

PtrTracker: Pragmatic Pointer Analysis

Sebastian Biallas
RWTH Aachen
biallas@embedded.rwth-aachen.de

Mads Chr. Olesen
Aalborg University
mchro@cs.aau.dk

Franck Cassez
NICTA and UNSW
franck.cassez@nicta.com.au

Ralf Huuck
NICTA and UNSW
ralf.huuck@nicta.com.au

Abstract—Static program analysis for bug detection in industrial C/C++ code has many challenges. One of them is to analyze pointer and pointer structures efficiently. While there has been much research into various aspects of pointer analysis either for compiler optimization or for verification tasks, both classical categories are not optimized for bug detection, where speed and precision are important, but soundness (no missed bugs) and completeness (no false positives) do not necessarily need to be guaranteed.

In this work we present a new pointer analysis tool for C/C++ code. The tool introduces the notion of heap graphs that are inspired by shape analysis without the computational overhead, but also without the verification soundness guarantees. We explain the underlying ideas and that it lends itself to a fast, modular and incremental analysis, features that are essential for large code bases.

To demonstrate the practicality of the solution we integrate the pointer analyzer into the C/C++ bug checking tool Goanna. We show that run-times of the new analyzer are close to compile times on large code bases and, most importantly, that the new solution is able to reduce false positives as well as to detect previously unknown pointer bugs in the Git source code.

I. INTRODUCTION

Pointer Analysis. A pointer analysis (or alias analysis) determines a set of all possible (symbolic) memory locations a pointer variable might point to during execution. The result of a pointer analysis is for instance instrumental in the *optimization pass* in compilers (e.g. to optimize register reloads) and also in the *static analysis* (verification) of programs (e.g. to check for possible NULL pointer dereferences.) However, the type pointer analysis required depends on the subsequent phase: the optimizing pass of a compiler uses the pointer analysis to optimize register reloads; the analysis must be fast, conservative but not necessarily precise. For verification purposes, the pointer analysis is used to improve the accuracy of the static analysis phase and to remove *false positives* (spurious warnings). A conservative (sound) static analysis usually assumes that every pair of pointers can alias each other and this causes some false positives. To remove the false positives, the pointer analysis must be more precise than for optimization purposes and thus it is usually slower and more memory intensive. Since pointer analyses play such a central role, a multitude of different analyses with different scopes

and granularity have been extensively studied in the past [1] and this is still an ongoing research topic [2].

Pointer Analysis for Static Analysis. It is now well established that *sound* static analysis is time-consuming and generates a lot of false positives. If we drop the soundness requirement for a *best effort* to find bugs, we can significantly gain in efficiency and accuracy [3].

In this paper, we consider pointer analysis in the context of bug finding without the requirement that our bug finding tool be sound (no bug is missed, i.e. no false negatives) or complete (every warning issued by the tool is true, i.e. no false positives). In this respect our pointer analysis is the middle-ground between a fast and coarse optimization-oriented pointer analysis and a sound and computationally expensive verification-oriented pointer analysis.

We distilled three key requirements for PtrTracker to make it applicable in the static analysis of large code bases composed of thousands of files and with millions of lines of codes: (1) the pointer analysis must be fast (less than the compile time) while still being precise; (2) the pointer analysis must be modular and incremental; modular means that we can analyze a function f without knowing the calling context or the callees and build a summary for the function f ; incremental means that we can later enrich the summary of f if in the course of the static analysis we later analyze a file containing a function g called by f and collect some useful information (summary) for g ; (3) in addition, it should be able to integrate with an existing tool with minimal adjustments to incorporate our pointer analysis results. In our case, we want to integrate in the tool Goanna [4], a state-of-the-art bug finding tool for C/C++ which yet lacks a fast and precise pointer analysis module.

An example of a bug we would like to find is represented by this C fragment:

```
void foo(struct bar *a, struct bar *b) {
    b = a;
    a->f = 0;
    42 / b->f;
}
```

A classic division by zero bug is masked by pointer indirection. If we do not detect/assume that a and b can alias, this bug will remain undetected. In real-world cases the statements are spread out with hundreds of lines in between, including branching or function calls.

Related Work. The area of pointer (alias) analysis has been researched extensively [1], [5], [2], mostly targeting compiler

optimization techniques, and focusing on computing *may-point-to* information. For static program analysis (i.e. bug finding), *must-point-to* information is much more valuable than *may-point-to* information, as it enables to decrease the false-positives rate. There is a substantial body of work addressing pointer analysis in the context of static analysis [6], [7] but the analysis times reported in the experiments are unlikely to scale to large code bases. Our own previous attempt at integrating pointer analysis in Goanna [8], resulted in a static analysis time increased by a factor of three to six, which is too slow in practice. PtrTracker is inspired by shape analysis [9]. Shape analysis is claimed to scale up [10], still the analysis times are far from those expected from an industrial-grade bug finding tool.

Our Contribution. We present PtrTracker, a tool that annotates all pointer dereferences in a program with a set of possible pointees, which themselves are normal variables. This information is gathered in a flow-sensitive way while we lazily introduce new symbolic names for traversed data structures. As we will show, this approach is very fast while offering the required precision to find bugs and suppress spurious warnings depending on alias information.

II. PRELIMINARIES AND THE EXISTING GOANNA ARCHITECTURE

Tool Overview. PtrTracker builds on and integrates in Goanna [4], [11], an automata-based static analyzer for detecting software bugs, memory leaks and security vulnerabilities in C/C++ programs. Goanna combines static analysis techniques (e.g. abstract interpretation) with CTL model-checking. The tools' high-level architecture is depicted in Figure 1. Goanna is used as a drop-in replacement for the compiler (e.g. gcc), for easy integration into existing tool chains. It takes as input a project composed of C/C++ files, a set of *checks* which are properties that characterize the presence of bugs (if the property is not satisfied there is a bug) and an optional database that records (previously analyzed) functions' summaries. Goanna analyses programs file by file and cross-relates information. For each file, it first parses the file and builds an Abstract Syntax Tree (AST) in the form of an XML document; this XML document is central in the architecture. The XML is subsequently annotated by a *value interval analysis*, where each AST node is annotated with interval information (e.g. range of some integer variables.) This annotated AST, AST_2 is converted to a Control-Flow Graph (CFG). The last stage of the analysis is performed by a model-checker that takes as input a set of CTL formulae (the formal definitions of the checks), the CFG and the optional database of summaries. In case a formula is not satisfied by the CFG, the model-checker generates a warning and returns a witness trace (sequence of instructions) to explain the bug. The model-checker features state-of-the-art abstraction refinement techniques to remove spurious bugs (that are artifacts of the abstract CFG.) Each trace (counter-example) returned by the model-checker is analyzed (False-Positive Elimination) using an SMT-solver [12] and if it is spurious (infeasible in the

concrete C/C++ semantics), the CFG is refined accordingly and the model-checker run again on the refined CFG. This enables us to remove spurious counter-examples (false-positives) and get more accurate analysis results. Notice that for each file, for each function in the file, a *summary* of the analysis can be stored in a database and can be re-used in subsequent analysis or in the analysis of files that are processed later. If the optional database Database₁ is used, it is enriched after each function analysis and updated to become Database₂.

Variables Binding in Goanna. Goanna has limited capability w.r.t the analysis of variables' ranges (and pointers): some variables are ignored if not *bound* to explicitly *declared* variables. To illustrate these concepts, consider the expression $a + b$ where a and b are integers. The AST's XML representation of the expression is

```
<Op2 op="Add">
  <Ref name="a" binder="7" />
  <Ref name="b" binder="8" />
</Op2>
```

An Op2 node is any binary operation, and a Ref node is a variable reference, where the *binder* uniquely represents the variables in the program (in cases a local variable overlaps a global variable.) A binder refers back to an XML node (7) containing the declaration of this variable. The interval analysis determines some possible ranges for the variables and this information is stored in the *annotated* XML AST_2 : *min* and *max* are new attributes added to the relevant nodes. For instance if we infer that a is in the interval $[1, 2]$ and b in $[42, 42]$, the node Op2 is annotated with $[43, 44]$, because the result of the expression $a + b$ is in $[43, 44]$.

```
<Op2 op="Add" min="43" max="44">
  <Ref name="a" binder="7" min="1" max="2" />
  <Ref name="b" binder="8" min="42" max="42" />
</Op2>
```

The current interval analysis only knows about *binders*, and because binders only exist for variables that are *syntactically* declared, some variables are ignored in the interval analysis. As an example consider a struct pointer dereference $p \rightarrow f$, which is represented in the AST's XML as:

```
<Op2 op="Arrow">
  <Ref name="p" binder="7" is_pointer="true" />
  <Field name="f" />
</Op2>
```

Because $p \rightarrow f$ is not explicitly *syntactically* declared, the interval analysis (and subsequent steps) ignores it. Notice that it does not matter whether field f above is a basic type or a pointer. It is ignored in the interval analysis. To be able to reason about pointers, we have to overcome the previous limitation imposed by explicitly declared binders. The solution to this is to introduce new variables (representing memory locations), which are binders, for syntactic items such as $p \rightarrow f$. Notice that we do this in a *lazy* manner, only when such a binder is needed.

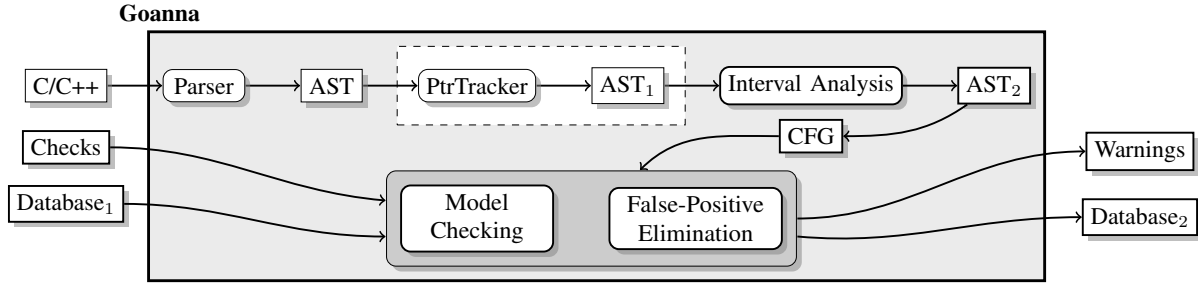


Figure 1. The architecture of Goanna, with the integration of PtrTracker highlighted.

III. PTRTRACKER: POINTER ANALYSIS WITH HEAP GRAPHS

A. Pointer Operations in C

The foundation for our pointer analysis builds on the facts that (1) C pointers are either an l-value (something on the left-hand side of an assignment) or an r-value (something on the right-hand side of an assignment) and (2) all pointer operations in C can be encoded with three basic operators:

- 1) **addr**: taking the address of an l-value and returning it as an r-value, e.g. `&p`,
- 2) **deref**: dereferencing an address (an r-value) and returning the l-value it points to, e.g. `*p`,
- 3) **value-of**: doing a deref followed by a addr operation. This is a derived operation, and needed in cases of pointer arithmetic and pointer assignments such as `p = q` where `q` is an l-value, but needs to be turned into an r-value. For our analysis, this is semantically equivalent to `p = &*q`.

The relationship between the pointer operations and the basic syntactic C expressions is rather straightforward. Figure 2 shows the semantics of the basic operators. Array references such as `p[i]` are handled as `*(p+i)` according to the C standard.

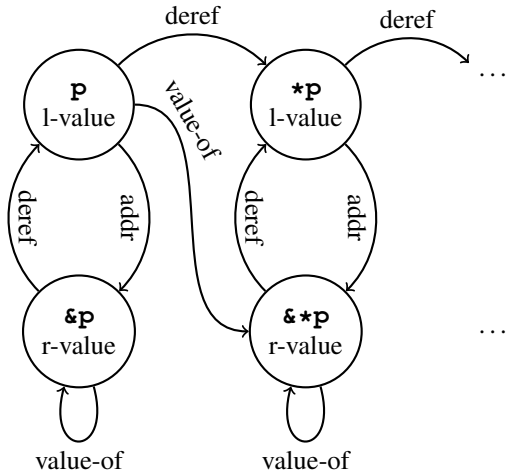


Figure 2. Pointer Operations in C

B. Heap Graphs

PtrTracker performs an abstract interpretation over the structure of the *heap*, where the underlying abstract domain is a *heap graph*. An example is depicted in Figure 3. Each node of the heap graph represents a contiguous chunk of memory (an “object” in C99 terms). Moreover, each node is made up of one or more memory cells, i.e. the memory node represents a *struct*, with a memory cell for each field. If the cell is of a pointer type (indicated by the bullets) it has one or more out-going edges, representing the memory cells it is possibly pointing to.

Note, pointers can point inside *structs*, `i` points to `a->c`. Non-*struct* objects, such as an `int*` pointer, are equivalent to a *struct* with a single field and are covered by the previous representation.

The abstract transformers for the heap graph domain are expressed in terms of the basic operations described in subsection III-A. Abstract transformers are currently implemented for C and a large part of C++.

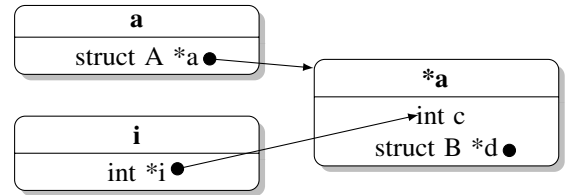


Figure 3. A heap graph

Assignments are performed as a *strong update* if we only have one l-value. In this case, we remove the existing arrow(s) and add arrows to the r-values. If, on the other hand, we have multiple l-values, we perform a *weak update* adding new arrows to all r-values.

C. Inter-procedural Analysis

Goanna itself supports inter-procedural analysis. Because functions can be analyzed in any order (Goanna also features concurrent/multi-core analysis) PtrTracker assumes as a minimal base:

- A1 *deep no-aliasing*: function parameters do not alias each other, and following parameter pointers also

do not lead to aliases to other parameters. As an example, consider the function `void foo(struct bar *a, struct bar *b);` we assume that `a` and `b` do not alias when we enter the function; the same holds for any pointer field `f a->f != b->f` for any sequence of dereferences.

- A2 *No-aliasing on global variables*: Parameters do not alias global variables.
- A3 *Side-effect free function calls*: Function calls do not modify parameters and always return new memory cells.

These assumptions are in general unsound, but they are essential for the pointer analysis to derive practical information. For instance without A3, if we analyze a function f that calls g and g has not yet been analyzed, we cannot assume anything of the (pointer) variables that are passed to g , nor of the global variables. This implies, after the call to g , we have to assume that two pointers can alias each other or a global variable. Note, if during the course of the analysis we discover that two parameters of f can alias each other, we can trigger a re-analysis of f under the revised assumption collecting *contextual* knowledge for the behavior of f . This feature will be available soon in PtrTracker.

IV. INTEGRATING PTRTRACKER AND GOANNA

The goal of introducing a pointer analysis in Goanna is to eliminate both false positives and false negatives. An essential part is that the analysis can integrate such that the key features of the tool are preserved: performance, ability to run as a compiler drop-in, re-use of existing checks and applicability to real-world programs. To be useful for bug-hunting the pointer analysis needs to be flow-sensitive, because flow-insensitive information would mask any manipulation done to pointers — which is counter-productive if one wants to detect bugs in pointer manipulation.

A. Implementation Details

There is a number of implementation details in PtrTracker which differs from a standard abstract interpretation or shape analysis, because of the special setting. Firstly, the analysis does *not* compute a fixed-point because this is unnecessary. In shape analysis summary nodes would be used to ensure termination (due to recursive data structures such as linked lists), but because we are more interested in must-information summary nodes are of little value.

Additionally, any syntactic element that does not point to any one unique memory location is also of little interest since this will disallow strong updates. Consider the C fragment:

```
struct list *a;
struct list *cur;
for (cur=a; cur!=NULL; cur=cur->next) {
    foo(cur, a);
}
```

Here the best information we can derive for the call to `foo` is that `cur` points to some element of the list starting at `a`.

However, we can derive that `a` always points to the same memory cell. Because of these observations, it is sufficient to recurse into recursive data structures until enough information has been gained such that loop invariants have been learned, and verified to be invariants.

The analysis is implemented in a lazy way, such that memory nodes are only added to the heap graph if needed. As such a `struct` with many members is only represented by the members referenced in the function currently being analyzed. Because the analysis computes information for each program point there is a lot of redundancy in the analysis result: we exploit this redundancy by storing the memory nodes in a flow-insensitive way, and only storing the edges (“points-to” information) of the graph for each program point. Both of these optimizations are essential for the performance of the analysis.

B. Arrays and Pointer Arithmetic

In C99, pointer arithmetic can only be used to move a pointer within a chunk of memory. That is, dereferencing a pointer outside “its” object is an error and should ideally be detected. In our analysis, we mark all nodes which are accessed via pointers which were subject to pointer arithmetic as containing arrays. Here, the index operator `[]` is handled as a special case of pointer arithmetic as described in Sect. III-A. Afterwards, we assume that the pointer aliases the complete array represented by the node. Hence, we can ignore pointer arithmetic altogether.

In a subsequent analysis, however, all pointer offsets can be recalculated using an integer analysis taking the pointer arithmetic into account. If we have must-alias information, we can infer tight bounds on the stride of memory that is accessed using this pointer.

C. Re-Using Existing Checks

To a large extent the existing Goanna checks were re-used without modification. In a few places some patterns were generalized to not look for explicit variable references, but also looking at other AST nodes that were annotated with a binder.

Some checks were rewritten to gain additional accuracy by taking pointer information into account: freeing allocated memory twice, losing the last reference to allocated memory, returning pointers to the local stack, and dereferencing a pointer that is possibly NULL. These rewritten checks improved both the rate of false positives and false negatives, compared to the old checks they replaced.

D. Path-Sensitive Checks

Our analysis determines alias information in a flow-sensitive way. This information, however, is not directly used to generate warnings. Instead, we employ a model-checker to verify certain properties of the program. This method can, in principle, recover the path-sensitivity for certain checks. Take, e.g., a possible double-free, which manifests itself as two calls to the free function where the argument points to the same object in both cases. In this case, our model-checker will determine a path containing two successive calls to the free function.

Instead of directly reporting this as an error, we first check whether this path is actually feasible. If the path is not feasible (e. g., it depends on two consecutive if-statements with contrary conditions), the false positive elimination module [12] can eliminate it and refine the CFG. When we analyze the refined CFG we effectively regain the path-sensitive information.

V. CASE STUDY

We have tried our prototype on a number of open source projects, to make sure that it supports a sufficient portion of C to be useful. We here report on running Goanna+PtrTracker on the well-known Git project¹.

The runtime of the stock Goanna on a single core without PtrTracker was 26m52s, which produced 266 warnings. Enabling PtrTracker slightly increased the runtime to 27m35s (an increase of 2.6%), while producing 271 warnings. Because Goanna is a drop-in replacement for the compiler in the build system both run times include the overhead of the actual compilation.

The 5 additional warnings are classified as follows:

- 1 true positive about the address of a local being stored in a global variable.
- 1 true positive about an arithmetic shift resulting in undefined behaviour, see Figure 4, found by an existing check being able to reason about pointers as additional variables.
- 1 true positive about a NULL dereference, involving interprocedural reasoning.
- 2 probably false positive about out of bounds array accesses using enum variables.

```

struct histindex index;
int sz;
...
index.table_bits = xdl_hashbits(count1);
sz = index.records_size = 1 <<
    index.table_bits;
...

```

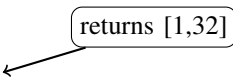


Figure 4. Example of bug found in Git: “interval [1, 32] is out of range of the shift operator”

VI. CONCLUSION & LESSONS LEARNED

We have integrated a new pointer analysis tool, PtrTracker, in the industrial-grade static analyzer Goanna. PtrTracker is a very pragmatic pointer analysis that satisfies the requirements of being very scalable, generally applicable and easily integratable. The analysis and design choices are interesting from a practical perspective: it gives an alternative to more traditional pointer-analyses in terms of usefulness for bug finding. While designing and implementing PtrTracker, we learned the following lessons:

- Real-world C programs contain complex pointer expressions (sometimes arising from pre-processor inlining) which must be handled compositionally. We were able

to reduce all expressions to the operation depicted in Fig. 2, which is key to the design of our analysis.

- A pragmatic approach to handle parameters and function calls is necessary (our assumptions A1–A3). Otherwise, a pointer analysis will likely generate no helpful results at all (i.e., aliasing everything) resulting in too many false positives.
- A multi-pass analysis and contextual analysis can be used to refine/invalidate assumptions, propagating alias information across function calls.
- Must-alias information, i.e. pointer dereferences with a fixed target, is much more helpful for bug-detection than may-alias information, and effectively rules out false positives.
- Pointer analysis is a complex and sometimes delicate topic. We keep the pointer analysis separate, and provide a set of possible variables for each pointer dereference, which helps tremendously in reducing the complexity.

REFERENCES

- [1] M. Hind, “Pointer analysis: haven’t we solved this problem yet?” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.
- [2] U. Khedker, A. Mycroft, and P. Rawat, “Liveness-Based Pointer Analysis,” in *Static Analysis Symposium*. Springer, 2012, pp. 265–282.
- [3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.
- [4] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer, “Goanna: Syntactic Software Model Checking,” *Automated Technology for Verification and Analysis*, pp. 216–221, 2008.
- [5] B. Hardekopf and C. Lin, “The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code,” in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 290–299.
- [6] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, “A novel analysis space for pointer analysis and its application for bug finding,” *Science of Computer Programming*, vol. 75, no. 11, p. 921, 2010.
- [7] V. B. Livshits and M. S. Lam, “Tracking pointers with path and context sensitivity for bug detection in C programs,” in *ESEC / SIGSOFT FSE*, 2003, pp. 317–326.
- [8] J. Brauer, R. Huuck, and B. Schlich, “Interprocedural Pointer Analysis in Goanna,” *Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 65–83, 2009.
- [9] R. Wilhelm, M. Sagiv, and T. Reps, “Shape Analysis,” in *Compiler Construction*, D. Watt, Ed. Springer Berlin Heidelberg, 2000, vol. 1781, pp. 1–17.
- [10] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O’Hearn, “Scalable Shape Analysis for Systems Code,” in *Computer Aided Verification (CAV)*, 2008, pp. 385–398.
- [11] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, and R. Huuck, “High Performance Static Analysis for Industry,” *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 3–14, 2012.
- [12] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, “SMT-based false positive elimination in static program analysis,” in *International Conference on Formal Engineering Methods (ICFEM)*, 2012, pp. 316–331.

¹<http://git-scm.com>