2010

Detection of Static Flaws in Changesets

Daniel George De Graaf
Iowa State University

Detection of static flaws in changesets

by

Daniel De Graaf

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Tien N. Nguyen, Major Professor
Suraj Kothari
Zhao Zhang

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

When performing static analysis checks intended to find flaws in software, information on the software's history is often ignored in the identification of bugs. This history information can improve the identification of the introductions several types of bugs, and can be used to reduce the false-positive rate of simple analysis tools.

A method for static detection of null pointer dereferences by examining changes in a distributed development environment is presented. A static analysis used to detect null pointer dereferences is extended to highlight potentially dangerous state changes that point to the accidental introduction of such flaws.

CHAPTER 1. Introduction

In a large, actively evolving software project, software development can take place in several distinct directions simultaneously. Development branches are tested in isolation and are then merged to produce a result that contains a sum of all of the development effort. This style of development allows for multiple changes of wide scope to be attempted at the same time, and can help reduce the impact of implementing an incorrect or useless change to the time and resources put into its creation.

While development in this manner is very efficient, running static and dynamic analysis on the source code can become a difficult process. As the sensitivity of a tool increases, false positives become more prevalent, and it becomes impractical to use such tools to evaluate every proposed change. Current tools do not take into account the information available from the software development process – they consider the code frozen at a single point in time. As a result, the output of such a tool is of limited use in an everyday evaluation of a project due to continual effort spent eliminating false positives.

Null pointer dereferences are class of bugs that cause software crashes and have played a part in several security vulnerabilities. It is also very simple to check a pointer for null prior to dereferencing it, so the primary cost of eradicating this type of bug is finding the locations where a null dereference can happen. Static analysis can be used to find all locations in a given program where memory is dereferenced without a preceding check. However, in most programs, this type of check will have a very high false positive rate, because function arguments may be specified in an API to never be null, and the additional checks would only add redundant error checking. This property makes a simple static analysis of null pointer dereferences impractical.

By having a software tool emulate the behavior of humans reviewing a single patch, the false positives can be reduced to only those in the areas changed by the patch. By evaluating individual patches, each flaw is identified only for the patch that introduces it, rather than in every codebase that that contains the false positive. However, if this per-patch checking method is used in an entire development cycle, the same number of false positives must be examined as would be reported by simply running the static analysis tool prior to the release.

Static analysis tools currently only identify flaws or potential flaws in software. Previous work [Ayewah and Pugh, 2008] has been done in categorizing the importance of flaws depending on their type, location, or other information. However, when evaluating a patch that introduces a flaw, it is useful to have more detailed information about the state of the program when no flaws are detected. In this paper, a method for using additional information from the development history to improve the accuracy of a static analysis tool is described.

By examining memory accesses within a function, its parameters can be classified into states that describe the safety of memory accesses using the parameter. This state information has very low accuracy in detecting bugs, because it is common for function parameters to be dereferenced without an explicit null check. However, changes in the state caused by a given patch can be more useful: for example, the removal of a check for null prior to a dereference is a more reliable indication of a null dereference bug than the isolated occurrence of a dereference without a check.

In addition to reviewing individual commits, the use of distributed development has made it important to check for logical conflicts that arise from the merging of two development branches. By checking the state of a variable in the branches being merged, it is possible to detect when an unexpected change takes place, and flag the merge for review.

In this paper, a static analysis algorithm that checks for null pointer access and produces three output states – two success and one failure – is presented. Using this algorithm, a series of tests based on the state changes are presented for evaluating individual patches, patch series, and branch merges. A tool implementing these algorithms is presented and evaluated when run against versions of the Linux kernel.

CHAPTER 2. Static analysis

Potential dereferences of a null pointer can be detected using purely static analysis. Finding all sites where a pointer dereference can occur is trivial; the difficulty lies in evaluating the safety of the dereference. Perfect evaluation of this safety is impossible because it would involve solving the halting problem, but it is possible to classify dereferences into states that describe the safety of the dereference.

The algorithm described in this section attempts to find checks for null that are run prior to the dereference, and uses these conditions to mark dereferences on a checked branch as safe. Variables within the function are then marked as safe, unsafe, or not dereferenced based on this information.

2.1 Precondition Generation

The null pointer dereference problem can be solved by requiring that the validity of all pointers passed into a function be stated in a function precondition. This precondition can then be used both to determine if the function could dereference a null pointer, and to verify that calling functions do not violate this precondition.

While this technique is the best solution to the problem, most existing and new code does not include formal function preconditions. Even if developers could be convinced to include such preconditions on all new code (an unlikely event), there would still be a large base of legacy code with no such preconditions.

Generating preconditions from existing code is a difficult problem. Unconditionally correct prototypes are impossible to generate, as any algorithm that can perfectly determine the preconditions has solved the halting problem. Existing projects have addressed this problem in two ways: static analysis that attempts to identify code patterns that are incorrect, and dynamic analysis that attempts to determine ranges from observed values.

Both of these techniques will result in some incorrect conclusions being present. Dynamic analysis is prone to false negatives because it cannot draw conclusions about execution paths that are not tested. Only a few types of preconditions are detected in this manner that would not be detected by simply running into the problem during the test execution; this dynamic analysis is more useful in detecting things like arithmetic overflow.

In contrast to dynamic analysis, static analysis can produce both positive and negative assertions about the safety of a program. Some types of safety detection are widely used even in the presence of false positives – for example, many compilers will detect possible access to uninitialized local variables. If a project is developed using such a compiler, any false positives that are detected will likely be fixed by adapting the code as it is written, rather than by noting the error as a false positive.

Any false positives detected by a compiler or a static analysis tool that examines a static source tree will need to be evaluated each time the tool is run. Even if they are known to be false positives, it is often easier to "fix" the false positive by creating equivalent code that does not trigger the false positive than to continue ignoring it in future reviews. When this is not possible, the cost of running the tool will increase with time due to the need to repeatedly examine the same false positives.

2.2 Parameter Classification

In order to detect null pointer dereferences due to a function parameter that is null, function parameters will be classified by evaluating their use in the function. Each function parameter will be in one of the following three states:

1. Not dereferenced (NONE) – either not a pointer, unused in the current implementation, or only passed to other functions that are not marked as dereferencing
2. Dereferenced safely (SAFE) – the pointer is dereferenced at least once, but is always checked for null first.
3. Dereferenced unsafely (UNSAFE) – there exists a code path that will cause the pointer to be dereferenced, without first checking for null

As an example, the function in Figure 2.1 has one parameter in each state – `count` is in state NONE as it not a pointer, `code` is in state SAFE as it causes the function to abort if it is null, and `output` is in state UNSAFE because it is filled without any check for null.

In order to improve the analysis of dereferences that are in a different function from the null check, some level of inter-procedural analysis may be useful. This analysis would increase the time cost of the static analysis, but will increase the accuracy of the classification.

This classification method has a high false positive rate when used to analyse a single revision of an application. Many functions in C have implicit or explicit preconditions that the parameters are

```
int pop_counters(int count, record* list,
                 summary_t* output) {
  int i, min = INT_MAX, max = INT_MIN;
  int inversions = 0;
  if (count == 0 || list == NULL)
    return -EINVAL;
  for(i=0; i < count; i++) {
    if (i && list[i-1].id > list[i].id)
      inversions++;
    if (list[i].id < min)
      min = list[i].id;
    if (list[i].id > max)
      max = list[i].id;
  }
  output->num_seen++;
  if (max - min > output->best_range) {
    output->best_range = max - min;
    output->best_list = list;
    output->min = min;
    output->max = max;
  }
  return inversions;
}
```

Figure 2.1 Example function, version 1

```
ENTRYPOINT:
    seteq.32    %r2 <- %arg1, $0
    seteq.32    %r4 <- %arg2, $0
    or-bool.1   %r5 <- %r2, %r4
    phisrc.32   %phi4(min) <- $0x7fffffff
    ...
    phisrc.32   %phi15(inversions) <- $0
    br          %r5, .L001, .L002

.L001:
    phisrc.32   %phi1(return) <- $0xfffffff9
    br          .L003

.L002:
    setlt.32    %r9 <- %r49, %arg1
    br          %r9, .L004, .L005

    ...

.L008:
    add.32      %r13 <- %r49, $-1
    muls.32     %r14 <- %r13, $4
    add.32      %r15 <- %arg2, %r14
    load.32     %r16 <- 0[%r15]
    muls.32     %r19 <- %r49, $4
    add.32      %r20 <- %arg2, %r19
    load.32     %r21 <- 0[%r20]
    setgt.32    %r22 <- %r16, %r21
    br          %r22, .L009, .L010
```

Figure 2.2   Partial linearization of Figure 2.1

not null, and so flagging all of these functions as containing unsafe accesses will hide the true unsafe accesses in the noise.

When used as a purely static analysis tool, this method will have a high false-positive rate in most applications, because function parameters are commonly specified with a precondition that the pointer is not null. The analysis is regarded as a negative result. However, changes between the states of a function's parameter due to a patch will yield a more useful result.

2.3 Implementation

This algorithm for finding NULL memory accesses has been implemented on top of sparse [Christopher Li, 2010], which is a C parser developed for checking the Linux kernel. Sparse processes C source code into a linearized form consisting of 63 opcodes that represent basic C operations on an unlimited-register virtual

```
if (!x)
  return;
while (i < len) {
  x[i] = compute(i);
  if (x[i] & 0x1)
    odd++;
  else
    even++;
  i++;
}
```

$$\begin{tikzcd}
A \arrow[d] \arrow[dr] & \\
B \arrow[d] \arrow[dr] & C \arrow[ddd] \\
D \arrow[d] & E \arrow[dl] \\
F \arrow[uu, bend left] \arrow[dr] & \\
& G
\end{tikzcd}$$

Figure 2.3 Control flow graph of a simple loop

machine. These opcodes are grouped into blocks that are linked to one another based on the program's flow control statements, with each block having at least one inbound and one outbound link.

2.3.1 Predicate Generation

Each block of statements is evaluated to find variables that contain information about the nullity or non-nullity of other variables. For example, the first two statements of the example linearization are testing arg1 and arg2 against zero. This type of test produces two predicates: if `%r4` is zero, this implies `%arg2` is nonnull, and if `%r4` is nonzero, then `%arg2` is null. Only the first predicate can be propagated through the or operation, so if `%r5` is zero, it is known that `%arg2` is nonnull, but if `%r5` is nonzero then nothing can be determined about `%arg2`.

After single-block predicates have been generated, the sources of each statement block are examined to determine what predicates can be propagated through the block. If the source of a block is a branch (for example, `.L002`), then predicates depending on the condition of the branch can be evaluated into definite statements about if a variable is null or non-null. The branch decided by `%r5` will cause both `%r4` and `%arg2` to be marked as definitely non-null in the block `.L002` and all child blocks that do not have multiple parents.

2.3.2 Loop Unhooking

Loops cause problems with the simple propagation of predicates. If a block has multiple parents, then for a predicate to be propagated into the block, it must appear in all parents – otherwise, it may not be true in the parent, and conclusions made from the predicate would be incorrect. Consider the code snippet and corresponding control flow graph in figure 2.3. The condition tested in block A cannot be propagated into block B because block B has a second parent of F.

To resolve this, a loop-detection predicate is defined that indicates that any predicates valid in a certain block are also valid in this block. In a loop, this predicate will propagate down the body of the loop and mark the final block in the loop as a descendant of the initial block. Once a parent block has been identified as a descendant, it is no longer considered when restricting the propagation of predicates from parents, allowing the predicates from outside the loop to enter the loop. In the example, block F would contain 2 such blocks: one for A and B. It would not contain a predicate for D or E because these blocks cannot propagate into the block, since they are not present in both parents of F.

The propagation of predicates is repeated a fixed number of times to allow nested loops to be detected and to allow propagation of conditions through blocks whose parent-child orderings do not match the line orderings in the source. This block propagation method avoids the problem of considering all possible 2^n execution paths of a list of chained if statements by considering each block only once. This will fail to detect some edge conditions where all possible execution paths satisfy a check, but do not do so in a straightforward manner.

2.3.3 Memory Safety Checking

Once a list of predicates has been generated for all blocks, all memory accesses are examined to determine if the variables they dereference have been checked. For each memory access, the predicates for the variable are checked for any definite knowledge; if such a predicate exists, it is used to mark the variable. If the variable was initialized by adding an integer to a pointer, the original pointer is also marked.

Once this pass has been run, all variables will be in one of three states: not dereferenced in this function, dereferenced only when tested for non-null, or dereferenced without check. The state of the arguments of the function under test are then reported for use by the classification algorithms.

2.3.4 Inter-Procedural Analysis

The accuracy of the evaluation of memory safety can be improved if information about the use of parameters to function calls is included. If one function checks a pointer and then passes it into a function that uses the pointer, the check may be ignored if the first function does not otherwise dereference the pointer. If the check cannot be identified as protecting against a null dereference, flagging its removal as an introduction of a flaw is impossible.

Without inter-procedural analysis, there are four ways to treat function calls that are passed pointer parameters. The simplest is to treat them as a no-op, which does not dereference the pointer in any way. This is an aggressive assumption that raises the number of parameters that are in the SAFE or NONE states, making the UNSAFE state more accurate. The second method is to treat them identically to a dereference, which will result in a much higher incidence of variables marked as UNSAFE, many of them inaccurately. A third option is to treat them as safe dereferences, which will increase the incidence of the SAFE state. The fourth option is to introduce an entirely new state for variables that are passed to functions, which will accompany the simple dereference state of the function. With this extra state information, the states would expand to: NONE, SUB-ONLY, SAFE, SAFE+SUB, UNSAFE. The state UNSAFE+SUB is not needed because the safety of the dereference cannot be decreased by any function call if the variable is already in the UNSAFE state.

In order to find flaws that span multiple functions, at least one level of inter-procedural analysis is required. To conduct inter-procedural analysis, the algorithm is run in its current form, saving its results to an intermediate storage that we will call stage 0. After this run, another analysis is performed, with the additional condition that function calls may also be treated as memory accesses. The arguments marked in state UNSAFE are treated identically to a load or store opcode to that memory address; the arguments marked in state SAFE are marked as dereferenced in the current function if they are not already so marked. This will eliminate many of the SUB-ONLY and SAFE+SUB states; however, these states can still exist if the parameter is passed as-is through multiple functions.

It is possible to apply this algorithm multiple times to extend the reach of the inter-procedural analysis; however, it is expected that after a relatively low number of stages, no additional useful information will be found. Multiple levels of inter-procedural analysis will also significantly increase the amount of work required to evaluate a patch or merge.

CHAPTER 3. Change Classification

Checking individual patches for the introduction of flaws can be useful in evaluating a software project that accepts patches from a wide audience of developers. Flaws introduced in a patch can be addressed prior to accepting the patch, preventing the flaw from interfering with others' work during the time between its introduction and fix. Evaluating a patch also gives useful history information to the analysis tool – it can evaluate the code in its previous and current states, and ignore flaws that are not introduced by the patch in question.

For the purpose of evaluating a patch, the "before" version will be treated as good – only changes in the behavior of the function due to the patch are considered when looking for flaws. A static analysis of the files changed in the patch is run on the "before" and "after" states of the patch, and the results are evaluated according to table 3.1.

As an example, if the function in figure 2.1 were changed to the function in figure 3.1, the state of the `list` argument would change from SAFE to UNSAFE, which would cause it to be flagged for inspection. The variable `output`, which was already in the UNSAFE state in figure 2.1, would not be flagged because the change did not introduce the incorrect dereference.

If a function's argument had previously been checked for null in every dereference, then it would execute safely when passed a null pointer. If the change being inspected has introduced an unsafe dereference, then the function can no longer safely accept null for that parameter. If the function's specification or the users of the function have not forbidden passing null pointers in this argument, this

Before	After	Result
Any	NONE	No output; new version has no possible issue
Any	SAFE	No output; new version has no possible issue
NONE	UNSAFE	Flag if requesting extra verbosity.
SAFE	UNSAFE	Flag for inspection
UNSAFE	UNSAFE	No output; previous version was assumed to be good, and has not changed.

Table 3.1 Output of a patch evaluation

```
int pop_counters(int count, record* list,
                 summary_t* output) {
  int i, min = list[0].id, max = list[0].id;
  int inversions = 0;
  if (count == 0 || list == NULL)
    return -EINVAL;
  for(i=0; i < count; i++) {
    if (i && list[i-1].id > list[i].id)
      inversions++;
    if (list[i].id < min)
      min = list[i].id;
    if (list[i].id > max)
      max = list[i].id;
  }
  output->num_seen++;
  if (max - min > output->best_range) {
    output->best_range = max - min;
    output->best_list = list;
    output->min = min;
    output->max = max;
  }
  return inversions;
}
```

Figure 3.1 Example function, unsafe change

$$\begin{tikzcd}
\text{Base} \arrow[r, "1"] \arrow[rrr, dashed, bend right, "\text{Overall Patch}"'] & A \arrow[r, "2"] & B \arrow[r, "3"] & \text{Result}
\end{tikzcd}$$

Figure 3.2 Patch Series

A series of patches represented as a single patch.

will be a potential crash.

The distinction between SAFE and NONE reduces false positives – if a pointer argument was not previously used, its first use is more likely to be correct because the code using the pointer is new, and is therefore under more scrutiny than might otherwise be the case.

3.1 Checking a Patch Series

If a large series of patches is being checked, it may not be practical to evaluate every change to every function using the above algorithm. Having 10,000 patches between releases of a software system is not uncommon in larger projects – the Linux kernel changesets between releases are about this size, and represent 3-4 months of active development. A faster method of checking large a series of patches is needed if this method is to be used to evaluate flaws introduced during release work in addition to evaluating single patches.

To evaluate a large series of patches, the entire series is considered as a single patch and evaluated in the same way as a single patch. Once a list of flagged arguments is created, the state of the software at some point in the middle of the queue is evaluated to determine which half of the queue introduced each bug. By recursively subdividing the patch queue for each flaw found, a single patch can be identified as the patch that introduces a flaw.

3.2 Identifying Functions of Interest

When evaluating a single patch to a large project, it may not be practical to rerun the analysis on every file in the project in the before and after states. For large projects, this may take some time to run, which will make the tool less useful when trying to quickly evaluate a diverse set of patches.

Because only changes in the state of functions are searched, if no inter-procedural analysis is done, then only those functions that have textual changes need to be re-evaluated. Textual changes within a function can come from several sources in C code: from changes in the file itself, or from changes in preprocessor macros that the function refers to. For this reason, it may be beneficial to keep a hash of the preprocessed form of a function in order to check that it is unmodified when a header file changes.

If inter-procedural analysis is done, then a change in the state of function parameters in a function can potentially cause a change in state in any functions that call this function. As the level of inter-procedural analysis increases, the number of functions that a single patch may influence – and which must be rechecked in the patch – will also increase.

CHAPTER 4. Concurrent Development

In large projects, development is commonly done in a distributed manner where each developer works on patches on their own development branch. Once the branch is ready for inclusion in the project, it is merged into the project's master branch. This merge produces code that has not been tested, although each patch that makes up the merge has usually been inspected or tested in some way. This is especially problematic with some version control systems such as git that aggressively try to merge patches.

Because a merge is intended to combine the effects of two development branches, it is possible to predict the state of variables in the result of a merge from the states in the two branches. The intent of a merge is to apply the changes by two distinct developers to a single source code repository. This is normally performed at a textual level; the intent is to perform it at a logical level, which will leave a gap that may introduce flaws.

In order to detect flaws introduced by a merge, the state of the codebase in both branches is used to predict the final state of the merge. A simple heuristic is that the final state should match the state of one of the two branches. However, if information about the state of the variable at the point where the two branches diverged is available, then it is usually possible to make a more accurate prediction. If the state of the variable in the base and the state in one of the branches are identical, then it is expected that the change in the other branch will be reflected in the merge.

Sides		Common Ancestor		
Side A	Side B	NONE	SAFE	UNSAFE
NONE	NONE	NONE	NONE	NONE
NONE	SAFE	SAFE	{NONE,SAFE}	{NONE,SAFE}
NONE	UNSAFE	UNSAFE	UNSAFE	{NONE,UNSAFE}
SAFE	SAFE	SAFE	SAFE	SAFE
SAFE	UNSAFE	{SAFE,UNSAFE}	{SAFE,UNSAFE}	SAFE
UNSAFE	UNSAFE	UNSAFE	UNSAFE	UNSAFE

Table 4.1 Expected state of merge result

$$\begin{tikzcd}
\text{Base} \arrow[r, "1"] \arrow[dr, "2"'] & A \arrow[dr, bend left] & \\
& B \arrow[r, "3"] & \text{Result}
\end{tikzcd}$$

Figure 4.1 Merge process

Changes (1) and (2) are made by different developers concurrently, and are then merged together by (3) to produce a result.

```
int pop_counters(int count, record* list,
                 summary_t* output) {
  int i, min = INT_MAX, max = INT_MIN;
  int prev = -1;
  int inversions = 0;
  if (count == 0 || list == NULL)
    return -EINVAL;
  for(i=0; i < count; i++) {
    if (!list[i].active) {
      output->skip_count++;
      continue;
    }
    if (prev >= 0 && list[prev].id > list[i].id)
      inversions++;
    prev = i;
    if (list[i].id < min)
      min = list[i].id;
    if (list[i].id > max)
      max = list[i].id;
  }
  output->num_seen++;
  if (max - min > output->best_range) {
    output->best_range = max - min;
    output->best_list = list;
    output->min = min;
    output->max = max;
  }
  return inversions;
}
```

Figure 4.2 Example function, side A

```
int pop_counters(int count, record* list,
                 summary_t* output) {
  int i, min = INT_MAX, max = INT_MIN;
  int inversions = 0;
  if (count == 0 || list == NULL)
    return -EINVAL;
  for(i=0; i < count; i++) {
    if (i && list[i-1].id > list[i].id)
      inversions++;
    if (list[i].id < min)
      min = list[i].id;
    if (list[i].id > max)
      max = list[i].id;
  }
  if (output) {
    output->num_seen++;
    if (max - min > output->best_range) {
      output->best_range = max - min;
      output->best_list = list;
      output->min = min;
      output->max = max;
    }
  }
  return inversions;
}
```

Figure 4.3 Example function, side B

```
int pop_counters(int count, record* list,
                 summary_t* output) {
  int i, min = INT_MAX, max = INT_MIN;
  int prev = -1;
  int inversions = 0;
  if (count == 0 || list == NULL)
    return -EINVAL;
  for(i=0; i < count; i++) {
    if (!list[i].active) {
      output->skip_count++;
      continue;
    }
    if (prev >= 0 && list[prev].id > list[i].id)
      inversions++;
    prev = i;
    if (list[i].id < min)
      min = list[i].id;
    if (list[i].id > max)
      max = list[i].id;
  }
  if (output) {
    output->num_seen++;
    if (max - min > output->best_range) {
      output->best_range = max - min;
      output->best_list = list;
      output->min = min;
      output->max = max;
    }
  }
  return inversions;
}
```

Figure 4.4 Example function, proposed merge

Table 4.1 gives the expected state of a variable after a merge, given the state of the variable on the two sides of the merge, and the state of variable in the common ancestor. If the actual state of a variable that is the result of a merge differs from the predicted state, the merge is likely to be incorrect, and should be flagged for inspection.

An example of concurrent development leading to an incorrect merge is shown figures 4.2, 4.3, and 4.4. The variable `output` is initially in the state UNSAFE. Side A added an access to the output variable, which is not a bug because the output currently cannot be null due to a later unchecked access in the function. Side B added a check for null to `output`, changing it to a SAFE state. From the table, the expected result of such a merge is SAFE. However, the code in figure 4.4 has output in an UNSAFE state; it is an incorrect merge of the two changes.

The changes in these functions were chosen so that merge tools would not report a conflict due to different parts of the function changing. In a larger project, the changes could be more widely separated, possibly even changing different source files (a level of inter-procedural analysis would be required to detect the mismerge in this case). A textual merge tool cannot reasonably flag all such conflicts; they are logical conflicts, not text conflicts.

CHAPTER 5. Tool Implementation

A tool implementing this detection mechanism has been developed for testing purposes. This tool is tied to the git revision control system, which was developed for the Linux kernel and has since been adopted by a number of open-source projects. The tool has two modes of execution: in the first mode, it takes a single commit as input (which is either a patch or merge) and evaluates the commit for flaws it may introduce. The second mode implements the flaw-finding algorithm in section 3.1.

The implemented tool was evaluated for its accuracy in finding the type of flaws that it is looking for, and for its time efficiency in execution.

5.1 Accuracy

The tool was run in the flaw-finding mode on the Linux kernel between versions 2.6.13 and 2.6.16. Version 2.6.16 was chosen as the endpoint because it was maintained with bug fixes from its release in in March 2006 until July 2008. This maintenance period contains a number of fixes, some of which are null pointer dereferences. Later kernel versions have not had such a long maintenance period to examine for bugfixes. Detection of the introduction of these bugs is used as a measure of the sensitivity of the tool.

The accuracy of the tool is measured by checking the results found in the analysis. Of the 17 reported unsafe changes found, only 5 of them were introducing a null dereference that could be unsafe. Of the other 12 false positives, 10 were due to the introduction of a dereference in a parameter that was already being passed to other functions. These false positives would be eliminated by changing the state of parameters that were passed through to other functions; however, this change would also eliminate some of the true positives. The other two false positives were due to conditionals that correctly checked the variables, but due to aliasing, were not correctly recorded as checking the variables before dereference.

Of the 5 introductions of flaws that could introduce a null dereference, 3 were changes that do not translate into a dereference that could be triggered using the existing codebase. This is an indication that the actual prototype of the function required the argument to be nonnull, and the checks of the parameter were due to the use of inline functions or macros that were written to check parameters even when this is not strictly necessary. These false positives cannot be eliminated without explicit annotations on the code, because if any other code (including possible third-party modules) relied on the previous behavior of calling the function with a null parameter, this code would break due to the introduction of the unchecked dereference. It is therefore useful to bring these changes to the attention of a maintainer, although detecting them as low-severity warnings would be more useful. Such detection would required that all call sites of the function in question be enumerated, which is not always possible, and then an inter-procedural analysis be done to determine if all calls avoid passing null.

Inspection of the changes between the versions 2.6.16 and 2.6.16.62 showed that null pointer dereferences were fixed in 20 of the changesets, 7 of which were both in the tested build of the kernel and were not dependent on external causes such as incorrect locking. Only 4 of the indicated bugs were null pointer accesses that could have been detected by the method described; the tool was able to detect two of them. Upon manual inspection, the other two involved a dereference in a function different from the checks, which could only be detected by a version of the tool implementing inter-procedural checks.

While the accuracy of 2 true bugs found out of 17 is too low to be directly useful due to the wasted time spent evaluating false positives, it is significantly better than searching the 13,032 identified unchecked dereferences that were found in the 2.6.16 kernel.

5.2 Time Efficiency

The execution of the tool is quite fast. On the test system, it was able to evaluate 1631 files from the 2.6.30 kernel in 28 seconds. The compilation that accompanied the check took 529 seconds, which makes the tool fast enough to be included in any kind of checking system that includes a compilation pass.

These measurements were taken on an Intel Core 2 system, dual-core at 2.13GHz, with 4GB of RAM. The check function was run using the kernel Makefile. The time taken by the check was determined by timing the execution of a compile without the checker enabled, and subtracting this value from the time taken by the execution of a compile with the checker enabled. A direct measurement of the performance on the entire source tree was not practical due to the various compilation flags used in different parts of the kernel source; however, when run on smaller test programs the parsing took an average of 10 milliseconds for a 1000-line file.

5.3 Other tests

The merge testing tool was run on a number of the merges within the Linux kernel, the sparse tool itself, and on the GnuTLS library. No mismerges were detected in any of the tested merge commits. This was primarily due to the high degree of compartmentalization within these projects; most merges are between authors that have been working on distinct areas of the project, and so do not have any conflicts due to their functions not calling one another directly. Additionally, many of the merge bugs are likely to require a level of inter-procedural analysis that was not implemented in the current version of the tool.

A specially crafted merge, derived from the example presented in the discussion of the merge detection, was prepared and tested; the tool was able to detect the incorrect merge within the single function.

CHAPTER 6. Related work

Static analysis of memory access errors such as null pointer dereferences is not a new problem; many analysis tools already exist to find these types of bugs, such as LCLint [Evans et al., 1994], Coverity [Coverity, 2010], or the Clang compiler [clang, 2010]. Tools such as Splint [Evans and Larochelle, 2002] additionally allow for the use of annotations to improve the accuracy of the analysis. Such annotations also provide a method to eliminate false positives detected by the tool by allowing incorrect assumptions to be fixed. These tools all include null dereference detection as part of their static analysis of a single snapshot of code. Incorporating findings provided by these tools into the analysis developed here will increase its accuracy and will allow it to highlight additional bugs.

The tool FindBugs [Hovemeyer et al., 2006] is a Java analysis tool that uses a similar approach of building a control flow graph at the bytecode level (which corresponds fairly closely to the level of the opcodes generated by sparse) in order to find probable null pointer accesses. Instead of using four internal states for each variable, it uses 8 internal states in order to handle paths in try/catch blocks (which are not supported in C) and to provide warnings about some useless checks for null that indicate programming errors. If the technique in this paper is applied to Java code, incorporating the additional states used by FindBugs will increase the usefulness of the output of this program by allowing it to provide more distinctions in its output.

Other tools have also looked at the analysis of patches to source code. Some of these tools such as PMD [Copeland, 2005] and FindBugs [Hovemeyer et al., 2006] inspect the patch itself and match it against known bug patterns. Others such as JADET [Wasylkowski et al., 2007] inspect high-level dependencies between systems and attempt to find violations of described rules. The method described in this paper is a type of pattern-matching that only finds localized bugs. These tools do not have the ability to evaluate a merge except by evaluating it in relation to a single parent.

CHAPTER 7. Further Work

While this method was only applied to null pointer dereferences, the general technique of including history information is applicable to the output of other analyses. Locking requirements for a multi-threaded application is another suitable problem that can be partially solved by static analysis. Improvements to the existing technique can also be made by increasing the state space of the analysis by using the output of other tools (such as FindBugs) directly, possibly adding additional informative states to enable more fine-grained transition detection.

7.1 Locking Safety

There are several important considerations when considering locking safety: locks that are taken must be released in every execution path, locks must be held by all threads accessing a field for it to be correctly protected by the lock, and locks cannot (usually) be taken twice in a single thread. The simplest way to solve these problems is to require a function that acquires a lock to release it, and to keep all accesses to the object protected by the lock within the function.

If this practice is followed, a normal static analysis will produce correct locking behavior – this is one of the checks that sparse can run. False positives are relatively rare, because the lock and unlock functions are easy to identify, and locking imbalances can reasonably be considered bugs even when it is impossible to trigger them due to complex external logic.

History information may be useful in determining what fields inside a structure are protected by a lock. By creating a list (possibly aided by annotations) of all locks held at any point in the control flow graph, locks that are always held while accessing a field can be identified. The field can then be marked in states corresponding to: accessed only while lock is held; read without lock held; written (and read) without lock held. Changes in the state of a field that decrease the memory access safety, such as a new read access to a field that is not protected by a lock, can then be identified and flagged for manual review.

7.2 Annotations

Incorporating existing annotation information into the analysis will allow an increase in accuracy, and will also allow for some changes that would be highlighted by this tool to be correctly identified as false positives. If a parameter has an annotation that declares that the parameters passed are not allowed to be null, the state change from SAFE to UNSAFE does not need to be highlighted because of the existing declaration. This will reduce or eliminate false positives that were present due to adding a dereference in a function that hat previously only passed its parameter to other functions, similar to the introduction of an inter-procedural analysis.

CHAPTER 8. Conclusion

Checking the output of static analysis tools within the context of continuous development and integration can produce useful results. Even with the weak 3-state analysis done in the test implementation, it was possible to narrow down identified flaws to a number feasible for human inspection. Automated verification of merges can provide assurance that the merge process does not introduce flaws due to logical conflicts; as distributed development becomes more popular, this type of verification will become more important. Extension of the patch and merge classification to support more possible states and to evaluate conditions other than null pointer dereferences is an area for future research that should yield similar results.

BIBLIOGRAPHY

[Ayewah and Pugh, 2008] Ayewah, N. and Pugh, W. (2008). A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA. ACM.

[Christopher Li, 2010] Christopher Li, S. T. (2010). Sparse wiki, http://sparse.wiki.kernel.org/.

[clang, 2010] clang (2010). LLVM/clang static analyzer. http://clang.llvm.org/staticanalysis.html.

[Copeland, 2005] Copeland, T. (2005). *PMD Applied*. Centennial Books.

[Coverity, 2010] Coverity (2010). Coverity prevent. http://www.coverity.com/.

[Evans et al., 1994] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. (1994). Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96.

[Evans and Larochelle, 2002] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*.

[Hovemeyer et al., 2006] Hovemeyer, D., Spacco, J., and Pugh, W. (2006). Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19.

[Wasylkowski et al., 2007] Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44. ACM.