionally dereferenced inside a called function will not be tracked by the `null` checker, potentially yielding false negatives as in the example below:

```
void bar(int* p, bool flag)
{
(1) if(flag) *p=2;
}
void foo()
{
(1) bar(null, true);
}
```

Clearly, the call to `bar` in line (1) in `foo` results in a `null` dereference error that does not trigger a warning by the analysis, resulting in a false negative.

Since most functions involve numerous pointer dereference expressions, incorporating every dereference of a potentially `null` interface object  $\phi$  with its corresponding guard, i.e.  $taint(\phi)$ , may be prohibitively expensive and may not scale to long call chains. Thus, we currently rule out a path-sensitive dereference summary as a realistic alternative. Another possible heuristic that can be used instead of the one described above in generating the dereference summary is including every interface object  $\phi$  where  $taint(\phi)$  is satisfiable. The disadvantage of this approach is that it results in an unacceptable number of false positives, although it would not contribute to sources of unsoundness in the analysis.

We can eliminate many of these problems by requiring user annotations that guide the analysis about which parameters or globals are assumed never to be `null` and which can potentially be `null`. This information can then be used to be more selective in choosing which interface objects to incorporate in the dereference summary.

Although annotations are somewhat cumbersome for the programmer, annotating the assumptions made about parameters and globals would be valuable in both scaling the analysis and reducing sources of unsoundness.

## 4.4 Return Summary

The return summary for a function is a boolean value that states whether it is possible for this function to return `null`. The return value summary is generated according to the following rule, where  $\Omega$  denotes a return expression at program point  $\rho$  with statement guard  $\psi$ .

### Return Summary Generation

$$\text{return\_sum} := \exists \Omega. SAT(\psi \wedge \text{points\_to\_guard}(\Omega, \rho, \text{drf}\{\text{null}\}))$$

The return summary is the boolean value `true` if there exists a feasible execution path through which the function can return `null`, otherwise the return summary is `false`.

To simulate the effect of the return summary, we first collect a set  $S$  of all functions whose return summaries are `true`. We generate an error if there exists a dereference expression  $\varepsilon$  at program point  $\rho$  with guard  $\psi$  such that the following condition holds:

### Error Condition for Return Summary

$$\tau := \text{points\_to}(\varepsilon, \rho, \psi)$$

Generate an error message if:

$$\exists t. (t \in \tau \wedge t = \text{return}(f) \wedge (f \in S) \wedge \neg \text{guarded}(\varepsilon, \psi))$$

This rule states that the dereference of the return value from a function which can return `null` must always be guarded, otherwise the analysis issues a warning.

It is important to note that our return summary represents a *may-return-null* rather than a *must-return-null* condition. This assumption may cause false positives as the following example illustrates:

```
void bar(bool flag)
{
(1) if(flag) return NULL;
}

void foo()
{
(1) int* p = bar(false);
(2) int a = *p;
}
```

Since the trace of `p` is the return value from a function which may return `null`, and since the statement guard does not imply that `p` cannot be `null`, the checker will generate an error message, resulting in a false positive. In order to eliminate false positives of this kind, we would either need programmer annotations or path sensitive information across function boundaries.

## 4.5 Alias Summary

There is a separate inter-procedural alias analysis written in the SATURN framework; the `null` dereference analysis uses the results of this may-alias analysis [1]. To see why inter-procedural alias information is important for the `null` dereference analysis, consider the simple example:

```
void bar(int** p, int** q)
{
```

```
(1) *p = *q;
}

void foo(int* p, int* q)
{
(1) p = NULL;
(2) bar(p, q);
(3) *q = 8;
}
```

In this example, `bar` introduces an aliasing relation between `p` and `q` such that after `bar` returns, `q` is `null`. Thus, the dereference in line (3) causes a `null` dereference error. Without the ability to track inter-procedural aliasing, it would not be possible to detect such errors.

## 4.6 Side Effect Summary

The previous alias analysis written in the SATURN framework does not track assignments of `null` and assignment of newly allocated memory locations to pointers. Since this behavior is very important for the `null` dereference analysis, we build a separate side-effect analysis in addition to the existing alias analysis that only tracks `null` assignments and assignments of newly allocated memory to pointers. To understand the importance of side effects for the `null` analysis, consider the code:

```
void bar(int** p)
{
(1) if(!*p) *p = malloc(sizeof(int*));
}

void foo(int* p, int* q)
{
```

```
(1) p = NULL;
(2) bar(p);
(3) *p = 8;
}
```

In this example, `p` is initialized to `null`, but the call to `bar` in line (2) assigns newly allocated memory to `p`. Thus, at the dereference site in line (3), `p` is no longer `null`, and the code is free of `null` dereference errors, assuming that `malloc` does not return `null`. However, without tracking the side effect introduced by `bar`, we would report a false alarm. Since this pattern of initializing invalid pointers via function calls is a common programming practice, ignoring side effects would result in a large number of false positives.

Similarly, `null` assignments to pointers inside function calls is also important for tracking `null` dereference errors. Consider the example:

```
void set_null(int** p)
{
(1) *p = NULL;
}
```

```
void error(int* p)
{
(1) set_null(p);
(2) *p = 8;
}
```

After the call to `set_null` in the line (1) of `error`, `p` has the value `null`. Thus, dereferencing it in line (2) causes a `null` dereference error, which would be not be detected if the analysis ignored inter-procedural side effects.

This discussion motivates the need for an algorithm to compute side-effects. To describe this algorithm, we need a new function *trace\_deref* that retrieves the points-to set for a given trace. Note that *trace\_deref* has the same functionality as *points\_to* except that it returns the points-to set for a given trace rather than for an expression. Let  $t$  be a trace and  $\rho$  be a program point. We define:

$$\text{trace\_deref}(t, \rho) = \tau$$

such that trace  $t$  may point to any of the traces in set  $\tau$  at program point  $\rho$ . Although *trace\_deref* is operationally very similar to *points\_to*, an example is still helpful:

```
void traces(int*** p, int** q, int** z, bool flag1, int* flag2)
{
(P1) if(flag1) *p = q;
(P2) if(flag2) q = z;
(P3) ...
}
```

At line (P3), we have:

$$\text{trace\_deref}(\text{drf}\{p\}, P3) = \{\text{drf}\{\text{drf}\{p\}\}, \text{drf}\{q\}, \text{drf}\{z\}\}$$

indicating the fact that the trace  $\text{drf}\{p\}$  can point to any of the traces on the set returned by *trace\_deref*.

To discuss our side effect computation algorithm, we introduce the concept of *visibility*. A trace is *visible* outside the calling context if:

1. Its root trace is a **global** or
2. Its root trace is an **arg** and the trace involves at least one **drf** operation.

Since parameters are passed by value in C, an assignment to a root trace value **arg** does not introduce a side effect; but a store into a parameter results in a visible

side effect in the caller. Thus, visible traces describe traces which can be modified by a called procedure and the modification is visible in the calling context.

We now present a naive first attempt to compute side effects:

**A Naive Algorithm to Compute Side Effect Summary**

let  $l$  be a pointer interface object that is visible outside the current procedure, and let  $\gamma$  represent the exit point <sup>1</sup> of function  $F$ .

$\tau := \text{trace\_deref}(l, \gamma)$

$\forall t \in \tau$  if  $\text{root-trace}(t) \in \{\text{null}, \text{new\_loc}\{*\}\}$ ,  $\langle l, t \rangle \in$  side effect summary

This algorithm considers the points-to set for every trace  $l$  that is accessible from a visible pointer interface object. If  $l$ 's points to set contains either `null` or a trace whose root is newly allocated memory, then the algorithm adds this side effect to the side effect summary.

Unfortunately, this algorithm is unable to identify all side effects of a function. We illustrate the problem with the following example:

```
struct state
{
  struct state* next;
  int* data_ptr;
}

void foo(struct state** s)
{
  (1) *s = malloc(sizeof(struct state))
  (2) (*s)->data_ptr = malloc(sizeof(int))
}
```

---

<sup>1</sup>Although a function may have multiple exit points, we can transform the function into a function with a single exit point by introducing additional conditional branches. See [7].

After the assignment in line (1), `(*s)->data_ptr` points to `drf{new_loc{1}}`; thus the assignment in line (2) changes the points-to-information for `field{drf{new_loc{1}}, data_ptr}` rather than the points-to set for `field{drf{drf{arg_0}}, data_ptr}`. As a result, the points-to set for the trace `field{drf{drf{arg_0}}, data_ptr}` at function exit is the same as its initial points-to set, and the side effect introduced in line (2) is not detected by the naive algorithm presented above.

Before we present a revised version of the initial algorithm, we first introduce some useful concepts. As we saw in the previous example, traces with root `new_loc` can potentially be visible outside the currently analyzed function, and thus changes in the points-to set of traces with root `new_loc` can be important to the caller function. We define *potential interface object*,  $IO^?$  as:

$$IO^?(l) = \text{new\_loc}\{id\} \mid *l \mid l.f$$

To determine whether a potential interface object is actually visible outside the calling context, we need the notion of *k-reachability*.

**1-reachable:** A potential interface object  $IO^?$  with root  $R$  is *1-reachable* from an interface object  $IO$  if the following holds:

$$\text{drf}\{R\} \in \text{trace\_deref}(IO, \text{exit})$$

**k-reachable:** Any potential interface object  $IO_1^?$  is *k-reachable*

from an interface object  $IO$  if there exists another potential interface object  $IO_2^?$  with root  $R_2$  such that  $IO_2^?$  is  $(k - 1)$ -reachable from an interface object and  $R_1$  is 1-reachable from  $R_2$ .

We say that a potential interface object is *reachable* if it is  $k$ -reachable from an interface object for any  $k$ , otherwise we say it is *not reachable*.

We consider an example that illustrates k-reachability:

```
void foo(int*** p)
```

```

{
(P1) *p = malloc(sizeof(int**)) /* Trace of new memory:  new_Loc{1}*/
(P2) **p = malloc(sizeof(int*); /*Trace of new memory:  new_Loc{2} */
}

```

In this example,  $new\_Loc\{1\}$  is 1-*reachable* from  $arg_1$  since  $drf\{new\_Loc\{1\}\} \in trace\_deref(arg_1, P1)$ . Also,  $new\_Loc\{2\}$  is 2-*reachable* from  $arg_1$  since  $new\_Loc\{2\}$  is 1-*reachable* from  $new\_Loc\{1\}$ , and  $new\_Loc\{1\}$  is 1-*reachable* from  $arg_1$ .

Next, we define the *replace* operation:

$replace(\pi, r)$ : Replaces the root of trace  $\pi$  with root  $r$ .

For example,

$$replace(field\{drf\{new\_Loc\{1\}\}, ptr\}, arg_1) = field\{drf\{arg_1\}, ptr\}$$

Finally, we introduce the *reach* operation, which generates a sequence of valid replacements that produce an interface object from a  $k$ -*reachable* potential interface object  $\pi$ :

$reach_1(\pi) = replace(\pi, l)$  where  $\pi$  is a potential interface object 1-*reachable* from interface object  $l$ .

$reach_k(\pi) = replace(\pi, reach_{k-1}(l))$  if  $\pi$  is 1-*reachable* from potential interface object  $l$  and  $l$  is  $(k-1)$ -*reachable* from an interface object.

Reconsider the previous example:

```

struct state
{
struct state* next;
int* data_ptr;
}

```

```

void foo(struct state** s)
{
(1) *s = malloc(sizeof(struct state)) /*New trace: new_loc{1}*/
(2) (*s)->data_ptr = malloc(sizeof(int)) /*New trace: new_loc{2}*/
}

```

We have:

$$\begin{aligned}
& reach_2(\{new\_loc\{2\}\}) \\
&= replace(new\_loc\{2\}, (reach_1(field\{drf\{new\_loc\{1\}\}, data\_ptr\})) \\
&= replace(new\_loc\{2\}, (replace((field\{drf\{new\_loc\{1\}\}, data\_ptr\}), arg_1)) \\
&= replace(new\_loc\{2\}, (field\{drf\{\{drf\{arg_1\}\}, data\_ptr\})) \\
&= field\{drf\{\{arg_1\}\}, data\_ptr\}
\end{aligned}$$

We now revise the first algorithm presented for computing side effects. The revised algorithm considers not only interface objects, but also all reachable potential interface objects to compute the side effects at the exit point of the function.

#### Revised algorithm to Compute Side Effect Summary

let  $l$  be any pointer  $IO$  visible outside the current function;

let  $l'$  be any  $IO^?$  reachable from a visible pointer interface object.

let  $\gamma$  represent the exit point of function  $F$ .

For each  $l$  compute:

$$\tau := trace\_deref(l, \gamma)$$

$$\forall t \in \tau. \quad \text{if } root\text{-}trace(t) \in \{\text{null}, new\_loc\{*\}\}, \langle l, t \rangle \in \text{side effect summary}$$

For each  $l'$  compute:

$$r = reach(l')$$

$$\tau := trace\_deref(l', \gamma)$$

$$\forall t \in \tau. \quad \text{if } root\text{-}trace(t) \in \{\text{null}, new\_loc\{*\}\}, \langle r, t \rangle \in \text{side effect summary}$$

We can compute the side effects of a function using this algorithm. Note that this algorithm is a *may-cause-side-effects* algorithm rather than *must-cause-side-effects*

algorithm.

We reconsider the previous example and show how the algorithm computes a side-effect-summary for the function `foo`.

```
void foo(struct state** s)
{
(1) *s = malloc(sizeof(struct state)) /*New trace: newLoc{1}*/
(2) (*s)->data_ptr = malloc(sizeof(int)) /*New trace: newLoc{2}*/
}
```

In this function,  $newLoc\{1\}$  is a potential interface object 1 – *reachable* from  $drf\{arg_1\}$ ; similarly,  $field\{drf\{newLoc\{1\}\}, data\_ptr\}$  is a potential interface object reachable from  $field\{drf\{drf\{arg_1\}\}, data\_ptr\}$ , an interface object visible outside of `foo`. (See the previous example for the derivation.) Since both traces point to newly allocated memory cells, the side-effect computation algorithm adds the pairs  $\langle drf\{arg_1\}, drf\{newLoc\{1\}\} \rangle$  and  $\langle field\{drf\{drf\{arg_1\}\}, ptr\}, drf\{newLoc\{2\}\} \rangle$  to the side effect summary of `foo`.

# Chapter 5

## Results

We have tried our `null` dereference analysis on some widely-used open source projects, such as OpenSSH, OpenSSL, and Sendmail. The table below summarizes the results of our runs<sup>1</sup>:

Software	Real errors	False Positives	Other <sup>2</sup>	False positive rate
OpenSSH-3.9p1	9	1	0	11.1%
OpenSSL-0.9.8a	47	6	2	10.9%
Sendmail-8-13-6	8	1	0	12.5 %

These experiments reveal some of the advantages of our approach. Most static analysis tools do not take into account aliasing between pointer variables, assuming that aliasing relationships do not typically lead to `null` dereference errors in code. Our experiments show that this assumption is unjustified. For example, about 10 of the 47 errors reported for the OpenSSL project resulted from aliasing relationships introduced between two pointer variables. However, most of the aliasing introduced between pointers was local to one function and did not involve inter-procedural aliasing.

---

<sup>1</sup>The error counts presented in this table do not include duplicate errors, such as those resulting from code that was copy-and-pasted in different parts of the code.

<sup>2</sup>This includes all error reports which we could not decide whether they were actual errors or not. In any case, they indicated bad programming style.

Our method for checking inconsistencies without relying on syntactic matching on `if` statements also proved to be valuable for catching subtle errors. We present an error reported by our tool for a function called `getmxrr` in Sendmail that illustrates the advantages of our approach:

```
/* From sendmail-8.13.6/sendmail/domain.c:getmxrr */
getmxrr(char* host, char** mxhosts, unsigned short mxprefs, ...)
{
238:  if (*host == 0) return 0;
...
278:  if (tTd(8, 1))
279:    sm("getmxrr: res_search(%s) failed (errno=%d, h_errno=%d)",
280:    host == NULL ? "<NULL>" : host, errno, h_errno);
}
```

In this code fragment, the programmer acknowledges the possibility that `host` may be `null` by comparing it against `null` in line 280, but `host` is dereferenced without being checked for `null` in line 238. Since we don't rely on syntactic matching on particular programming idioms, our analysis is able to detect these kinds of subtle errors as well.

Most of the false alarms generated by our tool result from features of the C language that we do not model. For example, in the SATURN representation, all array elements are represented with the same abstract memory location. Consequently, our tool generates false alarms if the programmer distinguishes between different array elements and makes different assumptions about which array elements may be `null`.

Another common source of false alarms results from the usage of macros in C code. Since macros are written for generic purposes, most macros that take pointer variables as input ensure that they are not `null` before dereferencing these pointers. On the

other hand, functions that call these macros generally make more specific assumptions about pointers, typically because these pointers are local variables which are known not be `null`. However, since we do not distinguish between code that belongs to a macro and code that is native to a function, we report inconsistency errors if macros are conservative about checking pointers for `null`.

# Chapter 6

## Future work

A first immediate future goal of this project is to run more experiments with our `null` dereference analysis on larger open source projects, such as the Linux Kernel, OpenBSD, and `gcc`. Also, as mentioned throughout this thesis, the `null` dereference analysis described here is not sound in its current form. Another eventual goal of this project is to verify the absence of `null` dereference errors in large projects written in C.

# Bibliography

- [1] Alex Aiken Brian Hackett. How is aliasing used in systems software? <http://theory.stanford.edu/~aiken/publications/new/aliasinguse.pdf>.
- [2] Andy Chow Chou. *Static analysis for bug finding in systems software*. PhD thesis, 2003. Adviser-Dawson Engler.
- [3] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.
- [4] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press.
- [5] Brian Hackett. *CLPA Manual*.
- [6] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, 2004.
- [7] George Necula, Scott McPeak, S. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, 2002.
- [8] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on*

*Foundations of software engineering*, pages 115–125, New York, NY, USA, 2005. ACM Press.

- [9] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, 2005.
- [10] Yichen Xie and Dawson Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, 2003.
- [11] Alex Aiken Yichen Xie. Saturn: A sat-based tool for bug detection. In *Lecture Notes in Computer Science*, volume 3576, pages 139–143. Springer, July 2005.