Detecting Null-dereference Bugs via a Backward Analysis

Qian Wang
State Key Laboratory of Networking and Switching Tech, Beijing University of Posts and Telecommunications
Beijing 100876, China
Email: qwang_bupt@163.com

Dahai Jin, Yunzhan Gong, Hongbo Zhou
State Key Laboratory of Networking and Switching Tech, Beijing University of Posts and Telecommunications
Beijing 100876, China
Email: {jindh, gongyz}@bupt.edu.cn, zhouhb84@yahoo.cn

***Abstract*—Null dereference is a common occurring bug in programming languages such as C. In this paper, we propose a path-sensitive and context-sensitive approach that performs a backward dataflow analysis to identify null-dereference bugs. One novel feature of our approach is that with the help of aliasing predicates, it can perform strong updates in presence of aliasing, thus eliminating false positives. The aliasing predicates are introduced on the premise of a canonical representation for program being analyzed. Moreover, a context-sensitive algorithm for inter-procedural null-dereference analysis is also presented in this paper, which also contributes to improve accuracy. We have implemented this approach, and give an evaluation of it on a set of open source benchmarks. The experimental results verify the effectiveness of our approach, and show that it is suitable for exploring large real programs with reasonable accuracy.**

***Index Terms*—Null-dereference Analysis, Aliasing, Strong updates, Context-sensitive Inter-procedural Analysis**

```
[1]:    foo(rec *x, rec *z) {
[2]:    z = NULL;
[3]:    p = &z;
[4]:    *p = x->f
[5]:    if(c) {
[6]:      b = 0;
        }
        else {
[7]:      b = 1;
        }
[8]:    if(b) {
[9]:      y = z;
        }
        else {
[10]:     y = x->f;
        }
[11]:   *y = 1 ;
        }
```

Figure 1. Example code

I. Introduction

Null dereference is a kind of bug that commonly occurs in programs, and many static tools and approaches have been developed for detecting such bugs (e.g. [1, 2, 5, 7, 13]). However, it's not an easy work to achieve the detection in an accurate and efficient way. Aliasing is something that one cannot ignore when doing the null-dereference analysis [19]. Failure to take into account aliasing can limit the usefulness of an approach. Furthermore, strong updates are required for precision, but are difficult to perform in presence of aliasing. Even pre-computed may-alias and must-alias information may not enable strong updates enough, since at a given program point two variables may be aliased under some paths and not aliased under other paths. In addition, inter-procedural analysis also needs to be considered, for the occurrences of the null-dereference bugs often involve interactions among multiple procedures, [1].

In this paper, we propose a bug-detection approach which is context-sensitive and path-sensitive, to address the problem of identifying the null-dereference bugs in C programs. Starting at a dereference point in the program be analyzed, our approach propagates a set of symbolic states backwards along the control flow graph (CFG), to find whether there are sufficient bases to report this dereference as a possible bug. The symbolic state takes predicates as a condition, under which the value held by some witness (which represents a single memory location) is null and will flow into the initial dereference point. Once the condition is satisfied, a null-dereference bug is found. During the backward analysis, the symbolic state may be updated due to the effects of the assignment statements, and the predicates can be evaluated according to several custom-defined rules rather than a constraint solver. In addition, branch correlations are also taken into account to realize the path-sensitive analysis. A backward analysis only explores the program paths that are relevant to analyzing a dereference point, which makes our approach scalable.

A novel feature of our approach is that by means of aliasing predicates, our approach achieves to perform

Manuscript received May 20, 2013; accepted August 24, 2013.
Project number: 2012AA011201, 61202080

strong updates in presence of aliasing. Considering a certain program point at which two l-value expressions [12] may be aliased with each other, our approach takes advantage of aliasing predicates to give a pair of hypotheses: one is that the aliasing relationship between the two expressions is definitely established; and the other is opposite. Then both the hypotheses can be respectively validated and invalidated by the backward analysis when it arrives at the statements that confirm or contradict the hypotheses. As a result, superfluous symbolic states are excluded and strong updates are accomplished, thus reducing many false positives.

The aliasing predicates are introduced on the premise that all the expressions in the program being analyzed are translated into a canonical representation. Owing to this canonical representation, our approach can explicitly model the address of a memory location that an l-value expression refers to. Furthermore, our approach utilizes constraints on such addresses to determine whether l-value expressions are aliased with each other or not. These constraints are just the so-called aliasing predicates.

In addition to aliasing, interactions among multiple procedures introduces another complication that a dereference operation is directly or indirectly associated with a parameter, for the possible values of the parameter can be only known after all the call sites examined. Furthermore, procedural side effects and return values also have non-ignorable effects on data flows, which one needs to pay attention to when performing inter-procedural analysis.

Our approach achieves to perform inter-procedural analysis in a context-sensitive manner through the way of partial transfer functions, or summary table [9, 10], with a few modifications to adapt backward traversal. The main idea is that to perform efficient analysis of called procedures, our approach computes and saves summary information at call sites; by reusing summary information, it avoids reanalyzing a procedure in a context in which the procedure has been analyzed previously.

We have implemented a prototype of our approach as an extension of Defect Testing System (DTS) [11], which is a general automatic bug-detection framework for C programs, especially GCC programs. To estimate the effectiveness of our approach, we apply this prototype on a set of open source GCC benchmarks. The preliminary experimental results are encouraging, for more bugs unknown before are found, and meanwhile false positives are reduce.

In summary, the main benefit of our approach is that it enables an accurate and efficient null-dereference analysis for C programs. That is it detects as many potential bugs as possible; it performs strong updates in presence of aliasing, hence eliminates false positives that are identified by the one depending on weak updates; and it scales to large programs. The main contributions of this paper include: (1) a novel set of designed features that together enable a bug-detection approach for null dereferences in real C programs with reasonable precision; (2) an implementation of this approach; (3) experiment studies, using large open-source, that illustrate the effectiveness and usefulness of this approach.

The rest of this paper is organized as follows: Section II explains the canonical representation introduced by our approach. And the main features of our approach are illustrated in section III, which also gives the algorithm for inter-procedural null-dereference analysis. Section IV first discusses the experimental environment and then reports on preliminary experimental results obtained from analyzing 5 open source GCC benchmarks. In section V, we survey related work, and conclude the paper in section VI.

II. Canonical Representation

After the abstract syntax tree is built, all the expressions in the program under test are translated into a canonical representation with the following syntax:

Constant	$c \in Const$	
Primitive address	$a \in Addr$	
Allocative address	$t \in Alloc$	
Expression	$e \in Expr$	$e ::= c \mid a \mid t \mid {*e} \mid e\#f$

There are five kinds of expressions in the canonical representation: constant *c*, primitive address *a*, allocative address *t*, dereference expression **e* and offset expression *e#f*. Constants indicate numeric constants, string constants or *sizeof* expressions in the source code. Primitive addresses and allocative addresses respectively model symbolic addresses of variables and memory allocation sites. What should be pay attention to is that there is one address t_{malloc} per allocation site, and one symbolic address a_x for each variable *x*. A dereference expression **e* denotes the value of the memory location that expression *e* points to. And an offset expression *e#f* means an address in memory which is obtained by adding an offset *f* to the base location that expression *e* refers to. In addition, arithmetical and relational operations among expressions are also taken into account, but not explained here for simplicity. If *x* is a variable in the source code, then the C expression *&x* is described as a_x; expression *x*, as $*a_x$; expression **x,* as $**a_x$; expression *x.f*, as $*(a_x\#f)$; and expression *x->f*, as $*((*a_x)\#f)$.

An l-value in C programs is a kind of expression that refers to a memory location [12]. Such an expression is translated into a dereference expression **e* in the

Formula	::=	Formula ∨ Conjunct ∣ Conjunct
Conjunct	::=	Conjunct ∧ Predicate ∣ Predicate
Predicate	::=	Term op Term ∣ (Formula)
Predicate	::=	¬Predicate
Term	::=	true ∣ false ∣ e
op	::=	!= ∣ == ∣ > ∣ < ∣ ≥ ∣ ≤

Figure 2. Structure of formula. The term *e* denotes an arbitrary expression in canonical representation.

canonical representation. The name l-value comes from the assignment $e_1 = e_2$ in which the left operand e_1 must be an l-value expression. Furthermore, for an l-value $*e$ in the canonical representation, the expression e explicitly holds the address of the memory location that $*e$ represents.

Aliasing is something that one cannot ignore when performing null-dereference analysis. And it is usually defined as follow:

- Two l-value expressions are aliased if and only if they refer to the same memory location [19].

Considering what has been mentioned above, in the canonical representation, the definition of aliasing can be evolved as:

- Two l-value expression $*e_1$ and $*e_2$ are aliased if and only if the predicate $e_1 == e_2$ (which means the address of the memory location that $*e_1$ refers to should be equal to the address of the one that $*e_2$ refers to) is valid.

In the rest of the paper, such kind of predicate is regarded as an aliasing predicate. And with the help of it, our approach can perform strong updates in presence of aliasing, thus enhancing precision of the null-dereference analysis. The details will be explained in section III.A.

Besides the work above, for facilitating subsequent backward analysis, our approach also takes some other measures to optimize the structure of the source code. For instance, it introduces temporary variables to eliminate side effects in expressions and flatten nested procedure calls; and it converts short-circuit operators such as &&, ||, and ?: into if-else statements to eliminate control flow within expressions; and moreover, as shown in Figure X, it also adds two extra expressions to explicitly denote the true and the false branch conditions of an if-else statement (similar measures are taken for other selection statements).

III. Null-Dereference Analysis

Our approach is composed of a backward data-flow analysis. Starting from a dereference point, it propagates a series of symbolic states backwards along the control flow graph (CFG) to identify whether there is a NULL value that may be eventually transferred to the initial dereference point along some path, which implies that a null-dereference bug may occur.

The symbolic state is of the form a tuple $<w, es>$ that consists of two components:

- Witness: the witness w, which is generally a dereference expression in our implementation, refers to a single memory location. It currently holds the value of interest that may flow into the initial dereference point. Besides, there are two special symbols ε and η for witness. The value of the former is definitely NULL and the value of the latter opposite.
- Execution state: the execution state es is a formula which is defined in Figure 2. It represents the condition under which a null-dereference bug might occur. Specially, a predicate in the execution state is called as a root predicate as long as it constraints the witness.

EXAMPLE 1. Consider the code shown in Figure 1 and we will use it as an example to demonstrate how our approach works. The CFG built for this code is shown in Figure 3, where all the expression has been translated into

TABLE I.
WITNESS TRANSFORMATIONS AS WELL AS SOME CORRESPONDING ALTERNATIONS IN EXECUTION STATE.

Statement		Witness transformation
Assignment	$*e_1 = *(e_2\ldots\#f)$	$< w[*(e_2...\#f) / *e_1], es[*(e_2...\#f) / *e_1] \wedge e_2 != NULL >$ if $*e_1 \in Sub^+(w) \cup \{w\}$
		$< w, es \wedge e_0 != e_1 >, < w[*(e_2...\#f) / *e_0], es[*(e_2...\#f) / *e_0] \wedge e_2 != NULL \wedge e_0 == e_1 >$ if $\exists *e_0 \in Sub^+(w) \cup \{w\}$ s.t. MayAlias($*e_1$, $*e_0$)
	$*e_1 = a_2$	$< \eta, es[a_2 / *e_1] >$ if $w =_s *e_1$
		$< w[a_2 / *e_1], es[a_2 / *e_1] >$ if $*e_1 \in Sub^+(w)$
	$*e_1 = t_2$	$<\varepsilon, es[t_2 / *e_1] >$ if $w =_s *e_1$
		$< w[t_2 / *e_1], es[t_2 / *e_1] >$ if $*e_1 \in Sub^+(w)$
	$*e_1$= NULL	$<\varepsilon, es[NULL/ *e_1] >$ if $*e_1 \in Sub^+(w)$
Branch condition	$*e_1$== NULL	$< \varepsilon, es[NULL / *e_1] >$ if $w =_s *e_1$

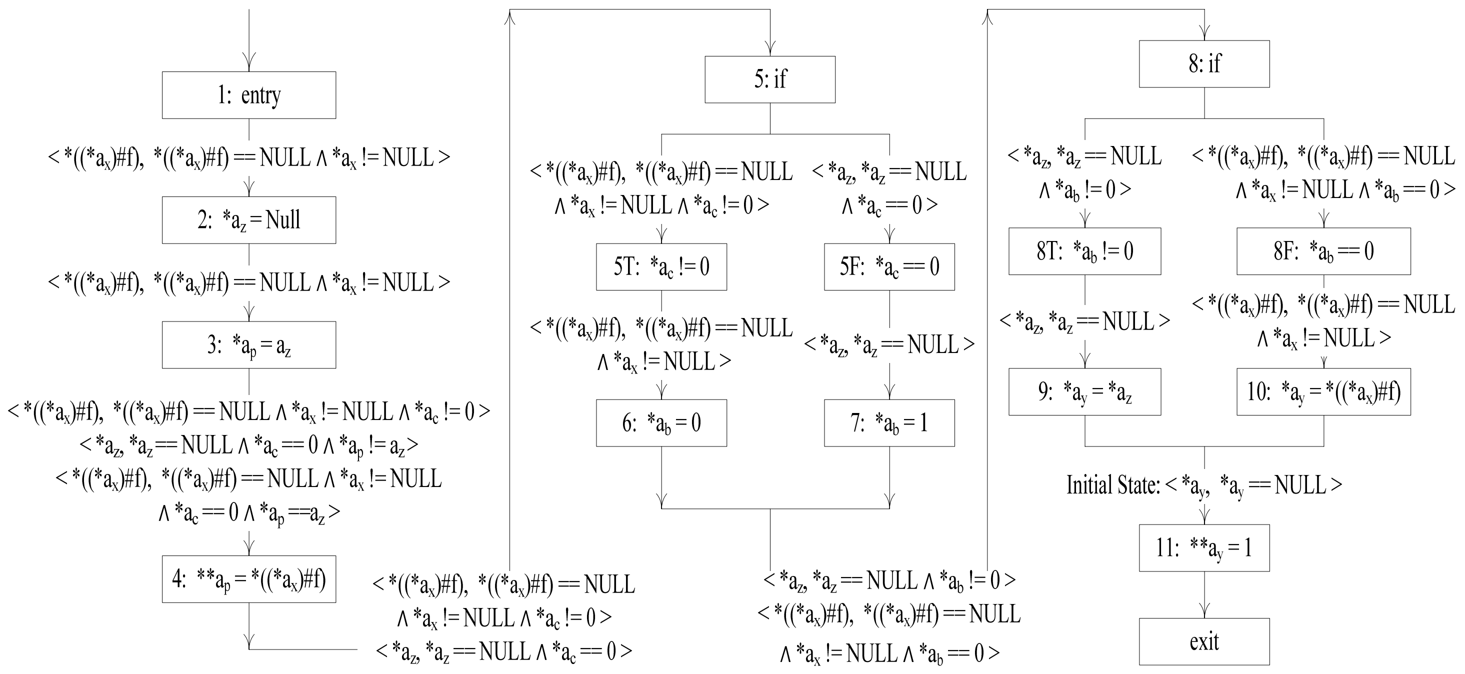
1: entry
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
2: *a_z = Null
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
3: *a_p = a_z
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_c != 0 >
< *a_z, *a_z == NULL ∧ *a_c == 0 ∧ *a_p != a_z >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_c == 0 ∧ *a_p == a_z >
4: **a_p = *((*a_x)#f)
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_c != 0 >
< *a_z, *a_z == NULL ∧ *a_c == 0 >
5: if
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_c != 0 >
< *a_z, *a_z == NULL ∧ *a_c == 0 >
5T: *a_c != 0
5F: *a_c == 0
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
< *a_z, *a_z == NULL >
6: *a_b = 0
7: *a_b = 1
< *a_z, *a_z == NULL ∧ *a_b != 0 >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_b == 0 >
8: if
< *a_z, *a_z == NULL ∧ *a_b != 0 >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_b == 0 >
8T: *a_b != 0
8F: *a_b == 0
< *a_z, *a_z == NULL >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
9: *a_y = *a_z
10: *a_y = *((*a_x)#f)
Initial State: < *a_y, *a_y == NULL >
11: **a_y = 1
exit

Figure 3. CFG built for the example code shown in Figure 1 and Symbolic states propagated backwards along the CFG to check whether there exists a null-dereference bug at label 11.

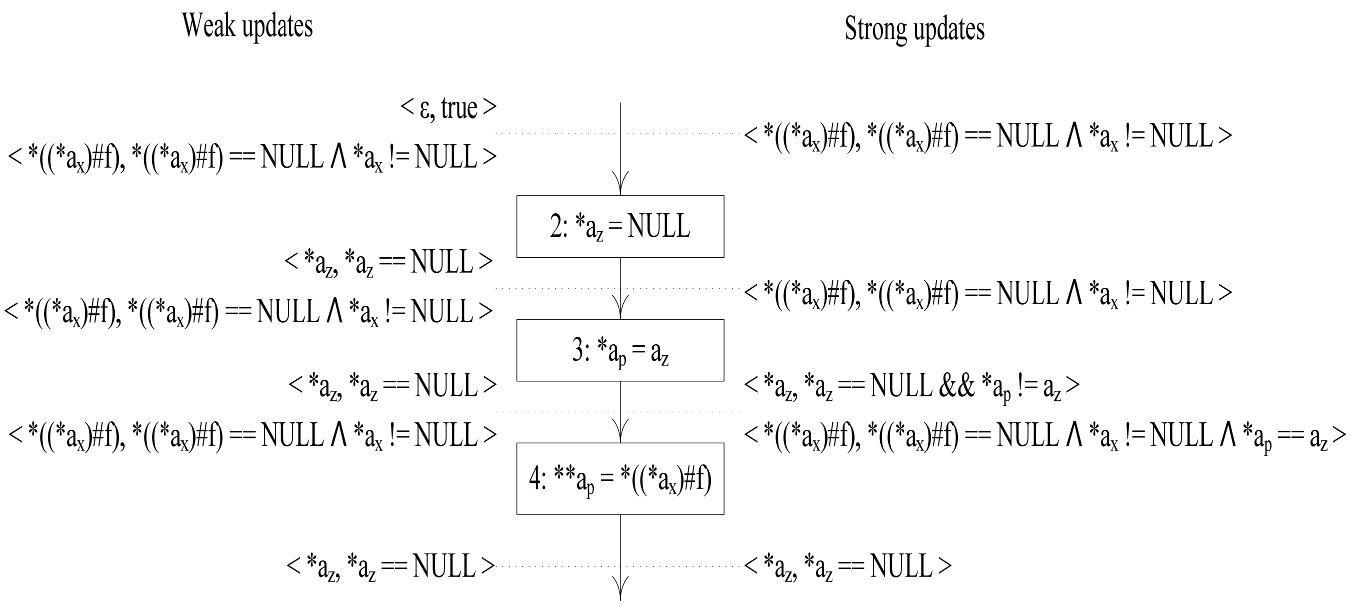
Weak updates
Strong updates
< ε, true >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
2: *a_z = NULL
< *a_z, *a_z == NULL >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
3: *a_p = a_z
< *a_z, *a_z == NULL >
< *a_z, *a_z == NULL && *a_p != a_z >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL >
< *((*a_x)#f), *((*a_x)#f) == NULL ∧ *a_x != NULL ∧ *a_p == a_z >
4: **a_p = *((*a_x)#f)
< *a_z, *a_z == NULL >
< *a_z, *a_z == NULL >

Figure 4. A comparison between weak updates and strong updates.

the canonical representation, and specially two extra nodes (5T and 5F) are added as successors of if-else statement at label 5 to explicitly represent its true and false branch conditions, similar measure is also taken on the if-else statement at label 8. In order to check the dereference operation at label 11 is a bug or not, our approach propagates a series of symbolic states backwards along the CFG. The initial state is $<*a_y,\ *a_y == NULL>$ in which the current witness is the dereference expression $*a_y$ and the current execution state is the root predicate $*a_y == NULL$.

Each statement potentially transforms the symbolic state that comes from its successor. The transfer functions for the two components of a symbolic state are

$$updateWitness :: Statement \times Witness \times Execution\ state \\ \rightarrow \rho(Witness \times Execution\ state)$$

and $$updateEState :: Statement \times Execution\ state \\ \rightarrow \rho(Execution\ state)$$

which are respectively explained in section III.A and section III.B. Then the function

$$update :: Statement \times Witness \times Execution\ state \\ \rightarrow \rho(Witness \times Execution\ state)$$

defines the overall backward effect of an individual statement on a state tuple:

$$update(st,\ <w,\ es>) = \\ \{\ <w',\ es'>\ \mid\ <w',\ es'> \in updateWitness(st, <w,\ es>) \\ \vee\ w = w',\ es' \in updateEState(es)\ \}$$

The objective of the transformation at an individual statement is to accept a post-state $\varphi = <w, es>$, and return a set of pre-states $\varphi' = <w', es'>$ that are over-approximation of the weakest preconditions [6] of φ concerning the statement.

A symbolic state is propagated backwards along the control flow graph until one of the following conditions encountered: a) the witness is updated to η; b) the execution state evaluates false; c) the witness is updated to ε and the execution state evaluates to true. In the first two cases, the analysis abandons traversal, since either there is no NULL value that flows into the initial dereference point, or the path that the analysis currently traces is infeasible. In the third case, a null-dereference bug is identified and a bug-found message will be emitted.

What should be emphasized is that: a symbolic state is regarded as invalid if it satisfies either of the first two cases; otherwise, a symbolic state is regarded as valid. Moreover, a symbolic state is called as a bug-found state whenever it satisfies the third case.

A. Witness Transformations and Strong Updates

In this subsection, we begin to discuss the witness transformations as well as some corresponding alterations in the execution state.

The idea behind the transformations is based on the observation that at every program point, there is only a single witness l-value that currently holds the value of interest, such that subsequent statements will copy the value of interest from this l-value to the dereference point being detected. Therefore, when proceeding backward analysis, our approach re-traces the chain of assignment statements that cause the value of interest to be transferred among l-values, and then accordingly updates the witness along with some corresponding alterations in the execution state.

In some simple cases, such transformations can be easily regarded as performing substitutions going backwards.

EXAMPLE 2. As shown in Figure 3, we start the null-dereference analysis from the statement at label 11 to identify whether there exists a bug or not. $*a_y$ is taken as the initial witness. At label 10, we find that the value of $*a_y$ is copied from $*((*a_x)\#f)$, thus we substitute $*((*a_x)\#f)$ for the witness accordingly and meanwhile replace every occurrence of $*a_y$ in the execution state with $*((*a_x)\#f)$. In addition, an predicate $*a_y\ != NULL$ is introduced into the execution state to ensure that the value of the witness can be transferred to the dereference point at label 11 without any exception. Similar transformation also occurs at label 9.

Besides the cases above, aliasing is something that one cannot ignore when doing the null-dereference analysis; failure to take into account aliasing can limit the usefulness of an approach. Furthermore, strong updates are required for precision but difficult to perform. Even precise pre-computed may-alias and must-alias information may not enable strong updates enough, since at a given program point two l-values may be aliased under some paths and not aliased under other paths.

Owing to the canonical representation, we can explicitly model the address of a single memory location that an l-value refers to. Moreover, for two l-value $*e_1$ and $*e_2$, we utilize the predicate $e_1 == e_2$ and $e_1\ != e_2$ to respectively represent two incompatible condition: the former under which $*e_1$ and $*e_2$ are aliased; and the latter under which they are not. Such predicates are the so-called aliasing predicates mentioned in previous section. With the help of them, our approach achieves to perform strong updates in presence of aliasing. Consider an assignment $*e_l = *e_r$ and suppose the witness $*e_w$ is not syntactically equal to but may be aliased with $*e_l$. We (i) hypothesize that $*e_w$ and $*e_l$ are aliased, then substitutes $*e_r$ for the witness and add an aliasing predicate $e_w == e_l$ into the execution state (in which every occurrence of $*e_w$ is replaced with $*e_r$) through logical AND operation; and

(1)	$Sub^+(c)$	$::= \emptyset$	$c \in Const$
(2)	$Sub^+(t)$	$::= \emptyset$	$t \in Alloc$
(3)	$Sub^+(a)$	$::= \emptyset$	$a \in Addr$
(4)	$Sub^+(*e)$	$::= \{e\} \cup Sub^+(e)$	$e \in Expr$
(5)	$Sub^+(e\#f)$	$::= \{e\} \cup Sub^+(e)$	$e \in Expr$
(6)	$Sub^+(es)$	$::= \bigcup_{e \in Term(es)} Sub^+(e)$	$es \in Formula$

Figure 5. Definition of sub-expressions

TABLE II.
EXECUTION STATE TRANSFORMATIONS AT SOME STATEMENTS

Statement		Execution state transformation
Assignment	$*e_1 = e_2$	$es[e_2 / *e_1]$ if $*e_1 \in Sub^+(es) \cup Term(es)$
		$es \wedge (e_0 \mathrel{!=} e_1), es[e_2 / *e_0] \wedge (e_0 == e_1)$ if $\exists *e_0 \in Sub^+(es) \cup Term(es)$ s.t. $MayAlias(*e_1, *e_0)$
Branch condition	e_1 op e_2	$es \wedge (e_1$ op $e_2)$

also (ii) hypothesize that $*e_w$ and $*e_l$ are not aliased, then keep $*e_w$ as the witness and add an aliasing predicate $e_w \mathrel{!=} e_l$ into the execution state (which has no other alterations) through logical AND operation too. Either of the hypotheses can be confirmed or contradicted by the subsequent analysis when it arrives at the statement that validates or invalidates the corresponding aliasing predicate.

EXAMPLE 3. Let's explore the symbolic states presented in Figure 3. $*a_z$ is one witness following the statement at label 4. We don't have a syntactic match between $**a_p$ and $*a_z$, but a query to pre-computed may-alias information shows that they may be aliased. To handle this situation soundly, we take measure as talked above to obtain two updated symbolic states: one in which an aliasing predicate $a_z == *a_p$ is introduced and the witness is accordingly updated to $*((*a_x)\#f)$; one in which an opposite aliasing predicate $a_z \mathrel{!=} *a_p$ is introduced and the witness is kept as $*a_z$. Subsequently at label 3, we abandon the second symbolic state, for its corresponding aliasing predicate gets invalidated due to the find that the value of $*a_p$ comes from a_z.

Moreover, we give a comparison between strong updates and weak updates in Figure 4 (the code shown in Figure 4 is a fragment of the one shown in Figure 1), and the weak updates presented here are very similar as the approaches adopted by PSE [21] and Xylem [7].The main difference between the two is whether to apply aliasing predicates during the analysis. The strong updates are able to exclude the spurious states generated at label 4. In contrast, the weak updates cannot, hence yield a false positive at label 2.

What has been mentioned above is all covered in the function *updateWitness*. It accepts a statement *st* and a witness *w* as well as an execution state *es*, and computes a set of l-values that are copied by *st* to *w*, meanwhile there are some corresponding alterations in *es*. Table I defines this function for some statements; and it is identity for others. The notations are as follow: the term e_0, e_1 and e_2 refer to arbitrary expressions in the canonical representation; a_2 denotes a primitive address, and t_2 represents a allocative address; for a witness *w* and expression e_1 and e_2, $w[e_1/e_2]$ denotes *w* in which every occurrence of *e2* is replaced by e_1, and $es[e_1/e_2]$ has a similar meaning; $Sub^+(e)$, which is defined in Figure 5, represents a set that includes all the sub-expressions of expression e, and one can easily deduce that any element in $Sub^+(e)$ maps a prefix of the C expression that *e* corresponds to; $*(e\ldots\#f)$ is used to represent either $*e$ or $*(e\#f)$. The transformations at assignments are not repeated again for the idea behind them has been discussed above. Given branch condition $*e_1 == NULL$, it is regarded as equivalent to assignment $*e_1 = NULL$ in case that it is a null check for the witness, then the witness is updated according to what occurs at assignment $*e_1 = NULL$.

B. Execution State Transformations and Grouping Symbolic States

In this subsection, we first talk about the function *updateEState*, which is used to perform the transformations for the execution state. It accepts a statement and an execution state, and then returns a set of possible execution states just preceding the statement.

Table II defines this function for assignments and branch conditions; and it is identity for other statements. Given an execution state *es* which is of the form a formula, the notation *Term(es)* means a set that includes all the terms in *es*; and as shown in Figure 5 the sub-expressions of each term in *es* together constitute $Sub^+(es)$.

Similar as what changes happen to a witness, every term in an execution state is potentially updated due to the effect of an assignment statement. And in respect to the transformation that occurs at a branch condition such as $e_1 == e_2$, the updated execution state is of the form a conjunct which contains the branch condition and the predicates in the incoming state. In this way, our approach keeps correlation among different branches and achieves path-sensitive analysis.

EXAMPLE 4. As shown in Figure 3, the symbolic state coming from the true branch of the if-else statement at label 8 cannot be propagated to the true branch of the if-else statement at label 5, for the branch condition contained in its execution state is invalidated by the statement at label 6.

(1)	$e_1 == e_1$	$\rightarrow$	true
(2)	$t_1 \mathrel{!=} t_2$	$\rightarrow$	true
(3)	$a_1 \mathrel{!=} a_2$	$\rightarrow$	true
(4)	$a_1 == NULL$	$\rightarrow$	false
(5)	$t_1 == NULL$	$\rightarrow$	true
(6)	$e_1\#p \mathrel{!=} e_2\#q$	$\rightarrow$	true
(7)	$e_1 == e_2 \wedge e_1 \mathrel{!=} e_2$	$\rightarrow$	false
(8)	$e_1 == c_1 \wedge e_1 > c_2$	$\rightarrow$	$e_1 = c_1$ if $c_1 > c_2$
(9)	$e_1 == c_1 \wedge e_1 > c_2$	$\rightarrow$	false if $c_1 \leq c_2$
(10)	$e_1 == e_2 \wedge e_1 == NULL \wedge e_2 == NULL$	$\rightarrow$	false

Figure 6. Simplification rules

Besides what has been mentioned above, in order to avoid an exponential blow-up of paths, our approach takes a measure that inspired by the forward analysis in ESP [4] to group the symbolic states. The function Merge, which is defined as follow,

$$Merge(ss) = \{ <w, \ \vee_{s\in ss[d]} es(s)> \ \mid w \in LVs \ \wedge \ ss[w] \neq \emptyset \}$$

$$where \ ss[d] = \{ w \mid s \in ss \ \wedge \ w = witness(s) \}$$

is used to accomplish the process. The term *LVs* means a set of l-value expressions in a procedure; *witness(s)* and *es(s)* are separately used to obtain the witness and execution state of a symbolic state *s*. This function accepts a set of symbolic states, and then groups the elements of the set based on the witness. All the execution states in one group are merged together to construct a formula by logical OR operation. For example, after the merging of $< *e, e == NULL \wedge *a_c == 0 >$ and $< *e, e == NULL \wedge *a_c != 0 >$, the symbolic state $< *e, e == NULL >$ is obtained.

C. Simplification Rules

After the above transformations, each predicate in an execution state can be evaluated to true, false or unknown according to some custom-defined rules; and furthermore, the formula can be validated, invalidate or simplified.

Figure 6 shows a few of sampling rules used in our simplifier. The notations e_1 and e_2 refer to different arbitrary expressions in the canonical representation; and either a_1 or a_2 means a primitive address that denotes the address of a variable; t_1 and t_2 are distinct allocative addresses for different memory allocation sites; as for the symbols *f*, *p* and *q*, they are used to describe distinct offsets away from some base locations.

Rule 1 is straightforward.

Rule 2-3 hold the observation that no matter a primitive address or an allocation address is unique, since the memory locations allocated at different sites are disjoint and different variable are stored in distinct memory locations. Furthermore, Rule 4 denotes that the primitive address for any variable is never equal to NULL. But as for an allocative address, because of the potential failure of memory allocation, it is considered to be equal to NULL for conservation. Rule 5 keeps this idea and it is reasonable, for many develop standards (e.g. MISRA C [20]) demand there must be a null check after each dynamic allocation in C program.

Rule 6 shows that for two offset expressions, as long as their offsets are different, no matter their bases are equal or not, they are distinct.

Rule 7 is based on the fact that a conjunct with a pair of conflicting predicates is invalid and evaluated to false.

Rule 8 demonstrates a kind of simplification method. That is for a conjunct with a predicate in the form of $e_1 == c_1$, it replaces the other occurrences of e_1 in the conjunct with c_1. Considering the conjunct shown in Rule 8, after the replacement, if the constant c_1 is really greater than the constant c_2, then the conjunct is reduced to $e_1 == c_1$; otherwise it is evaluated to false, for its sub predicate $c_1 > c_2$ is invalid.

```
Global
CS:            call stack;
∑(p, φ):       summary information for procedure p
that maps state φ to associated sets of states;
Γ(p):          container for procedure p that includes
states propagated to entry of p.

Function       AnalyzeProcedure
Input          st: statement to start analysis from;
               Φ: state preceding st;
               p: procedure being analyzed.
Output         result: set of states propagated to entry of p
Declare        Worklist: list of CFG nodes;
               ss, ss': set of states;
               Out(n): obtains outgoing edges of node n;
               Info(e): gets states stored on edge e;
               Exit(p): gains exit node of p;
               Add(n, φ): adds node n to Worklist and maps
state φ onto incoming edges of n.
Begin
1.             Worklist = Ø
2.             result = Ø
3.             st → n
4.             Add(n, φ)
5.             while Worklist ≠ Ø do
6.               remove a node n from Worklist
7.               ss = Ø
8.               st ← n
9.               for each edge e ∈ Out(n) do
10.                ss = Merge(Info(e) ∪ ss)
11.              for each state s ∈ ss do
12.                if st calls a procedure m then
13.                  if ∑(p, s) is defined then
14.                    ss' = ∑(p, s)
15.                  else
16.                    push(CS, p)
17.                    ss' = AnalyzeProcedure(m, Exit(p), s)
18.                    pop(CS)
19.                else if st is ENTRY then
20.                  result =result ∪ ss
21.                  break
22.                else
23.                  ss' = Update(st, s)
24.                for each state s' ∈ ss' do
25.                  s" = Simplify(s')
26.                  if s" is valid then
27.                    if s" represents a bug-found state then
28.                      Omit a bug message
29.                    else
30.                      Add(n, s")
31.            if CS ≠ Ø then
32.              ∑(p, φ) = result
33.              result = Ø
34.            return result
End

Function  AnalyzeNullDereference
Begin
1.             set CS to empty
2.             set all entities in ∑, Γ to empty
3.             for each procedure p in reverse-topologic order do
4.               pre = Ø
5.               for each statement st in p do
6.                 if st dereferences a variable v then
7.                   pre = pre ∪ AnalyzeProcedure(m, st, *a_v==NULL)
8.                 if st calls a procedure q with Γ[p] ≠ Ø then
9.                   for each state s ∈ Γ(q) do
10.                    pre = pre ∪ AnalyzeProcedure(q, st, s)
11.              if pre ≠ Ø then
12.                Γ(q) =Γ(q) ∪ pre
End
```

Figure 7. Algorithm for inter-procedural analysis

Rule 9 takes a similar method of Rule 8 to do the simplification.

What should be pay attention is that our approach repeatedly applies these simplification rules on a formula until a fix pointer is reached.

D. Inter-procedural Algorithm and Optimizations

The algorithm for inter-procedural null-dereference analysis is presented in Figure 7. Our approach achieves to perform inter-procedural analysis in a context-sensitive manner through the way of partial transfer functions, or summary table [9, 10], with a few modifications to adapt backward traversal. The reason why partial transfer functions are used is that: an effective and classical way to do inter-procedural analysis is by means of transfer functions to summarize the behavior of procedures for all possible inputs [10]; however, for null-dereference analysis that relates to pointers, enumerating all the possible aliasing combinations for every input to form complete transfer functions is impractical, moreover, most of those combinations typically never occur in the program under test; thus we compute partial transfer functions to summarize the procedures for relevant inputs that occur in the program.

The main idea behind the algorithm is as follow: suppose we are processing procedure *foo* and we encounter a call to another procedure *bar*. We would like to apply a transfer function to map the symbolic state s at the exit of bar to associated states that would result at the entry of bar after propagating s through the procedure (and its transitive callees). However, since the body of *bar* contains multiple statements, the transfer function must be generated dynamically by analyzing *bar*. This is done be maintaining and updating a summary table ∑ for *bar*. When a call to *bar* is encountered in *foo* with symbolic state *s*, the summary table for *bar* is consulted. If no corresponding summary information exits, the algorithm descends into *bar* to analyze it. A call stack *CS* is used to ensure context-sensitive processing of called procedure. After returning from *bar*, the algorithm saves the summary information to reuse in subsequent analysis.

```
L1:  #include <stdlib.h>
L2:  void f1() {
L3:     int *x, *y, **p;
L4:     int b = 0;
L5:     x = NULL;
L6:     p = &x;
L7:     *p = &b;                        //safe
L8:     y = x;
L9:     *y = 1;                         //false positive
L10: }

L11: void f2(int flag, int *p, int *q) {
L12:    if(flag ==0) {
L13:       *p = 1;
L14:    }
L15:    else {
L16:       *q = 1;
L17:    }
L18: }

L19: void f3() {
L20:    int c = 3;
L21:    f2(1, NULL, &c);                //false-positive
L22:    f2(0, NULL, &c);                //null-dereference bug
L23: }
```

Figure 8. Experimental sample code.

On reaching the entry of *foo*, the algorithm collects the symbolic states propagated here. If *foo* is not being analyzed in a specific context (i.e. the call stack is empty), the algorithm continues to propagate these states through all the predecessors of *foo* (and its transitive callers) until they can be validated or invalidated. This process is accomplished with the help of a container Γ.

Because of the trade-off among efficiency, cost and accuracy, we take some optimization measures to determine the extent to which a symbolic state is explored.

First, we bound the number of predicates in an execution state. To deal with this, we associate an age with every predicate, which is the number of statements it has been propagated through; we have a threshold k_1, and drop (i.e. reduce to true) a predicate whenever its age increase beyond k_1. The idea behind dropping old predicates is such an observation that branch correlations in paths typically occur between branches that are near each other in the code. What should be emphasized is that the root and aliasing predicates are never dropped.

Second, we restrict the length of l-value expressions in a symbolic state. Another threshold k_2 is used here. If the length of an l-value expression exceeds k_2, we switch that expression to an abstract location. The abstract location representation is an identifier taken from a finite partition of all memory locations obtained from a pre-computed flow-insensitive points-to analysis. This representation is less precise, since a single abstract location may represent a set of memory locations. We use abstract locations to ensure termination of the analysis (e.g. on programs with recursive data structures).

Beside the above two, we group the symbolic states propagated to the same point in the program. This process has been discussed in section III.B and will be not repeated again here.

IV. Experimental Evaluation

A. Experimental Setup

We have implemented a prototype of our approach as an extension of Defect Test System (DTS), which is a general automatic bug-detection framework for C programs, especially GCC programs. In the next subsection, to estimate the effectiveness and the accuracy of our approach, we conduct two experiments to compare it with the original approach that DTS used to perform null-dereference detection. That approach depends on a forward interval analysis and has been verified as a reliable approach [11]. All the two experiments are run on a dual-processor 1.80GHZ Pentium E2160 with a 2GB

TABLE III.
RESULT OF EXPERIMENTATION 1

LOC	BP	DTS-F	DTS-B
L7	N		
L9	N	R	
L21	N	R	
L22	Y	R	R

physical memory, and have been measured with enough repetitions to avoid mistakes. Besides, for simplicity, the notation DTS-F and DTS-B are used to respectively indicate the approach that DTS ever used and the approach that we propose in this paper.

B. *Experimentation Analysis and Discussion*

EXPERIMENTAION 1. We first select some sample codes inserted with null-dereference bugs manually as experimental object to prove the effectiveness of our approach in some extent.

There are four inspection points in the sample code that is demonstrated in Table III. The BP entries denote which one is confirmed as a genuine bug by artificial identification. As shown in Table III, DTS-F reports three bugs at *L9*, *L21* and *L22*, among which the first two are both false positives. Since DTS-F takes a conservative measure to deal with the aliasing situation, it believes that the value of y at *L9* may be NULL, therefore leading to a false positive. Due to the context-insensitive manner taken by DTS-F to do inter-procedural analysis, it cannot distinguish the distinct context conditions at different call sites, and then it asserts not only the invoking at *L22* but also the one at *L21* causes a null-dereference bug inside of procedure *f2*. By contrast, DTS-B finds the genuine bug at *L22* with no false positive, which owns to strong updates in presence of aliasing and inter-procedural analysis in a context-sensitive manner.

EXPERIMENTAION 2. In this part, to investigate the capability of DTS-B that detects null-dereference bugs in practice, we apply both DTS-B and DTS-F on 5 open source benchmarks. We believe all the benchmarks to be challenging and interesting, since they all contain many complex structures.

The experimental result is presented in Table IV, in which: the *LINE(s)* entries indicate the total lines of source code; the term *REP* and *DEF* respectively denote the number of bugs reported by an approach and the corresponding number of genuine bugs identified by manual confirmation; the *FPR* entries represent the false positive rate of an approach, whose computational formula is *FPR=(REP-DEF)/REP*100%* ; and the *FNR* entries show the false negative rate of an approach, whose computational formula is

FNR(DTS-F)=

*OTHER(DTS-B)/(DEF(DTS-F)+DEF(DTS-B)-SAME)*100%*

and *FNR(DTS-B)=*

*OTHER(DTS-F)/(DEF(DTS-F)+DEF(DTS-B)-SAME)*100%* .

According to the statistics, there are *100725* lines of source code together in all the benchmarks. *DTS-F* reports *377* bugs with *321* bugs identified, whereas DTS-B finds *405* bugs with *365* bugs confirmed. In contrast, *DTS-B* detects 1*3.71%* more bugs (*365 → 321*); and its FPR drops by 4.98%, meanwhile its FNR decreases by *9.26%*. All those imply that *DTS-B* can improve the accuracy of detection. Some details are given below.

No surprising, both the approaches find some common bugs. For instance, such kind of bug "*p = malloc(); *p = ...;*", which means that there exists no null check between a memory allocation and its relevant dereference operation, is frequently detected. Moreover, DTS-B eliminates some false positives and also finds some fresh bugs unknown before. Considering the code fragment presented in Figure 9, at line 99 in file *antiword-0.37/worddos.c*, DTS-F reports a null-dereference bug that occurs inside the callee *vGetPropertyInfo*, for its second actual parameter is NULL. But in fact, this bug can happen only when the second parameter is NULL and meanwhile the last parameter is equal to 7 or 8. Owing to context-sensitive inter-procedural analysis, DTS-B eliminates this false positive. As for the dereference operation at line 384 in Figure 10, DTS-B excludes its possibility as a null-dereference bug, since the false branch condition of if statement at line 379 ensures that the value of *pAnchor* cannot be NULL. What is shown in Figure 11 is a genuine bug found by DTS-B, whereas DTS-F neglects it falsely. That is a NULL value held by *psys* may flow into the statement at 682 through the true branch of if statement at 665.

We also analyze the false positives introduced by DTS-B and have found some typical reasons. Specific structures applied in the test code give rise to some false positives. For instance, in file */barcode-0.98/code128.c*, there exists an array variable *codeset* consisting of over

```
File: antiword-0.37/worddos.c
In caller procedure iInitDocumentDOS at line 70
99: vGetPropertyInfo(pFile, NULL,
              NULL, 0, NULL, 0);      //false positive

File: antiword-0.37/perperties.c
In callee vGetPropertyInfo at line 17
17: void
18: vGetPropertyInfo(FILE *pFile, const pps_info_type *pPPS,
          const ULONG *aulBBD, size_t tBBDLen,
          const ULONG *aulSBD, size_t tSBDLen,
          const ULONG *aucHeader, int iWordVersion)
22: {
        ......
37:     switch (iWordVersion) {
        ......
83:       case 6:
84:       case 7:
85:            vGet6Stylesheet(pFile, pPPS->tWordDocument.ulSB,
86:                   aulBBD, tBBDLen, aucHeader);
        ......
```

Figure 9. One false positive eliminated by DTS-B

100 members, which causes several analyses failed. Some

TABLE IV.
RESULT OF EXPERIMENTATION 2

PROJECT	LINE(S)	DTS-F						DTS-B					
		REP	DEF	SAME	OTHER	FPR	FNR	REP	DEF	SAME	OTHER	FPR	FNR
antiword-0.37	20213	49	43	39	4	12.24%	0	39	39	39	0	0	8.16%
barcode-0.98	3409	6	6	5	1	0	27.27%	10	8	5	3	0.2	9.09%
spell-1.0	1991	39	21	21	0	46.15%	7.84%	33	25	21	4	24.24%	0
sphinxbase-0.3	22517	110	97	94	3	11.82%	15.86%	129	117	94	23	9.30%	2.07%
uucp-1.07	52595	173	154	148	6	10.98%	12.79%	194	176	148	28	9.28%	2.74%
total	100725	377	321	307	14	14.85%	12.21%	405	365	307	58	9.88%	2.95%

other false positives are mainly due to optimization measures taken by DTS-B viz. dropping aged predicates and abstracting overlength l-values. Those usually happen when recursive data structures are encountered. Given the efficiency of our approach proposed in this paper, these accuracy losses can be acceptable.

In summary, the experimental results demonstrate the effectiveness of the approach proposed in this paper, and show that it is scalable to large real programs with reasonable accuracy.

V. RELATED WORK

Many approaches and tools [1, 2, 5, 7, 13, 24] have been developed for analyses for null dereference and similar safety properties. From the vast literature covering this space, we briefly review some of the relevant related work.

Xylem [7] is the most closely related approach to ours, though it targets null-dereference analysis of Java program. Our approach has several attributes that are inspired by Xylem: a backward dataflow analysis from each dereference, predicates as dataflow facts, custom-defined simplification rules for predicates rather than a constraint solver. Compared to Xylem, our technical innovation is in terms of how we perform strong updates instead of weak updates in presence of aliasing; strong updates are required for better precision in null-dereference analysis.

Salsa [3], also proposed by Xylem's authors, is an approach that aims at sound null-dereference verification. It is based on abstract interpretation and gradually expands the inter-procedural scope of analysis to establish the safety of a dereference. The goal of Salsa is to show the absence of bugs. But it may often report many spurious warnings (or false positives). By contrast, our approach focuses on bug detection to identify as many bugs as possible and it emphasizes not on reporting all potential bugs, but on reducing false positives. The two kinds of approaches represent different trade-offs and can be complementary.

FindBugs [14] is a widely used tool for Java that has paid particular attention to finding null dereference bugs [15]. FindBugs pattern-matches on constructs that are common sources of certain error classed and performs some data-flow computation. As our approach is target for C programs, it is not possible to do a direct comparison. Nevertheless, it is clear that FindBugs would not find the many path-sensitive, inter-procedural, and aliasing-dependent bugs that our approach uncovers.

Similar as our approach, PSE [21] performs a backward symbolic analysis with the goal of tracing back null-dereference bugs and disprove such bugs. But PSE does not represent the entire path condition, and sometimes falls back to abstract representations of the heap.

Strom and Yellin [5] define a partially path-sensitive backward dataflow analysis for checking typestate properties, specifically uninitialized variables. By comparison, our approach is able to track a value backward through pointer-based data structures and handle memory aliasing. And our approach prunes out infeasible paths through evaluation of predicates.

Prefix [1] can detect possible null-dereference bugs in C and C++ programs by symbolic simulation. Like our approach, Prefix uses procedure summaries for scalability and is path-sensitive. However, Prefix explicitly explores paths one at a time, which is expensive for procedures with many paths. Heuristics limit the search to a small set of "interesting" paths. In contrast, our approach implicitly represents all paths using predicate constraints and path exploration is as part of predicate evaluation.

Xie et.al. [8] present similar approaches for detecting a broad class of memory errors. Their approaches feature a bottom-up analysis of procedures to compute summaries, and a forward path-sensitive analysis within each

```
File: antiword-0.37/stylelist.c
const style_block_type *
pGetNextStyleInfoListItem(const styple_block *pCurr)
{
    const style_mem_type *pRecord;
    size_t tOffset;

    if(pCurr == NULL) {
        if(pAnchor == NULL) {
            /* There are no records */
            return NULL;
        }
        /* The first record is the only one without
         a predecessor */
        return &pAnchor->tInfo;     //false positive
    }
```

Figure 10. Another false positive eliminated by DTS-B

```
File: uucp-1.07/cu.c
234: int
235: main (argc, argv)
236:        int argc;
237:        char **argv;
238: {
              ......
665:      if(qsys == NULL)
666:      {
667:          const char *zrem;
668:
669:          if(flooped)
670:              zrem = "remaining ";
671:          else
672:              zrem = "";
673:          if(sinfo.fmatched)
674:              ulog(LOG_FATAL, "%s: ALL %smatching ports in use",
675:                    zsystem, zrem);
676:          else
677:              ulog(LOG_FATAL, "%s: No %smatching ports",
                        zsystem, zrem);
678:      }
679:
680:      iusebaud = qsys->uuconf_ibaud;          //null-dereference bug
```

Figure 11. Null-dereference bug found by DTS-B

procedure that prunes out infeasible paths. By contrast, our approach performs a backward analysis within each procedure to do the bug detection.

Some approaches attack null dereferences using user annotations on procedure parameters and local checking of each procedure body. LCLint [16] uses an unsound method to check the safety of dereferences of parameters annotated as may-be-null. More recent annotation-based systems are much closer to being sound [17, 18]. Current annotation languages, which mark a single parameter as possibly null or definitely not null, are not expressive enough to capture the more path-sensitive and inter-procedural relationships.

Model checking [2, 13, 22, 23] is used to check a number of safety properties, involving null dereferences. Saturn [2] and Calysto [13] generate constraints in propositional logic and use Boolean satisfiability solvers to discharge the constraints. Scalability of the techniques depend both on the scalability of the underlying SAT solvers as well as carefully tuned heuristics which keep the size of the constraints small. Notably, Saturn computes modular summaries to enable inter-procedural summary-based analysis. Similar measure is also taken by our approach. Whereas Calysto does not perform summary-based inter-procedural analysis, but makes use of inlined callee representations instead.

VI. Conclusion And Future Work

For identifying null-dereference bugs, we have presented an approach that is based on backward dataflow analysis. Owing to aliasing predicates, this approach can perform strong updates in presence of aliasing, thus eliminating many false positives. In addition, the other designed features, for instance context-sensitive inter-procedural analysis, have also contributed to improve precision. We have implemented this approach, and applied it on a set of 5 open source GCC benchmarks. The preliminary experimental results verify the effectiveness of this approach, and show that it is suitable for exploring large real programs with reasonable accuracy. Future work will be guided by the objective of continuing to improve the efficiency of the approach, while still remaining its precision. In particular, we would like to investigate techniques to deal better with references to arrays and recursive data structures. Besides, we would also like to investigate applications of our approach to check problems other that null-dereference analysis.

Acknowledgment

This research is supported by the National 863 Programs (Grant No. 2012AA011201) and National Natural Science Foundation of China (Grant No. 61202080).

References

[1] William R. Bush, Jonathan D. Pincus and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Softw., Pract. Exper.*, vol. 30, pp. 775–802, 2000.

[2] Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *POPL*, pp. 351-363, 2005.

[3] A. Loginov, M. G. Nanda, et al. Verifying Dereference Safety via Expanding-scope Analysis. In *ISSTA*, pp. 213–224, 2008.

[4] Manuvir Das, Sorin Lerner and Mark Seigle. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proc. of the 2002 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 57-68, 2002.

[5] Robert E. Strom and Daniel M. Yellin. Extending Typestate Checking Using Conditional Liveness Analysis. *IEEE Trans. on software Engineering*, vol. 19, pp. 478-485, 1993.

[6] E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, pp. 15-23, 1976.

[7] M. G. Nanda and S. Sinha. Accurate Interprocedural Null-dereference Analysis for Java. In In *Proc. of the 31th Intl. Conf. on Softw. Eng.*, pp. 133-143, 2009.

[8] Y. Xie, A. Chou and D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In *Proc. of the 9th European Softw. Eng. Conf. / 11th ACM SIGSOFT Intl. Symp. on Found. of Softw. Eng.*, pp. 327-336, 2003.

[9] T. Reps, S. Horwitz and M. Sagiv. Precise Interprocedural Data Flow Analysis via Graph Reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, 1995.

[10] R. Wilson and M. Lam. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proc. of the 1995 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1995.

[11] Z. H. Yang, Y. Z. Gong, et al. DTS: A Software Defects Testing System. *Proceeding of 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 269-270, 2008.

[12] B. Kernighan and D. Ritchie. The C Programming Language. Prentice-Hall, second edition, 1988.

[13] D. Babic and A. J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *Proc. of the 30th Intl. Conf. on Softw. Eng.*, pp. 211-220, 2008.
[14] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, vol. 39, pp. 92-106, 2004.
[15] 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*, pp. 13-19, 2005.
[16] D. Evans. Static Detection of Dynamic Memory Error. In *Proc. of the Conf. on Prog. Language Design and Implementation*, pp. 44-53, 1996.
[17] 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 Programming, Systems, Languages and Applications*, pp. 302-312, 2003.
[18] C. Flanagan, R. Leino, M. lillibridge, et al. Extended Static Checking for Java. In *Proc. of the conf. on Prog. Language Design and Implementation*, pp. 234-245, 2002.
[19] X. Ma, J. Wang, and W. Dong. Computing Must and May alias to Detect Null pointer Dereference. In *ISoLA*, pp. 252-261, 2008.
[20] MISRA C, http://www.misra-c.com/
[21] R. Manevich, M. Sridharan, S. Adams, M. Das and Z. Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. *SIGSOFT Software Engineering Notes*, vol. 29, pp. 63-72, 2004.
[22] Huiling Shi, Wenke Ma, Meihong Yang, Xinchang Zhang. A Case Study of Model Checking Retail System with SPIN. *Journal of Computer*, vol 7, No 10, pp. 2503-2510, 2012.
[23] Conghua Zhou, Bo Sun. Abstraction In Model Checking Real-Time Temporal Logic of Knowledge. *Journal of Computer*, vol 7, No 2, pp. 362-370, 2012.
[24] Mohammad Muztaba Fuad, Debzani Deb, Jinsuk Baek. Static Analysis, Code Transformation and Runtime Profiling for Self-healing. *Journal of Computer*, vol 8, No 5, pp. 1127-1135, 2013.

Qian Wang (1983 -) is currently a PhD candidate in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China.

His research interests include software testing and program static analysis.

Dahai Jin (1974 -) received his PhD in computer science from Armored Force Engineering Institute, Beijing, China, in 2006.

He currently serves as an associate professor in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His research interests include software testing and program static analysis.

Gongyun Zhan (1962 -) received his PhD in computer science from Institute of computing technology, Academia Sinica, Beijing, China, in 1992.

He currently serves as a professor in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His research interests include software testing, program static analysis and automatic test case generation.

Hongbo Zhou (1984 -) is currently a PhD candidate in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China.

His research interests include software testing and program static analysis.