Effective Dynamic Detection of Alias Analysis Errors

Jingyue Wu, Gang Hu, Yang Tang, Junfeng Yang
Columbia University, United States
{jingyue,ganghu,ty,junfeng}@cs.columbia.edu

ABSTRACT

Alias analysis is perhaps one of the most crucial and widely used analyses, and has attracted tremendous research efforts over the years. Yet, advanced alias analyses are extremely difficult to get right, and the bugs in these analyses are one key reason that they have not been adopted to production compilers. This paper presents NEONGOBY, a system for effectively detecting errors in alias analysis implementations, improving their correctness and hopefully widening their adoption. NEONGOBY detects the worst type of bugs where the alias analysis claims that two pointers never alias, but they actually alias at runtime. NEONGOBY works by dynamically observing pointer addresses during the execution of a test program and then checking these addresses against an alias analysis for errors. It is explicitly designed to (1) be agnostic to the alias analysis it checks for maximum applicability and ease of use and (2) detect alias analysis errors that manifest on real-world programs and workloads. It emits no false positives as long as test programs do not have undefined behavior per ANSI C specification or call external functions that interfere with our detection algorithm. It reduces performance overhead using a practical selection of techniques. Evaluation on three popular alias analyses and real-world programs `Apache` and `MySQL` shows that NEONGOBY effectively finds 29 alias analysis bugs with zero false positives and reasonable overhead; the most serious four bugs have been patched by the developers. To enable alias analysis builders to start using NEONGOBY today, we have released it open-source at `https://github.com/columbia/neongoby`, along with our error detection results and proposed patches.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Algorithms, Design, Reliability, Experimentation

Keywords

Error Detection, Alias Analysis, Dynamic Analysis

1. INTRODUCTION

Alias analysis answers queries such as "whether pointers `p` and `q` may point to the same object." It is perhaps one of the most crucial and widely used analyses, and the foundation for many advanced tools such as compiler optimizers [30], bounds checkers [20], and verifiers [15, 14]. Unsurprisingly, a plethora of research [33, 24, 25] over the last several decades has been devoted to improve the precision and speed of alias analysis, and PLDI and POPL alone have accepted over 37 alias analysis papers since 1998 [32]. (Most citation lists in this paragraph are seriously incomplete for space.)

Unfortunately, despite our reliance on alias analysis and the tremendous efforts to improve it, today's production compilers still use the most rudimentary and imprecise alias analyses. For instance, the default alias analysis in LLVM for code generation, `basicaa`, simply collapses all address-taken variables into one abstract location; the default alias analysis in GCC is type-based and marks all variables of compatible types aliases. These imprecise analyses may cause compilers to generate inefficient code [28, 16].

We believe one key reason hindering the adoption of advanced alias analyses is that they are extremely difficult to get right. Advanced alias analyses tend to require complex implementations to provide features such as flow sensitivity, context sensitivity, and field sensitivity and to handle corner cases such as C unions, external functions, function pointers, and wild `void*` and `int` casts. As usual, complexity leads to bugs. Buggy alias results at the very least cause research prototypes to yield misleading evaluation numbers. For instance, our evaluation shows that LLVM's `anders-aa`, implementing an interprocedural Andersens's algorithm [9], appeared more precise than `basicaa`, but is actually *less* precise than `basicaa` (§8.1.1) after we fixed 13 `anders-aa` bugs. Worse, buggy alias results cause optimizers to generate incorrect code, commonly believed to be among the worst possible bugs to diagnose. Moreover, they compromise the safety of bounds checkers and verifiers, yet this safety is crucial because these tools often have high compilation, runtime, or manual overhead, and are applied only when safety is paramount.

This paper presents NEONGOBY,[1] a system for effectively detecting errors in alias analysis implementations, improving their correctness and hopefully vastly widening their adoption. NEONGOBY detects the worst type of bugs where the alias analysis claims that two pointers never alias, but they actually alias at runtime. We explicitly designed NEONGOBY to be agnostic to the alias analysis it checks: the only requirement is a standard `MayAlias(p,q)` API that returns true if `p` and `q` may alias and false otherwise.[2] This minimum requirement ensures maximum applicability and ease of use. To check an alias analysis with NEONGOBY, a user additionally chooses a test program and workload at her will. For instance,

[1]We name our system after the neon goby fish which helps other fish by cleaning external parasites off them.

[2]NEONGOBY can be easily extended to check must-alias but few alias analyses implement a more-than-shallow must-alias analysis.

she can choose a large program such as `Apache` and `MySQL` and a stressful workload that together exercise many diverse program constructs, such as the corner cases listed in the previous paragraph. This flexibility enables NeonGoby to catch alias analysis bugs that manifest on real-world programs and workloads.

NeonGoby works as follows. Given a test program, NeonGoby instruments the program's pointer definitions to track pointer addresses. To increase checking coverage, NeonGoby instruments all pointer definitions including intermediate results, *e.g.*, pointer arithmetics. The user then runs the instrumented program on the workload, and NeonGoby dynamically observes pointer addresses (henceforth referred to as *addresses*) and checks them against the alias analysis. It emits bug reports if the addresses contradict the alias results, *i.e.*, the pointers did alias during the test run but the alias analysis states that the two pointers never alias. We use `DidAlias(p,q)` to refer to NeonGoby's algorithm for determining whether pointers `p` and `q` did alias. The invariant NeonGoby checks is thus `DidAlias(p,q)` $\rightarrow$ `MayAlias(p,q)`. To ease discussion, we use `DidAlias`/`MayAlias` to refer to both the corresponding algorithm and the set of pointer pairs on which `DidAlias`/`MayAlias` returns true.

Although the idea of dynamically checking alias analysis enjoys conceptual simplicity, implementing it faces two major challenges: performance overhead and false positives.

Performance overhead. NeonGoby must have low performance overhead for two reasons. First, NeonGoby is designed to detect alias analysis errors that manifest on real-world programs, and large overhead may disturb the executions of these programs [26], such as triggering excessive timeouts. Second, different users may have different resource budgets and coverage goals when testing their alias analyses. Since users can best decide what overhead is reasonable, NeonGoby should provide them the flexibility to make their own tradeoffs between bugs and overhead.

NeonGoby addresses this challenge using two ideas. First, NeonGoby provides two checking mode, enabling a user to select the mode best suited for her alias analysis, test program, and workload. In the *offline* mode, NeonGoby logs pointer definitions to disk when running the test program, and checks the log offline after the test run finishes. Since checking does not slow down the test run, NeonGoby affords to check more thoroughly: it checks alias queries on pointers potentially in different functions, or *interprocedural queries*. However, the logging overhead in the offline mode may be high, so NeonGoby offers another mode to reduce overhead. In the *online* mode, NeonGoby checks alias queries on pointers only in the same function, or *intraprocedural queries*,[3] with efficient inlined assertions, but it may miss some bugs the offline mode catches.

Second, NeonGoby further reduces performance overhead without losing bugs using an optimization we call *delta checking*. This optimization assumes a correct baseline alias analysis, such as LLVM's `basicaa`, often simple enough to have few bugs. NeonGoby then checks only the pointer pairs that may alias according to the baseline but not the checked alias analysis. By reducing the pointer pairs to check, NeonGoby reduces overhead.

False positives. Another challenge NeonGoby faces is how to reduce false positives or FPs, a major factor limiting the usefulness and adoption of error detection tools [11]. Recall that NeonGoby checks the invariant `DidAlias(p,q)` $\rightarrow$ `MayAlias(p,q)`. Therefore, NeonGoby may emit FPs if the `DidAlias` set it computes incorrectly includes pointer pairs that did not alias. `DidAlias` may be incorrect for two main reasons:

```
// no alias if field-sensitive
struct {char f1; char f2;} s;
p = &s->f1;
q = &s->f2;

// no alias if flow-sensitive
for(i=0; i<2; ++i)
  p = ADDR[i];
q = ADDR[0]; // p's address is ADDR[1]

// no alias if context-sensitive
void *foo(int *p, int *q) {
  ... // p and q do not alias
}
foo(ADDR0, ADDR1);
foo(ADDR1, ADDR0);
```

Figure 1: FP examples caused by sensitivities.

First, a naïve `DidAlias` algorithm may not account for all sensitivities that a checked alias analysis offers. Figure 1 shows three examples on which a naïve `DidAlias` may cause FPs. Specifically, if a naïve `DidAlias` considers pointers with one byte apart as aliases (because they likely point to the same object), it may cause a FP for a field-sensitive alias analysis on the first example. If it considers pointers ever assigned the same address as aliases, it may cause a FP for a flow-sensitive alias analysis on the second example or for a context-sensitive alias analysis on the third example. If it does not distinguish the same variable in two different invocations of `foo`, it may cause a FP for a context-sensitive alias analysis on the third example. To reduce such FPs while remaining agnostic to the alias analysis checked, NeonGoby must provide a `DidAlias` that accounts for all sensitivities that a checked alias analysis may offer.

Second, the same observed address of a pointer is not always intended to refer to the same object due to spatial memory layout or temporal memory reuse. Spatially, pointers may have invalid addresses (*e.g.*, go off bound or are uninitialized). For instance, an off-by-one pointer for marking the end of an array may accidentally have the same address as a pointer to the next object. Off-by-one pointers are allowed by ANSI C specification as long as they are not dereferenced. Therefore, NeonGoby must handle them to avoid FPs. Temporally, the same piece of memory may be reused for different objects. For instance, two heap memory allocations may return the same address if the first allocation is freed before the second allocation. Thus, NeonGoby cannot simply claim that two pointers did alias if their addresses are identical; instead, it may need to track whether a pointer is valid and, if so, what object it points to. This problem appears familiar to the problem bounds checkers solve, but it is actually very different: NeonGoby assumes that a test program is largely correct and runs it to detect alias analysis errors, whereas bounds checkers are for preventing buffer overflow attacks. Thus, it is an overkill for NeonGoby to borrow complex bounds-checking techniques such as tracking base and bounds for each pointer.

NeonGoby addresses this challenge using two ideas. First, when computing `DidAlias`, NeonGoby considers two pointers alias only when they have the same address *simultaneously* at a point in an execution. This idea makes `DidAlias` field-sensitive, flow-sensitive, and context-sensitive. For instance, in the first example in Figure 1, NeonGoby considers `p` and `q` do not alias, because their addresses are different. In the second example, NeonGoby considers `p` and `q` do not alias even if `p` points to `ADDR[0]` in the first iteration. In the third example, although `q` in the first invocation and `p` in the second invocation point to the same ad-

[3]Intraprocedural queries can still be answered by interprocedural analyses.

```
void bar(int *r) { *r = 1; } // r aliases q
int main() {
  int *p = (int *)malloc(sizeof(int));
  free(p);
  int *q = (int *)malloc(sizeof(int)); // memory reuse
  bar(q);
  free(q);
  return 0;
}
```

Figure 2: Example test program.

dress, NEONGOBY does not report p and q alias because these two pointer definitions belong to different invocations. Second, NEONGOBY bridges the gap between an address and an object using a practical selection of techniques. For instance, NEONGOBY versions memory, so if a piece of memory is reused, the addresses before and after the reuse get different versions; it also pads memory allocation to tolerate off-bound pointers.

These two ideas enables NEONGOBY to avoid all FPs for test programs that do not (1) have undefined behavior per ANSI C specification, *e.g.*, using pointers off bound by many bytes, or (2) call external functions unknown to NEONGOBY that return reused memory addresses (§3.1).

We implemented NEONGOBY within the LLVM compiler and checked three popular LLVM alias analyses, including (1) the aforementioned `basicaa`, LLVM's default alias analysis; (2) the aforementioned `anders-aa`, later used as the basis for two alias analyses [25, 29]; and (3) `ds-aa`, a context-sensitive, field-sensitive algorithm with full heap cloning [24], later used by [10, 14, 12]. To check these analyses, we selected two real-world programs `MySQL` and `Apache` and the workloads their developers use. NEONGOBY found 29 bugs in `anders-aa` and `ds-aa`, including 24 previously unknown bugs, with zero FPs and reasonable overhead. We have reported eight bugs to `ds-aa` developers, and four most serious ones have been patched [2, 3, 4, 5].

This paper makes four main contributions: (1) our formulation of an approach that dynamically checks general alias analysis with the invariant `DidAlias(p,q)` $\rightarrow$ `MayAlias(p,q)`; (2) NEONGOBY, a long overdue system toward improving advanced alias analyses into production quality and widening their adoption; (3) a practical selection of techniques to reduce overhead and FPs; and (4) our evaluation results, including real bugs found in two LLVM alias analyses and our proposed patches. Our key inspiration is our anecdotal struggles with some existing alias analyses in our research, so we hope that alias analysis builders can start applying NEONGOBY to improve their alias analyses into production-quality analyses today. As such, we have released it open-source at `https://github.com/columbia/neongoby`, along with our error detection results and proposed patches.

2. AN EXAMPLE AND OVERVIEW

Figure 2 shows an example test program. It has three pointers: `p` and `q` in function `main`, and `r` in `bar`. Among these pointers, only `q` and `r` alias. Suppose `buggyaa`, a buggy alias analysis misses this only alias pair and reports no alias for all pointers.

To check `buggyaa` with this test program using the offline mode of NEONGOBY, a user first compiles the code into `example.bc` in LLVM's intermediate representation (IR), and runs the following three commands:

```
% neongoby --offline --instrument example.bc
% ./example.inst
% neongoby --check example.bc example.log buggyaa
```

The first command instruments the program for checking: (1) it transforms the program to avoid FPs caused by memory reuse, off-bound pointers, and undefined values and (2) it inserts a logging operation at each pointer definition, memory allocation, function entry and exit to log information for offline checking. The second command runs the instrumented program `example.inst` to generate a log of pointer definitions, memory allocations, and function calls and returns. The third command checks this log against `buggyaa` for errors. It first computes `DidAlias` for all three pairs of pointers, including pointers not in the same function. It excludes pair `p,q` and pair `p,r` from `DidAlias` even if the two `malloc()` calls return the same address because the versions of the address are different. It includes pair `q,r` in `DidAlias` because `q` and `r` share the same address and version. NEONGOBY then checks `DidAlias` against `buggyaa`, emitting an error report because `MayAlias(q,r)` returns false. To diagnose this error, the user can run NEONGOBY to dump log records or slice the log for records explaining why `q` and `r` did alias (§6.2).

To check `buggyaa` with this test program using the online mode of NEONGOBY, a user runs the following commands:

```
% neongoby --online example.bc buggyaa
% ./example.ck
```

The first command iterates through each function in `example.bc`, queries `buggyaa` on each pair of pointers in the function, and, if `MayAlias` on the two pointers returns false, embeds an assertion that the two pointers never alias at runtime. In this example, NEONGOBY embeds an assertion "`assert(p!=q || p==NULL)`" after the second `malloc()`. The online mode prevents the two memory allocations from returning the same address using a simple trick of deferring memory deallocation. The second command runs the instrumented program `example.ck` to check whether this assertion may be triggered, which never happens.

Each mode of NEONGOBY has pros and cons. The offline mode checks more thoroughly, whereas the online mode checks only intraprocedural queries, missing the bug in `buggyaa`. The offline mode can reuse one log to check multiple alias analyses, amortizing the cost of running the tests, whereas the online mode can check only one alias analysis at a time. However, the offline mode has to log information to disk because the log may grow larger than the RAM for some real-world programs and workloads, and on-disk logging can be costly. In contrast, the inlined assertions the online mode embeds are much faster to check. By providing two modes of operations, NEONGOBY enables a user to select the mode that suits her purpose.

3. OFFLINE MODE

This section describes how NEONGOBY operates in the offline mode. Figure 3 shows the offline mode architecture. It has three components: the *instrumenter*, *logger*, and *offline detector*. Given a program in LLVM's intermediate representation (IR), a more low-level representation of the program than source code, the instrumenter transforms the program to avoid FPs and inserts logging operations to collect runtime information. When a user runs the instrumented program, the logger records pointer definitions, memory allocations, function entries and exits to disk. Since the logger runs within a test program, we explicitly designed it to be simple and stateless, reducing runtime overhead and avoiding perturbing the execution of the test program. After the run finishes, the offline detector checks the log against an alias analysis and emits error reports. Since the logger is much simpler than the other components, we focus on describing the instrumenter (§3.1) and offline detector (§3.2), and give a brief discussion (§3.3).

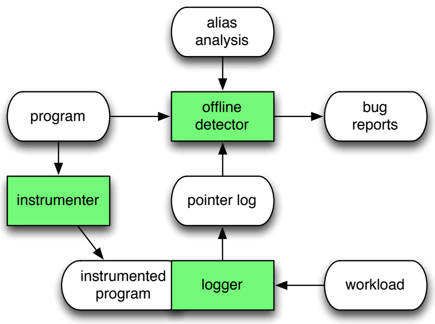

Figure 3: Architecture of the offline mode.

3.1 Instrumenter

The instrumenter does six main transformations, the first to ensure that the logger and the offline detector can consistently refer to the pointer variables in a program, the second to collect pointer addresses, and the last three to reduce FPs caused by memory reuse, off-bound pointers, and undefined values. We describe the six transformations below and highlight some of the differences between NEONGOBY and bounds checkers.

Assigning IDs to pointers. The logger and offline detector run in different phases, so they need a consistent way to refer to the pointers in a program. To identify the pointers, the instrumenter traverses the program's control flow graph and assigns to the n^{th} visited static pointer variable a numeric ID of n. To keep IDs consistent, the offline detector uses the same deterministic algorithm (depth-first traversal in our implementation) to assign IDs to pointers. Bounds checkers need not assign IDs to pointers because they cannot defer checking to offline. To check alias analyses more thoroughly, NEONGOBY identifies not only the pointer variables in the source code, but also intermediate pointers in the LLVM IR.

Instrumenting pointer definitions. To catch errors caused by all different types of pointers, NEONGOBY instruments all pointer definitions which assign addresses to pointers. Function pointers and global pointers are particularly crucial: they are widely used in real-world programs such as `MySQL` and `Apache`, yet they are often mishandled by alias analyses. Our experiments found 7 bugs caused by mishandled function and global pointers (§8.1.1).

NEONGOBY logs four types of pointer definitions: pointer assignment (`p = addr` where `addr` can be a pointer arithmetic), pointer load (`p = *addr`), function argument passing, and global variable initialization. NEONGOBY instruments all pointer definitions in the LLVM IR, including intermediate results. For instance, LLVM translates C source code `p = o->f` into two IR instructions

```
%1 = %o + <offset of f>
%p = load %1
```

, and NEONGOBY instruments the definitions of both `%1` and `%p`. NEONGOBY instruments a pointer assignment or load by inserting "`log_ptr(ptr,addr)`" right after the definition where `ptr` is `p`'s statically assigned ID, `addr` is the address assigned, and `log_ptr` is a logging operation that, when executed, appends the ID and the assigned address to the current on-disk log. NEONGOBY adds calls to `log_ptr` similarly for function arguments at function entries and for global variables at the entry of the `main` function. Bounds checkers must also track these pointer definitions to propagate pointer base and bound information.

Instrumenting memory allocations. As discussed in §1, NEONGOBY cannot use only addresses to compute `DidAlias` because the same address may point to different objects if memory is reused. To enable the offline detector to handle memory reuse, NEONGOBY instruments all memory allocations, including the allocations of heap, stack, and global variables. (Although global memory cannot be reused, NEONGOBY also instruments it to handle all memory allocations uniformly.) For each allocation, it inserts `log_alloc(addr, size)` to record the allocation address and size. For stack variables, NEONGOBY inserts `log_alloc` at the function entry. For global variables, NEONGOBY inserts `log_alloc` at the entry of `main`. For heap variables, NEONGOBY inserts `log_alloc` after a call to one of the following functions:

1. C memory allocation functions: `malloc`, `calloc`, `valloc`, `realloc`, and `memalign`;
2. C++ `new` (mangled names `_Znwj`, `_Znwm`, `_Znaj`, and `_Znam`);
3. Other library functions: `strdup`, `__strdup`, and `getline`.

Users can easily add more heap allocation functions. (One tricky point is that a memory allocation such as "`p = malloc(...)`" is also a pointer definition, and NEONGOBY must insert `log_alloc` before `log_ptr` for the offline detector to correctly handle memory reuse (§3.2). Another point is that LLVM shares memory between two constant strings for space if one is the suffix of the other; to avoid these false aliases, NEONGOBY disables this optimization.)

To reduce logging overhead, NEONGOBY does not instrument memory deallocations, such as `free()` calls, and relies on the offline detector to lazily discover when memory is freed. Thus, a corner case such as "`free(q); p = q;`" may cause NEONGOBY to emit FPs on a flow-sensitive alias analysis that understands `free`: although `p` and `q` have the same address, they technically point to nothing. However, use-after-free behavior is undefined per ANSI C standard. Therefore, as long as the test program is correct, NEONGOBY emits no FPs of this type. Bounds checkers in contrast must handle memory deallocations if they want to catch use-after-free errors.

Instrumenting function entries and returns. NEONGOBY needs to distinguish the same variable in different invocations of a function to reduce FPs on a context-sensitive alias analysis. To track function invocations, for each function, NEONGOBY inserts `log_entry(func)` at the entry, and `log_ret(func)` before each return, where `func` is the ID of the function.

Handling off-bound pointers. Pointers may be assigned off-bound addresses that accidentally alias other pointers, causing FPs. Fortunately, most programs use off-bound pointers only to mark the ends of arrays, and these pointers are off by only one byte. This type of off-bound pointer is also the only type allowed by the ANSI C standard. To eliminate FPs caused by off-by-one pointers, NEONGOBY transforms a program to add one extra byte for each memory allocation, a technique borrowed from [23]. NEONGOBY currently does not handle other off-bound pointers because they occur very rarely, and we experienced no FPs caused by these pointers in our experiments (§8.1.2). Bounds checkers in contrast may have to handle these pointers because their FPs are fatal and abort executions.

Handling undefined values. Variables may be uninitialized and have undefined pointer values that accidentally look like addresses of other pointers. These values may further propagate through assignments, such as assignments of a `struct` with an uninitialized pointer field to another `struct`, causing NEONGOBY to log bogus addresses and emit FPs. NEONGOBY handles undefined pointer values by setting them to `NULL` because `NULL` aliases nothing. Resetting undefined pointer values eliminates FPs caused by undefined values as long as the test program is correct, because ANSI C specification disallows using an uninitialized variables, *e.g.*, pointer arithmetics or wild `void*` and `int` casts.

3.2 Offline Detector

Given a log of pointer definitions and memory allocations, NEONGOBY's offline detector finds alias analysis errors in two steps: it scans the log to compute the `DidAlias` results, and then checks `DidAlias` against an alias analysis and emits error reports.

From a high level, NEONGOBY computes `DidAlias` results as follows. To distinguish a pointer in different function invocations, NEONGOBY assigns a unique context number to each invocation, and maintains a call stack CS of context numbers. It also maintains two maps: a (conceptual) map V from an address to a version number for handling memory reuse, and a map P from a pointer's unique ID and its context number to an address and version for tracking where pointers point to. We use *definition* to refer to a pointer-context pair, and *location* to refer to an address-version pair. Given a log, NEONGOBY scans the records sequentially from the beginning. Upon a memory allocation record, NEONGOBY updates V to assign a new version number for the addresses within the allocated range, so the same addresses get different versions before and after this allocation. Upon a function entry record, NEONGOBY generates a new context number for this invocation, and pushes it to the call stack. Correspondingly, upon a function return record, NEONGOBY pops the call stack so that the caller's context number will be on top. Upon a pointer definition record l with pointer ptr and address $addr$, NEONGOBY searches V for $l.addr$'s current version and updates P to make $\langle l.ptr, top(CS)\rangle$ point to location $\langle l.addr, V[l.addr]\rangle$. It then searches P for pointer definitions that point to the same location; for each such pointer definition d, it adds $\langle l.ptr, d.ptr\rangle$ and $\langle d.ptr, l.ptr\rangle$ to `DidAlias` unless $d.ptr$ and $l.ptr$ are in the same function and their contexts are different. (Note that $\langle l.ptr, l.ptr\rangle$ is in `DidAlias` because a pointer aliases itself.)

As discussed in §1, NEONGOBY must be very precise when computing `DidAlias` to avoid FPs. The algorithm described above is field-sensitive because it considers that two pointers did alias only when their locations are identical, which requires their addresses to be identical. It is context-sensitive because each invocation gets a unique context number, and two pointers in the same function did alias only when they point to the same location in the same context. It is flow-sensitive because (1) the SSA form of LLVM IR guarantees each pointer is statically defined only once, which provides some flow-sensitivity already, and (2) map P maintains only the latest location of a pointer definition.

Once NEONGOBY computes the `DidAlias` results, it checks an alias analysis as follows. It iterates through each pointer pair in `DidAlias`, and checks that the pair is also in `MayAlias`. It emits an error report otherwise. Since the `DidAlias` results do not depend on the alias analysis checked, NEONGOBY can reuse them to check multiple alias analyses, amortizing the cost of logging and computing `DidAlias`.

Algorithm 1: Offline detection algorithm

```
   Input: program $Prog$, alias analysis $A$, and log $L$
1  OfflineDetection($Prog$, $A$, $L$)
2      $P[\forall definition] \leftarrow \langle null, 0\rangle$      // definition-to-location map
3      $Q[\forall location] \leftarrow \emptyset$      // location-to-definitions map
4      $V[\forall address] \leftarrow 0$      // address-to-version map
5      $V_G \leftarrow 0$      // global version number
6      $C_G \leftarrow 0$      // global context number
7      $CS \leftarrow \emptyset$      // call stack
8      $DidAlias \leftarrow \emptyset$      // pointer pairs that did alias
9      $AP[\forall context] \leftarrow \emptyset$      // active pointers
10     foreach record $l \in L$ do
11         if $l$ is MemAllocRecord then
12             $V_G \leftarrow V_G + 1$
13             $V[l.start \cdots l.end] \leftarrow V_G$
14         else if $l$ is EntryRecord then
15             foreach context $C$ of $l.callee$ s.t. $C \notin CS$ do
16                 foreach $p \in AP[C]$ do
17                     $def \leftarrow \langle p, C\rangle$
18                     $Q[P[def]] \leftarrow Q[P[def]] \setminus def$
19                     $P[def] \leftarrow \langle null, 0\rangle$
20                 $AP[C] \leftarrow \emptyset$
21             $C_G \leftarrow C_G + 1$
22             push($CS$, $C_G$)      // push $C_G$ onto $CS$
23         else if $l$ is ReturnRecord then
24             pop($CS$)
25         else if $l$ is PointerRecord then
26             $C \leftarrow 0$
27             if $CS \neq \emptyset$ then
28                 $C \leftarrow$ top($CS$)
29             $def \leftarrow \langle l.ptr, C\rangle$
30             if $P[def] \neq \langle null, 0\rangle$ then
31                 $Q[P[def]] \leftarrow Q[P[def]] \setminus def$
32                 $P[def] \leftarrow \langle null, 0\rangle$
33             if $l.addr$ is null then
34                 continue
35             $loc \leftarrow \langle l.addr, V[l.addr]\rangle$
36             $P[def] \leftarrow loc$
37             $Q[loc] \leftarrow Q[loc] \cup def$
38             $AP[C] \leftarrow AP[C] \cup def.ptr$
39             foreach definition $d \in Q[loc]$ do
                   // parent($p$) returns $p$'s containing function
40                 if parent($d.ptr$) $\neq$ parent($def.ptr$) or
                   $d.context = def.context$ then
41                     $DidAlias \leftarrow DidAlias \cup \langle def.ptr, d.ptr\rangle$
42                     $DidAlias \leftarrow DidAlias \cup \langle d.ptr, def.ptr\rangle$
43     foreach $\langle p, q\rangle \in DidAlias$ do
44         if not A.MayAlias(p, q) then
45             ReportError($\langle p, q\rangle$)
```

Our actual algorithm to detect errors offline, shown in Algorithm 1, does three optimizations for space and speed. The first optimization implements the address-to-version map V with an interval tree whose key is an address range and value the version number of the entire address range (lines 4, 13, and 35). An interval tree is much more space-efficient than a version number per address. Upon a memory allocation record, NEONGOBY removes V's existing address ranges that overlap with the allocated range (because these ranges must have been freed), increments a global version number, and inserts the new range and version to V. Second, instead of scanning the pointer map P for pointers that point to the same location, NEONGOBY maintains a reverse map Q from a location back to pointers (lines 3, 18, 31, 37, and 39). Third, when entering a function, NEONGOBY removes from P all *outdated* pointer definitions in this function (lines 15–20). NEONGOBY considers a pointer definition outdated if its context number is no longer on the call stack. Without removing outdated definitions, $|P|$ could be very large because a pointer can be defined in many function invocations. To efficiently implement this optimization, we use another mapping AP to maintain all active (*i.e.*, non-outdated) pointer definitions indexed by context numbers. With these optimizations, the space complexity of our algorithm is $O(|P| + |M| + |DidAlias|)$, and the time complexity is $O(|L|(\log|M| + \log|P| + N \log|DidAlias|))$, where $|M|$ is the number of memory allocations in the log, and N is the maximum

Table 1: Different techniques in bounds checkers and NeonGoby.

	Bounds	NeonGoby
online only	Yes	No
use alias analysis	Maybe	No
pointer definition	Yes	Yes
pointer metadata	Yes	No
pointer dereference	Yes	No
allocation	Yes	Yes
deallocation	Yes	No
off-bound-pointer	Yes	Only off-by-one
undefined value	Yes	Only pointer

size of $Q[location]$. In our experiments, N never exceeds 400, $|M|$ is typically 1% of $|L|$, and the size of `DidAlias` is less than 10^6.

3.3 Discussion

Some of the problems NeonGoby addresses, such as tracking pointer definitions and handling memory reuse, off-bound pointers, and undefined values, overlap with what bounds checkers must handle. However, NeonGoby has very different assumptions and goals than bounds checkers: it assumes a test program is largely correct, and uses the program to detect errors in alias analyses, whereas bounds checkers prevent buffer overflow attacks to a program. False negatives in bounds checkers may lead to exploits, and FPs wrongly abort executions. In contrast, the effects of NeonGoby's FPs and negatives are much less serious. Because of these differences, it is an overkill for NeonGoby to borrow complex bounds-checking techniques.

Table 1 summarizes the different techniques in typical bounds checkers and NeonGoby. Bounds checkers must check buffer overflows online to stop exploits, whereas NeonGoby can defer costly detection completely offline. Bounds checkers may assume a correct alias analysis (and other static analyses) and use them to remove unnecessary checks, whereas NeonGoby is intended to detect errors in alias analyses. Bounds checkers need to maintain pointer base and bound information with fat pointers, maps, or trees [23], which break backward compatibility or have high overhead. In contrast, NeonGoby maintains no pointer metadata. Bounds checkers check pointer dereferences and track memory deallocations to catch bugs, whereas NeonGoby does neither. Bounds checkers may need to accurately handle pointers off by more than one bytes and undefined integer values to avoid wrongly aborting executions, whereas NeonGoby ignores these cases.

4. ONLINE MODE

NeonGoby's offline mode checks interprocedural alias queries to find more bugs, but its logging may be costly. Thus, NeonGoby provides an online mode to reduce performance overhead. This section describes how NeonGoby operates in the online mode.

Algorithm 2: Online mode

```
  Input: program $Prog$ and alias analysis $A$
1 OnlineInstrumentation($Prog$, $A$)
2     foreach function $F \in Prog$ do
3         foreach pointer definition pair $\langle p, q \rangle \in F$ do
4             if $p$ reaches $q$ and not $A.MayAlias(p, q)$ then
5                 insert "assert($p \neq q$ or $p$ is null)" after $q$
6     foreach external function call $C$ freeing a heap object do
7         replace $C$ with "call deferred_free"
```

The online mode focuses on checking intraprocedural queries because they are often considered more crucial than interprocedural queries. For instance, compiler optimizations tend to issue mostly intraprocedural queries. To check intraprocedural queries, NeonGoby embeds the alias analysis checks as regular program assertions into a test program. NeonGoby reports an alias analysis bug if one of the assertions fails when a user runs the test program. These assertions are much cheaper than costly on-disk logging at runtime, as shown in our experiments (§8.3).

Algorithm 2 shows the algorithm to embed the checks. It iterates through each pair of point definitions `p` and `q` of a function (line 3), and inserts an assertion "`assert(p!=q || p==NULL)`" (line 5) if `MayAlias(p,q)` returns false (line 4). One issue is that the inserted assertion requires that both `p` and `q` are defined. NeonGoby solves this issue with a standard control flow reachability analysis (line 4), and inserts the assertion only if `p`'s definition reaches `q`. (If pointer `p` is undefined along some incoming edges to `q`'s basic block, NeonGoby creates a new ϕ-instruction using LLVM's SSA transformation, not shown in Algorithm 2.)

Similar to the offline mode, `DidAlias` computed in the online mode is very precise. It is field-sensitive, because an assertion fails only when `p=q`; it is flow-sensitive, because an assertion uses the latest addresses of the pointers; it is context-sensitive, because it focuses on intraprocedural queries.

To avoid FPs caused by memory reuse, off-bound pointers, and undefined values, the online mode borrows the techniques from the offline mode, with one refinement: it no longer versions memory. The insight is that NeonGoby checks only intraprocedural queries in the online mode, so it need handle only heap memory reuse, which can be handled in a much simpler way. Specifically, it defers heap memory deallocations so the allocations almost always return different addresses. To do so, it replaces functions that free heap memory, including C's `free` and C++'s `delete` (mangled names `_ZdlPv` and `_ZdaPv`), with a function that queues the free request without actually freeing memory. When the queue is full, NeonGoby processes half of the queued requests, ensuring that heap memory reuse occurs after at least $n/2$ free operations where n is the queue capacity. By default, n is 20K, large enough that no FPs of this type occurred in our experiments.

One additional advantage of the online mode is that the embedded assertions explicitly inform us what to check, enabling NeonGoby to leverage symbolic execution tools such as KLEE and WOODPECKER [13] to generate inputs that cause the assertions to fail. We leave this for future work.

If a function has an extremely large number (denoted n) of pointers that do not alias each other, the online mode need insert $O(n^2)$ assertions, which may run slower than the $O(n)$ logging operations inserted by the offline mode. To avoid high overhead caused by such pathological cases, NeonGoby bounds the number of assertions it inserts to 10^6 for each function, and switches to the offline mode for the function otherwise. In our experiments, we did encounter one such case: a `yacc`-generated function called `MYSQLparse` in `MySQL` needs much more than 10^6 assertions, so NeonGoby always checks this function offline (§8.2).

5. DELTA CHECKING

NeonGoby provides an optimization called *delta checking* to speed up both online and offline modes without losing any error. The insight is that not all pointer pairs are equally hard to handle by an alias analysis, so NeonGoby can focus on checking the hard-to-handle pairs and skip the easy ones. To compute what pairs are easy, NeonGoby takes a user-specified baseline alias analysis assumed to be simple enough to be correct. It then skips checking all pointer pairs `p` and `q` on which the baseline's `MayAlias(p,q)` returns false. Intuitively, if an imprecise baseline alias analysis can infer that two pointers do not alias, then most likely they never alias in any execution, so `DidAlias` would return false and NEONGOBY would not find any error on the pointers.

We envision two ways this optimization can be used. First, a user specifies an alias analysis she trusts, such as `basicaa` which computes very conservative alias results, then enjoys speedup without losing errors when applying NEONGOBY to check an advanced alias analysis. Second, an alias analysis builder incrementally checks each precision improvement she makes to her alias analysis. For instance, if her alias analysis reports 10 pointer pairs that each do not alias prior to the improvement and 50 pairs after, she can use NEONGOBY to check this difference of 40 pairs each indeed never alias on some test programs and workloads.

To implement delta checking for the offline mode, we simply change line 44 in Algorithm 1 to

if B.MayAlias(p, q) **and** not A.MayAlias(p, q)

where B is the baseline alias analysis. To implement delta checking for the online mode, we simply change line 4 in Algorithm 2 to

if p reaches q **and** B.MayAlias(p, q) **and** not A.MayAlias(p, q)

Our results using `basicaa` as baseline show delta checking reduces compilation time, offline detection time, and runtime overhead.

6. IMPLEMENTATION

We implemented NEONGOBY in LLVM. It works with version 3.0 and above. It consists of 5,403 lines of C++ code, with 909 for the instrumenter, 168 for the logger, 875 for the offline detector, 642 for the online mode, and the remaining 2,809 for common utilities.

In the remainder of this section, we describe three additional techniques within NEONGOBY: the first to further reduce overhead (§6.1), the second to help users diagnose error reports (§6.2), and the third to support multiprocess or multithreaded programs (§6.3).

6.1 Detecting Errors Using Dereferenced Pointers Only

Dereferenced pointers are presumably more crucial than the ones not dereferenced, so are the alias results on dereferenced pointers. Thus, NEONGOBY provides users an option to detect alias analysis errors using only dereferenced pointers, including the pointers used in load and store instructions and those passed to external functions because NEONGOBY conservatively assumes that these functions dereference their pointer arguments. Although NEONGOBY with this option may lose some alias analysis errors, it enjoys two benefits. First, the error reports are of higher quality because they are on the more crucial pointers. Second, NEONGOBY runs faster when checking fewer pointer pairs in both offline and online modes. We evaluate this bugs v.s. overhead tradeoff in §8.3.

6.2 Simplifying Error Diagnosis

When NEONGOBY reports an error, it emits two pointers that did alias yet are not marked as aliases by the checked alias analysis. To diagnose such a report, it may be time consuming to manually inspect all records in the log, so NEONGOBY provides a diagnosis tool to slice the log into a small subset of records that explains why two pointers did alias. The core idea is to trace data dependencies of the two pointers back to a common parent pointer from which both pointers are derived. NEONGOBY traces only direct data dependencies on pointers. For instance, given "`p = q + x`" where `p` and `q` are both pointers, NEONGOBY only traces `p`'s dependency on `q`, not `x`. Similarly, given "`p = *q`," NEONGOBY only traces `p`'s dependency on the previous instruction that stores to the address of `q`, and ignores `p`'s dependency on `q`. NEONGOBY stops tracing back when it finds the common parent pointer or it cannot trace the dependencies further due to (for example) external functions whose source is not available to NEONGOBY. To use this tool on an error report, a user needs to (re)run NEONGOBY's logger (§3) to log more operations than pointer definitions and memory allocations, including store instructions that store pointer values and call and return instructions of functions that return pointers.

6.3 Supporting Parallel Programs

As discussed in §1, NEONGOBY is explicitly designed to detect alias analysis bugs that manifest on real-world programs such as `Apache` and `MySQL`. These programs often use multiple threads and processes for performance and ease of programming, so NEONGOBY must handle threads and processes. It needs to do so only in the offline mode because the online mode checks intraprocedural queries. Specifically, NEONGOBY shares one log over all threads in a process, and protects the log using a mutex. It assigns one log to each process. When a process forks, NEONGOBY creates a new log for the child process. NEONGOBY can then check each log in isolation. The only modification to Algorithm 1 is maintaining a call stack for each thread. NEONGOBY assumes race freedom as most compilers do, and data races in the worst case may cause some FPs. Fortunately, data races occur so rarely that no FPs of this type occurred in our experiments (§8.1.2).

7. LIMITATION

False positives. NEONGOBY assumes that test programs do not have undefined behavior per ANSI C specification and may emit FPs on buggy test programs. For instance, NEONGOBY may emit FPs on pointers off bound by many bytes (§3.1) which are disallowed by ANSI C specification. Moreover, NEONGOBY works within a compiler, so external functions that return reused addresses and are not treated as memory allocation functions by NEONGOBY (§3.1) may cause FPs. For instance, if an external function `my_realloc` frees and reallocates heap memory, and returns the reallocated memory address, NEONGOBY would miss this memory reuse and emit FPs. However, in practice, external functions seldom cause issues for two reasons: external functions not treated by NEONGOBY as memory allocation functions seldom return reused addresses; without function summaries, alias analyses cannot characterize the address-reusing behavior of those functions either.

False negatives. NEONGOBY is a dynamic tool, and detects only alias analysis errors that manifest on the executions it checks. Moreover, we explicitly designed NEONGOBY to be general to check many alias analyses with low false positives, but this generality comes at a cost: NEONGOBY cannot easily find bugs that violate a specific precision guarantee intended by an alias analysis. In our future work, we plan to specialize NEONGOBY's checking toward specific precision guarantees by varying the precision of its `DidAlias`. In addition, although NEONGOBY checks that `DidAlias(p,q)` $\rightarrow$ `MayAlias(p,q)`, it cannot dynamically check that if `MayAlias(p,q)`, then there exists an execution s.t. `DidAlias(p,q)`, for the following reasons: (1) `MayAlias` may conservatively return true even if the two pointers never alias in any execution; and (2) even if the pointers do alias in some execution, the given program and workload may not trigger this execution.

8. EVALUATION

We evaluated NEONGOBY on three popular alias analyses:

1. `basicaa`: LLVM's default alias analysis, an intraprocedural, flow-insensitive analysis that collapses all address-taken variables. We chose the version of `basicaa` in LLVM 3.1.

Table 2: Descriptions of the bugs found. Starred bugs were either already reported by others or mentioned in the comments of the code. **File** indicates the file (and the line if there is a clear place to add the fix) containing the bug.

#	AA	File	Description
1	`ds-aa`	TopDownClosure.cpp:207	incomplete call graph traversal in the top-down analysis stage
2	`ds-aa`	StdLibPass.cpp:703	matched formal argument n to actual argument $n+1$
3	`ds-aa`	TopDownClosure.cpp:80	symptom: missed aliases between actual parameters and the return value of an indirect call
4	`ds-aa`	Local.cpp:833	mishandled variable length arguments
5	`ds-aa`	Local.cpp:551	mishandled `inttoptr` and `ptrtoint` instructions
6	`ds-aa`	StdLibPass.cpp	did not handle `errno`; pointers returned from `errno` may alias
7	`ds-aa`	StdLibPass.cpp	did not handle `getpwuid_r` and `getpwnam_r`, whose argument and return value alias
8	`ds-aa`	StdLibPass.cpp	did not handle `gmtime_r`-like functions whose return value and the 2nd argument alias
9	`ds-aa`	StdLibPass.cpp	did not handle `realpath` whose value and the 2nd argument alias
10	`ds-aa`	StdLibPass.cpp	did not handle `getenv` whose return value aliases for the same environmental variable
11	`ds-aa`	StdLibPass.cpp	did not handle `tzname`, an external global variable
12	`ds-aa`	StdLibPass.cpp	did not handle `getservbyname` whose return values may alias
13	`ds-aa`	StdLibPass.cpp	did not handle `pthread_getspecific` and `pthread_setspecific`; the value stored via `pthread_setspecific` aliases that loaded via `pthread_getspecific` with the same key
14*	`ds-aa`	StdLibPass.cpp	did not handle `strtoll`; the dereference of the 2nd argument may alias the 1st argument
15*	`ds-aa`	StdLibPass.cpp	did not handle the `ctype` function family; the return value of `__ctype_b_loc`-like function may alias
16*	`ds-aa`	StdLibPass.cpp	did not handle `freopen` whose return value may alias `stdin`, `stdout`, or `stderr`
17	`anders-aa`	Andersens.cpp:1882	`HUValNum` incorrectly marked a pointer as pointing to nothing.
18	`anders-aa`	Andersens.cpp:2588	mishandled indirect call arguments; points-to edge to argument n may be attached to argument $n \pm 1$
19	`anders-aa`	Andersens.cpp:2585	points-to nodes representing indirect calls are swapped, but argument info is not updated accordingly
20	`anders-aa`	Andersens.cpp:764	queries on a function pointer and a function always return no alias, even though they do alias
21*	`anders-aa`	Andersens.cpp	did not handle `inttoptr` and `ptrtoint` instrucitons
22*	`anders-aa`	Andersens.cpp	did not handle `extractvalue` and `insertvalue` instructions
23	`anders-aa`	Andersens.cpp:924	incorrect summary for `freopen` whose return value may alias the 3rd argument
24	`anders-aa`	Andersens.cpp	did not handle `__cxa_atexit`
25	`anders-aa`	Andersens.cpp	mishandled variable length arguments
26	`anders-aa`	Andersens.cpp	did not handle `pthread_create`
27	`anders-aa`	Andersens.cpp	did not handle `pthread_getspecific` and `pthread_setspecific`
28	`anders-aa`	Andersens.cpp	did not handle `strcpy`, `stpcpy` and `strcat` whose return value aliases the 1st arguments
29	`anders-aa`	Andersens.cpp	did not handle `getcwd` and `realpath`

2. `ds-aa`: a context-sensitive, field-sensitive alias analysis with full heap cloning [24], actively maintained by LLVM developers. `ds-aa` is used by [10, 14, 12]. We chose revision 160292 from `ds-aa`'s SVN repo [8].
3. `anders-aa`: an interprocedural Andersen's alias analysis with three constraint optimizations: hash-based value numbering [18], HU [18], and hybrid cycle detection [17]. We ported the version of `anders-aa` in LLVM 2.6 to LLVM 3.1.[4]

Both `anders-aa` and `ds-aa` have much better quality than typical research-grade analyses; `ds-aa` in particular is used by many researchers, regularly tested, and actively maintained.

Our test programs are `MySQL` and `Apache`, two widespread server programs. Our workloads are benchmarks used by the server developers themselves: `SysBench` [7] for `MySQL`, which randomly selects, updates, deletes and inserts database records; and `ApacheBench` [1] for `Apache`, which repeatedly downloads a webpage. We compiled these programs and benchmarks with Clang 3.1 and `-O3`. Since `MySQL` and `Apache` are server programs, we quantified NEONGOBY's overhead on them by measuring throughput.

Our evaluation machine is a 2.80 GHz Intel dual-CPU 12-core machine with 64 GB memory running 64-bit Linux 3.2.0. We made both `SysBench` and `ApacheBench` CPU bound by fitting the database or web contents in memory; we ran both the client and the server on the same machine to avoid masking NEONGOBY's overhead with network delay; we used four threads for the server and client, and split the total eight threads on different cores to avoid CPU contention.

[4]`anders-aa` was maintained up to LLVM 2.6, so we ported it to LLVM 3.1 with a patch that removes 67 lines and adds 115. This patch is included in our release of NEONGOBY. It does not change `anders-aa`'s functionality; it merely fixes compatibility issues between LLVM 2.6 and 3.1: it replaces debug output `dout` with `dbgs`; migrates `anders-aa`'s handling of an allocation instruction because LLVM 3.1 replaces this instruction with other instructions; adds code to handle a new type of constant (`ConstantDataSequential`); and changes the alias query interface to include sizes. For each bug found in our port, we verified that the bug also exists in the original `anders-aa`.

The remainder of this section focuses on three questions:

§8.1: can NEONGOBY detect many bugs with zero FP?
§8.2: what is NEONGOBY's overhead?
§8.3: what are the bugs v.s. overhead tradeoffs with different NEONGOBY techniques?

8.1 Bug Detection Results

This subsection shows the bugs (§8.1.1) we found using NEONGOBY and how we detected them (§8.1.2).

8.1.1 Bugs Found

NEONGOBY found total 29 bugs, 16 in `ds-aa` and 13 in `anders-aa`. Of the 29 bugs, 24 are previously unknown, and four `ds-aa` bugs have been fixed by the developers [2]. Table 2 shows all bugs. Of the 29 bugs, nine (1–5, 17–20) are logical bugs; two (21, 22) do not handle certain LLVM instructions; the remaining eighteen mishandle external functions or global variables because of analysis incompleteness. Although adding summaries for external functions and global variables can alleviate analysis incompleteness, systematically handling unknown external functions and global variables remains challenging. Researchers and developers strive to make their analyses complete to adopt them into production. For instance, one novel feature of `ds-aa` is completeness tracking technique to support unknown external functions [24]; `anders-aa` also strives to handle the incompleteness of external functions (see function `AddConstraintsForCall` [6]).

We pinpointed the root causes of all bugs in Table 2 to locations in the source code. Since `anders-aa` is relatively simple, we fixed all bugs NEONGOBY detected in this alias analysis.

Table 3: Alias analysis precision. Percentages are no-alias ratios.

	basicaa	anders-aa	fixed anders-aa
Apache	10.9%	24.3%	10.5%
MySQL	3.7%	5.1%	2.7%

Next we elaborate on two most interesting bugs: bug 1 in ds-aa and bug 17 in anders-aa. Both cause the points-to graphs to miss edges, and require tricky fixes.

Bug 1 is caused by an incomplete call graph traversal in ds-aa. ds-aa constructs its point-to graph in three stages: constructing a local point-to graph for each function, a bottom-up analysis to clone each callee's point-to graph into the caller, and a top-down analysis to merge each caller's point-to graph into the callees. The bottom-up stage computes an unsound call graph G_b, and the top-down stage computes a sound graph G_t based on G_b by merging nodes and adding missing edges. Suppose the top-down stage merges node A and B of G_b into node C of G_t. When the top-down stage traverses G_t, it needs to traverse both A and B within node C. However, the code incorrectly traverses only one of them. We reported this bug to ds-aa developers and they have fixed this bug.

Bug 17 is caused by an incomplete depth-first search (DFS) of the constraint graph in anders-aa's implementation of the HU algorithm. anders-aa answers alias queries by collecting and solving load, store, assignment, and address-of constraints. It organizes these constraints in a constraint graph. It runs HU to identify the points-to sets of pointers and unify the pointers with the same points-to sets. To do so, it runs a DFS over all nodes. It keeps a visited flag per node (Node2Visited), and sets the flag to true when it first reaches the node. As an optimization, when visiting a node representing *p, if the points-to set of the node representing p is already determined to be empty, anders-aa simply sets the points-to set of *p to be empty. The bug lies in anders-aa's logic to determine *when* the points-to set of p is already determined: it wrongly believes the set is determined when p's visited flag is true, even though it has not finished exploring p's descendants or even initialized p's points-to set. We fixed this bug by adding a new flag per node to indicate when DFS has finished exploring the node.

How bugs affect precision. As discussed in §1, alias analysis bugs may cause tools to mistakenly believe that pointers do not alias when they indeed do, invalidating research findings and compromising safety. To illustrate, we measured how bugs affect alias analysis precision using LLVM's AliasAnalysisEvaluator, which statically queries an alias analysis with all intraprocedural pointer pairs and computes statistics of the results. We define precision as the percentage of queries with no-alias results over all queries. Table 3 shows the precision of basicaa, anders-aa, and the anders-aa after we fixed all its detected bugs. Although anders-aa appears more precise than basicaa on both MySQL and Apache, the fixed anders-aa is actually *less* precise than the supposedly very imprecise basicaa. This results illustrates that buggy alias results can indeed invalidate evaluation numbers.

8.1.2 Bug Detection Methodology

To detect as many bugs as possible, we ran NEONGOBY in the most thorough way: the offline mode without any optimization. (§8.3 shows how the number of bugs varies with different modes and optimizations.)

Our results show NEONGOBY emits no FPs on the three alias analyses and the two applications we checked. We verified all NEONGOBY's reports are true positives as follows. For anders-aa, NEONGOBY emitted many reports. Fortunately, one bug typically causes thousands of reports, so we verified the reports as follows. We diagnosed one report, produced a patch, reran NEONGOBY on the patched anders-aa to regenerate reports, marked the reports that disappeared as true positives, and repeated. After about 10 iterations, NEONGOBY emitted no more reports. For ds-aa, NEONGOBY emitted a relatively small number of reports, so we manually inspected each report. Some reports are fairly simple to diagnose, such as incorrect external function summaries. For more complex ones, we created small testcases to reproduce the problems or applied our diagnosis tool (§6.2) to compute a slice of relevant log records to simplify diagnosis. We confirmed all ds-aa reports as true positives, and pinpointed the root causes in ds-aa's code for about half of the reports. We could not pinpoint the other reports or reproduce them with small testcases, so we conservatively excluded them from our bug count. Thus, the actual number of ds-aa bugs found may be significantly larger than what we report. We released our bug reports together with NEONGOBY.

Table 4: Bug reports and bugs. The second row shows the number of bugs found from the reports. The numbers of ds-aa bugs may be significantly larger than those shown in the table (starred) because we did not count a report as a bug if we could not pinpoint its root cause in the source or reproduce it with a simple testcase.

	ds-aa		anders-aa	
	MySQL	Apache	MySQL	Apache
Bug reports	395	162	20,551	4,945
Bugs	10*	7*	13	9

Table 4 shows the results on ds-aa and anders-aa with our test programs and workloads. (We did not include basicaa because NEONGOBY emitted no reports on it.) For ds-aa, NEONGOBY emitted 395 reports on MySQL and 162 on Apache. For anders-aa, NEONGOBY emitted 20,551 on MySQL and 4,945 on Apache.

The second row of Table 4 shows the number of bugs found. NEONGOBY found at least 10 ds-aa bugs with MySQL and 7 with Apache. Interestingly, these two sets of bugs only overlap by one bug, illustrating NEONGOBY's benefit of using real-world programs with diverse programming constructs as testing programs. NEONGOBY found 13 anders-aa bugs with MySQL and 9 with Apache, and the Apache bugs are a subset of the MySQL ones.

8.2 Overhead

To quantify NEONGOBY's overhead, we ran it in the most optimized way: the online mode with all optimizations. (§8.3 shows how the overhead varies with different modes and optimizations.) Table 5 shows the results on basicaa, anders-aa, the fixed anders-aa, and ds-aa. The compilation time of Apache for every checked alias analysis is within 50s. The compilation time of MySQL is relatively longer mostly because anders-aa and ds-aa are slower on MySQL. The throughput highly depends on the precision of the alias analysis. For instance, the throughput for ds-aa is smaller than that for basicaa, because ds-aa is more precise. Interestingly, the bugs in anders-aa made it appear very "precise", so its throughput is also small. However, after we fixed all its bugs, its throughput almost doubled. NEONGOBY checks function MYSQLparse offline (§4), so we also measured this time. Since NEONGOBY logged only operations from MYSQLparse, the log was very small, and most of the offline detection time was spent on querying the checked alias analysis.

8.3 Bugs and Overhead Tradeoffs

NEONGOBY provides both the offline and online modes and several optimizations to enable users to flexibly trade bugs for low overhead. This subsection evaluates these tradeoffs using

Table 5: NeonGoby's overhead. Compile shows the total compilation time including the time to query the checked alias analysis (**AA**), insert alias checks (**Insert**), and generate the executable from the transformed bitcode (**Codegen**). **TPUT** shows the relative throughput with NeonGoby over without. **Detect** shows the offline detection time for function `MYSQLparse`; NeonGoby checks it offline because this `yacc`-generated function has too many pointers (§4). All times are in seconds.

	MySQL				Apache			
	`basicaa`	`anders-aa`	fixed `anders-aa`	`ds-aa`	`basicaa`	`anders-aa`	fixed `anders-aa`	`ds-aa`
Compile	130.65	421.59	738.43	1714.02	19.21	41.73	22.46	39.37
AA	8.53	213.35	656.83	1493.30	0.53	3.93	7.01	1.90
Insert	65.30	89.60	47.72	113.26	10.35	18.88	10.49	19.07
Codegen	56.82	118.64	33.88	107.46	8.33	18.92	4.96	18.40
TPUT	59.47%	33.32%	75.16%	34.78%	68.05%	41.58%	81.95%	45.62%
Detect	48.02	244.08	679.47	1536.63	n/a	n/a	n/a	n/a

Table 6: Bugs and overhead tradeoffs. The **base** columns represent the baseline of the offline and online modes; **delta** with delta checking (§5); **delta+deref** with both delta checking and using dereferenced pointers only (§6.1). The row titles match Table 4 and Table 5. To collect all reports in the online mode, we changed the online mode to emit a report upon an error instead of aborting the current execution.

	offline			online		
	base	**delta**	**delta+deref**	**base**	**delta**	**delta+deref**
Compile	3.82	3.82	2.92	226.97	189.62	41.73
TPUT	4.83%	4.83%	11.56%	15.52%	16.17%	41.58%
Detect	1373.6	1204.8	579.08	n/a	n/a	n/a
Reports	4945	4945	2155	3861	3861	2068
Bugs	9	9	8	8	8	6

`anders-aa` because we have understood and fixed all its bugs. We chose `Apache` as the test program. Table 6 shows the results.

Offline v.s. online. Columns **base** show that the online mode trades compilation time and a few bugs for significantly increased throughput and reduced detection time. With less than 230s complication time, the online mode improves the throughput of `Apache` by about three times, and eliminates the offline detection time of about 1500s. It emits 22% fewer reports, but misses only one bug. A bug often triggers many reports, so NeonGoby can still catch a bug as long as some of its reports are emitted.

Delta checking. This optimization improves performance for both the offline and online modes without losing bugs (§5). We chose `basicaa` as the baseline. Columns **delta** show that delta checking reduces the detection time by 1.89% in the offline mode; it reduces compilation time by 16.46% and increases the throughput by 4.19% in the online mode. The improvements would be even larger if a user incrementally checks her refinements to her alias analysis.

Detecting errors using dereferenced pointers only. This optimization improves the performance of both modes, but may lose bugs (§6.1). Columns **delta+deref** show that this optimization reduces the compilation time by 77.99% for the online mode; it increases the throughput for both offline and online modes by 80.28% and 157.14% respectively; it reduces the offline detection time by 45.62%; and it misses 1 out of 9 bugs in the offline mode, and 2 out of 8 in the online mode.

9. RELATED WORK

Previous sections have discussed how NeonGoby is related to bounds checkers (or general memory safety tools); this section discusses other related work.

Alias analysis. A plethora of work has been devoted to creating faster, more precise alias analyses [33, 24, 25]. This previous work is complimentary to ours because our goal is to effectively detect errors in alias analysis implementations. There have been several studies on alias analyses, though their focus is on precision and overhead, not correctness. Specifically, Mock et al. [28, 27] measures the precision of static pointer analyses by comparing their results with dynamic points-to sets. However, the dynamic points-to sets they computed are not precise enough for error detection, *e.g.*, the points-to sets are not context-sensitive and they did not handle address reusing. LLVM's `AliasAnalysisEvaluator` collects statistics about an alias analysis, such as how many pointer pairs do not alias and how many may alias. Hind and Pioli [21] implemented six context-insensitive alias analysis algorithms and compared their precision, time and memory consumption on 24 programs up to 30 K lines of code. Jablin et al. [22] compared the performance of their system using different alias analyses, and found that the combination of research grade alias analyses [19, 24, 25] sometimes performs worse than the production-quality alias analysis in LLVM.

Software error detection. A plethora of work has also been devoted to software error detection or verification (*e.g.*, [11, 34]). Most of these systems target general programs, whereas NeonGoby targets alias analyses. These analyses take programs as inputs, do complex computations, and compute abstract results with difficult-to-specify guarantees. Thus, prior systems are not directly applicable to detect alias analysis errors. Testing and verifying compilers [31] has also been an important topic for programming language researchers, though, to the best of our knowledge, we are not aware of any prior system for effectively detecting alias analysis errors.

10. CONCLUSION

We have presented NeonGoby, a system for effectively finding alias analysis bugs. NeonGoby dynamically observes pointer addresses and emits errors if the addresses contradict an alias analysis. Our results show that NeonGoby can effectively detect many bugs in popular alias analyses with zero FPs and reasonable overhead. Our key inspiration of this work is our anecdotal struggles with some existing alias analyses, so we hope that NeonGoby can help improve advanced alias analyses into production-quality analyses and vastly widen their adoption. As such, we have released it open-source at `https://github.com/columbia/neongoby`, along with our error detection results and proposed patches.

11. REFERENCES

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Bug 12744 - Missing Call Edges, LLVM Bugzilla. `http://llvm.org/bugs/show_bug.cgi?id=12744`.

[3] Bug 12786 - External Function Summaries, LLVM Bugzilla. `http://llvm.org/bugs/show_bug.cgi?id=12786`.

[4] Bug 14147 - Handling of Variable Length Arguments, LLVM Bugzilla. `http://llvm.org/bugs/show_bug.cgi?id=14147`.

[5] Bug 14190 - Handling Function Pointers, LLVM Bugzilla. `http://llvm.org/bugs/show_bug.cgi?id=14190`.

[6] LLVM's Implementation of Andersen's Interprocedural Alias Analysis. `http://llvm.org/svn/llvm-project/llvm/branches/release_26/lib/Analysis/IPA/Andersens.cpp`.

[7] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`.

[8] `ds-aa`'s svn repository, revision 160292. `http://llvm.org/svn/llvm-project/poolalloc/trunk`.

[9] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, Feb. 2010.

[12] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, 2007.

[13] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[14] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, pages 144–157, 2006.

[15] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, 2008.

[16] A. Diwan, K. S. Mckinley, J. Eliot, and B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23:30–72, 2001.

[17] B. Hardekopf and C. Lin. The Ant and the Grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 290–299, 2007.

[18] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *14th International Static Analysis Symposium (SAS '07)*, pages 265–280, 2007.

[19] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages (POPL '09)*, pages 226–238, 2009.

[20] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of 10th International Symposium on Code Generation and Optimization (CGO '12)*, 2012.

[21] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. In *Science of Computer Programming*, pages 31–55, 1999.

[22] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of 10th International Symposium on Code Generation and Optimization (CGO '12)*, 2012.

[23] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of The Third International Workshop on Automatic Debugging (AADEBUG '97)*, pages 13–26, 1997.

[24] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

[25] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages (POPL '11)*, pages 3–16, 2011.

[26] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, June 2009.

[27] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*, pages 71–80, 2002.

[28] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*, pages 66–72, June 2001.

[29] R. Nasre and R. Govindarajan. Prioritizing constraint evaluation for efficient points-to analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*, pages 267–276, 2011.

[30] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages (POPL '11)*, pages 159–172, Jan. 2011.

[31] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 335–346, 2012.

[32] L. Shang. Pointer analysis in pldi/popl from 1998. `http://www.cse.unsw.edu.au/~shangl/topconf.htm`.

[33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 131–144, June 2004.

[34] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.