Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 254 (2009) 65–83
www.elsevier.com/locate/entcs

# Interprocedural Pointer Analysis in Goanna

Jörg Brauer[b,2] Ralf Huuck[a] Bastian Schlich[b,2]

[a] *National ICT Australia Ltd. (NICTA)*
*Locked Bag 6016, University of New South Wales*
*Sydney NSW 1466, Australia*[1]

[b] *Embedded Software Laboratory*
*Ahornstr. 55, RWTH Aachen University*
*52074 Aachen, Germany*

**Abstract**

Goanna is an industrial-strength static analysis tool used in academia and industry alike to find bugs in C/C++ programs. Unlike existing approaches, Goanna uses the off-the-shelf model checker NuSMV as its core analysis engine on a syntactic flow-sensitive program abstraction. The CTL-based model checking approach enables a high degree of flexibility in writing checks and scales to large code bases. In this paper, a new approach to pointer analysis for C is described. It is detailed how this technique is integrated into the model checking approach in order to perform interprocedural analysis. The performance and precision of this approach are demonstrated using a case study.

*Keywords:* Static Analysis, Interprocedural Analysis, Model Checking, Pointer Analysis

## 1 Introduction

Automatic tools support software developers in detecting bugs as early as possible in the development process, and thus, help minimizing cost of development and testing. Static analysis [19] tools identify syntactically correct but semantically incorrect programs without executing or simulating the analyzed program. Static analyzers typically do not guarantee the absence of

[1] National ICT Australia is funded by the Australian Governments Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.
[2] This work was carried out while being on leave at NICTA.


doi:10.1016/j.entcs.2009.09.060

defects. In recent years, much effort has been put into the development of static analyzers and some tools have become widely used in industry [11].

Model checking [6] is an automatic technique used in the field of formal verification, which allows verifying specifications for a given system by exploring its complete state space. Goanna [16] is a static analyzer for C and C++ programs, which differs from other static analyzers in that it uses model checking techniques to perform static analysis [12,21,20]. Using model checking to conduct static analysis allows a straightforward specification of desired program properties in Computation Tree Logic (CTL) [2]. Furthermore, if a specification is violated, a counterexample leading to the error is automatically generated, which is a valuable support for locating and fixing the defect.

Many static analyzers perform intraprocedural analyses without taking the effects of procedure invocations into account. Even though many defects can be found using intraprocedural analyses, other failures occur through the use of procedures in the wrong program context. A procedure may expose a correct local behavior but may still lead to a false program execution due to unexpected input values, for example. This paper describes an approach to interprocedural analysis of pointers for C based procedure summaries. Using procedure summaries allows to capture the influence of a procedure call on the program state and reuse these results whenever the corresponding procedure is called. In our approach, summaries are computed based on an intraprocedural pointer analysis.

## 2 Program Analysis in Goanna

Goanna is a static analyzer for C and C++ programs. This section first describes the general approach of Goanna, before the construction of models and the translation into the input language of NuSMV are detailed. NuSMV is used as the core analysis engine. A more thorough description of Goanna and the underlying intraprocedural analysis framework is given by Fehnker et al. [12].

### *2.1 Overview*

The basic idea of the approach implemented in Goanna is to map a C/C++ program to its control flow graph (CFG) and to automatically label nodes in the CFG with syntactic constructs of interest such as declarations of variables. The CFG together with the labels can be seen as a Kripke structure [6], which can be easily mapped to the input language of a model checker. Goanna uses an interval constraint solving approach based on the work of Gawlitza and Seidl [14] to detect buffer overruns. This approach is also used to per-

form false path elimination [13], a technique related to counterexample guided abstraction refinement [5].

### 2.2 Model Construction

An abstract syntax tree (AST) of a C/C++ function over alphabets of attributes $\Sigma_L$, $\Sigma_E$ can be seen as an attributed tree $(L, E, \mu_L, \mu_E)$ with nodes $L$, edges $E$, and labeling functions $\mu_L : L \to \Sigma_L$ and $\mu_E : E \to \Sigma_E$. The labeling functions assign attributes to nodes and edges, respectively. Nodes are labeled with program statements and expressions, while edges are attributed with the role of a branch. For instance, the edges leaving an `if-then-else` statement are labeled with *then* and *else*, or the edges leaving a node representing a binary operator are labeled with *rhs* and *lhs* to indicate right-hand side and left-hand side operands.

```
1  int fibonacci(int n) {
2    int x = 0, y = 1, q, i = 0;
3    do {
4      int oldy = y;
5      y = x;
6      q = x + oldy;
7      x = q;
8      i++;
9    } while (i < n);
10   return q;
11 }
```

Fig. 1. Example C program.

From an AST, a CFG can be constructed in a straightfordward manner. A CFG is a directed graph with a single root node. Note that a CFG does not contain all information available in the AST, only the control structure down to the level of statements is contained. No information about expressions or types is present. Given the set of atomic propositions $AP$, a triple $(L_f, E_f, \mu_f)$ with nodes $L_f$ representing statements, an edge relation $E_f \subseteq L_f \times L_f$, and an additional labeling function $\mu_f : L_f \to 2^{AP}$ defines a CFG. The labeling function $\mu_f$ defines the set of atomic propositions holding in a program location.

The CFG of the function `fibonacci()` from Fig. 1 is depicted in Fig. 2. The node labels in the CFG correspond to line numbers. The CFG is annoted with atomic propositions for the use of variable `q`. The variable `q` is declared

in line 2 and then assigned a value in line 6. Its value is read in lines 7 and 10.

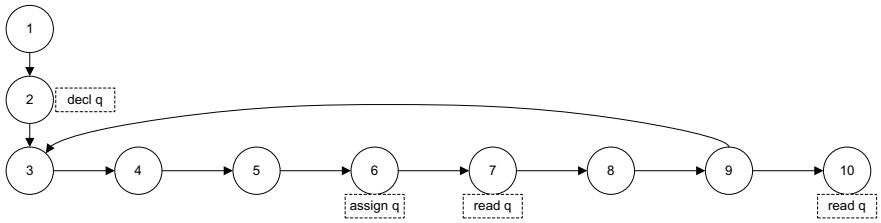


Fig. 2. CFG of the function `fibonacci()` in Fig. 1.

### *2.3 Translation to* NuSMV

In order to automatically check the generated model with respect to defined properties, it is translated into a NuSMV model. For a C/C++ function $f$, we translate the corresponding labeled CFG $(L_f, E_f, \mu_f)$ and a set of specifications $CTL_f$ over $AP$ into a NuSMV model $NuSMV_f = (var_f, \Delta_f, Def_f, CTL_f^{\text{NuSMV}})$, where:

- $var_f$ is an enumerated type variable over the set of nodes $L_f$.
- $\Delta_f \subseteq L_f \times L_f$ is the transition relation defined as $\Delta_f := \{(l, succ(l)) | l \in L_f \wedge succ(l) = \{l' | (l, l') \in E_f\}\}$. The target of each transition is the set of reachable locations.
- $Def_f = \{define(p) = \{l | \mu_f(l) = p \wedge l \in L_f\} | p \in \Sigma_f\}$, where every $define(p)$ is a `DEFINE` declaration, which indicates that an atomic proposition $p \in AP$ holds in a particular set of locations.
- $CTL_f^{\text{NuSMV}}$ is the set of CTL specifications $CTL_f$ in NuSMV syntax.

This model is automatically generated and used as the input for NuSMV. If one of the specifications in $CTL_f^{\text{NuSMV}}$ is violated, a counterexample trace is generated and mapped to the corresponding C statements, which helps users to understand how the defect emerged. Consider the CTL specification $\mathsf{AG}\ (\mathsf{decl_q} \Rightarrow (\mathsf{A}\ \neg \mathsf{read_q}\ \mathsf{W}\ \mathsf{write_q}))$. This formula expresses that whenever variable q has been declared, no path exists such that q is read before it is assigned a value. The CFG in Fig. 2 satisfies this formula.

## 3 Intraprocedural Pointer Analysis

This section details our approach to intraprocedural pointer analysis for C, which serves as a basis for both the generation of procedure summaries and

the actual detection of bugs (see Sect. 4). This is achieved by augmenting the NuSMV model with additional interprocedural information.

In presence of aliasing, a memory location can be accessed through different variables. In combination with structured types, this leads to complex problems to be tackled when analyzing pointer dependencies. Our underlying memory model abstracts from the physical representation to a coarse symbolic model but does not handle complex pointer arithmetic. The presented algorithm computes for each variable and each program location the sets of memory locations a variable can alias, its so-called points-to sets. The approach comprises the following steps:

(i) An intraprocedural abstraction of the physical memory is generated from the analyzed procedure.

(ii) Based on this representation, a static memory ownership model is established.

(iii) An equation system that represents dependencies between pointers in all program locations is generated.

(iv) A reduction pass that resolves aliasing dependencies is conducted.

(v) The least solution of the reduced equation system is computed.

These steps are resembled by the structure of this section and detailed in the following. In the end, an example is presented. Given a procedure $f$, the following notations are introduced: Let $L^f$ be the set of all program statements in $f$, and let $\mathbb{V}^f$ be the set of all variables under the scope of $f$. For the following computations, each procedure is converted into canonical static single assignment form [7]. This means, a unique instance of each variable is introduced for each program location. In this representation, a variable is a pair $(v, l) \in \mathbb{V}^f \times L^f$, which we denote by $v_l$. We denote the initial statement in $L^f$ by 0, that is, the initial values of $v$ under the scope of a procedure are denoted by $v_0$.

### *3.1 Abstract Memory Model*

In the memory abstraction used, memory locations are represented by symbolic values, which are induced through a number of constructs in C. A call of `malloc()`, for instance, allocates memory and returns an address, or memory locations may be introduced by parameters. Two special purpose labels are introduced: one for the address `NULL`, which is typically assigned to uninitialized pointers, and $\star$ to denote addresses resulting from operations not modeled in our abstraction. Nested pointers such as `int**` are currently not supported. The set of all memory locations in a procedure $f$ is denoted by $\mathbb{M}^f$. The

number of labels in $f$ is finite and the corresponding powerset $2^{\mathbb{M}^f}$ forms a complete lattice with the common operations on sets, $\bot = \emptyset$, and $\top = \mathbb{M}^f$. Furthermore, we define $\mathbb{M}^f_+ := \mathbb{M}^f \setminus \{\star, \texttt{NULL}\}$.

### *3.2 Memory Ownership*

A static ownership model for each symbolic memory value is computed. This means, for every memory label in $\mathbb{M}^f_+$ it is detected which variable it is first assigned to. This model is used both for the generation of intraprocedural alias information and interprocedural summaries (cp. Sect. 4). A variable $v_l$ *owns* a memory location $m \in \mathbb{M}^f_+$ iff $v$ is assigned $m$ in $l$, and $l$ is the first occurence of $m$. The ownership relation defines a mapping $\theta : \mathbb{M}^f_+ \to \mathbb{V}^f \times L^f$, which is used to distribute effects of operations on memory locations to all corresponding aliases in the following steps.

### *3.3 Encoding of Memory Aliasing*

In data flow analysis, a standard approach to express and to solve relations of variables in different program locations is to encode these in terms of equation systems [19], which allows to resolve cyclic dependencies. In case of pointer analysis, a variable represents a subset of $\mathbb{M}^f$. For each variable $v \in \mathbb{V}^f$ and each program location $l \in L^f$, we introduce an equation. These equations have one of the following two forms:

- $v_l \cong x$: Here, $x$ is a variable or a memory location, $x$ is assigned to variable $v$ in program location $l$.
- $v_l \cong \{\phi(v_k) | k \text{ is predecessor of } l\}$: This equation unites values of $v$ coming from predecessors of $l$ in the CFG if $v_l$ is not changed in $l$. The function $\phi : \mathbb{V}^f \to \mathbb{V}^f$ transforms the incoming values according to the encoding described in the following. That is, it maps variables to variables in order to handle aliasing.

In case a complex expression is assigned to $v_l$, we set $v_l \cong \star$. That is, the lattice element representing *unknown* is assigned to $v_l$. Intuitively speaking, the equation system assigns to each $v_l$ either a value assigned in $l$ or propagates incoming information along the CFG if $v_l$ is not changed. The challenge with modeling aliasing in structures is that assignments to a field of a structure influence its own children and the children of all respective aliases. To model this, we cover four different situations during the generation of an equation for a variable $v \in V^f$ in a program location $l \in L^f$.

(i) For $v \in V^f$ such that $v_0 \neq \theta(m)$ for all $m \in \mathbb{M}^f$, set $v_0 \cong \bot$. That is, all variables that are not initialized in the first program location, for

instance, through a function parameter, are set to $\perp$.

(ii) There exists $m \in \mathbb{M}^f$ such that $v_l = \theta(m)$. In this case, the equation $v_l \cong m$ is generated. This is the case if a fresh memory label is introduced at location $l$ and assigned to variable $v$, for instance by a C statement such as `int *q = (int*)malloc(sizeof(int))`.

(iii) $v_l$ is an alias created in the C program in location $l$, for instance, through an assignment of the form `int *v = w`. The value assigned to $v$ is the value of $w$ coming from a predecessor of $l$ in the CFG. The source variable $w$ in the assignment is marked as a reference, denoted by $v_l \cong \overrightarrow{w_l}$. The explicit notation of references is used to denote variables that alias a memory location owned by another variable. References are used to track transitive dependencies between owning and non-owning variables. If $v$ is of structured type, the same procedure is applied to all children of $v$. That is, we set $v\texttt{->}p_l \cong \overrightarrow{w\texttt{->}p_l}$. Otherwise, assignments made to children of a non-owning alias would not affect the children of the owning alias.

(iv) If the value of $v_l$ is not changed by the statement in $l$, then we set $v_l \cong \{\phi(v_k) | k \text{ is a predecessor of } l\}$ where $\phi$ replaces all references $\overrightarrow{w_{k'}}$ in $v_k$ with updated references $\overrightarrow{w_l}$, which expresses that $v_l$ corresponds to $w_l$. If there exists $l' \in L^f$ and $m \in \mathbb{M}^f$ such that $v_{l'} = \theta(m)$, then $v_k$ is replaced with $m$. The other values remain unchanged by $\phi$.

Step (iv) transforms references in incoming variables to express that the references in the equation correspond to the variables in the current program location $l$. References – denoted by $\overrightarrow{w_l}$ – are a syntactic means used to model that $v$ corresponds to $w$ in the same program location. If the equations are generated following the rules described above, a unidirectional syntactic dependency between owners and non-owners of a memory location is established in the equation system. As these rules explicitly tackle aliasing, they are only applied to pointer variables.

### *3.4 Reduction*

During the reduction phase, all references in the equation system are eliminated to perform a unification of owning and non-owning aliases. The modeling of aliasing in the previous section has introduced references on the right-hand sides of the equation system such as, for instance, $v_l \cong \overrightarrow{w_l}$. These references were introduced to track aliasing dependencies.

In the reduction step, references are resolved to distribute information from actions applied to non-owning variables to other aliases. Without this step, the effects would only be visible to the owning variable. As a result, all aliases correspond to the same memory labels. If we write $v\texttt{->}v'$ in a structure, then

$v$ and $v'$ may be structured variables themselves. Using this notation, the described reduction can easily be applied to nested structures. The equation system is reduced by applying the following actions:

**Phase 1** The first phase reduces references in structures. If $v\texttt{->}v_l^{'}$ owns a memory location $m \in \mathbb{M}^f$ and the direct predecessor $v_{l^-}$ of $v_l$ contains a reference $\overrightarrow{w_{l^-}}$, then the following four actions are conducted:

(i) The existing equation for $w\texttt{->}v_l^{'}$ is removed from the equation system.

(ii) The value $m$ is assigned to $w\texttt{->}v_l^{'}$, all variables referenced in the original equation of $w\texttt{->}v_l^{'}$, and all variables referencing $w\texttt{->}v_l^{'}$. This operation can be implemented efficiently by storing all references while the equation system is generated in the first place.

(iii) The reference $\overrightarrow{w_{l^-}}$ in the equation of $v_{l^-}$ is replaced by $w_{l^-}$.

(iv) For all attributes $w'$ of the structure $w$, we set $w\texttt{->}w_l' = \bot$.

These steps are repeated for all variables owning memory locations. If $v_{l^-}$ contains a reference $\overrightarrow{w_{l^-}}$, this means that in a statement preceeding the current statement, the variable $v_{l^-}$ is an alias of a memory location owned by the variable $w$ in some program location.

**Phase 2** All references that still exist in the equation systems are replaced with the corresponding values, for instance, a reference $\overrightarrow{w_l}$ is replaced with $w_l$.

In the resulting equation system, all dependencies caused by aliasing between structures are resolved and replaced by assignments of values.

### *3.5 Resolving Dependencies*

The points-to sets are then generated by computing the least fixed point of the equation system, which is implemented using the worklist algorithm [19]. The least fixed point of the equation system defines the points-to set of each variable in each program location. The existence of least fixed point is guaranteed by the finiteness of $\mathbb{M}^f$ and the monotonicity of the operations.

### *3.6 Example*

This section describes the intraprocedural pointer analysis for the example function `perform()` depicted in Fig. 3. Here, memory for two structures of types `dev_t` and `cont_t` is allocated. An alias `q` for the field `s->p` is created, and using this alias, memory for the field `q->v` is allocated. That means, that `q->v` and `s->p->v` alias the same memory location.

The structure of the stack and the heap as well as the points-to dependencies of this program after execution of the corresponding line numbers is

```
1 struct cont_t { int *v; };
2 struct dev_t { struct cont_t *p };
3
4 void perform () {
5   struct dev_t *s = (struct dev_t*)malloc(sizeof(dev_t));
6   s->p = (struct cont_t*)malloc(sizeof(cont_t));
7   struct cont_t *q = s->p ;
8   q->v = (int*)malloc(sizeof(int));
9 }
```

Fig. 3. Example program for nested aliasing of structures in C.

depicted in Fig. 4.

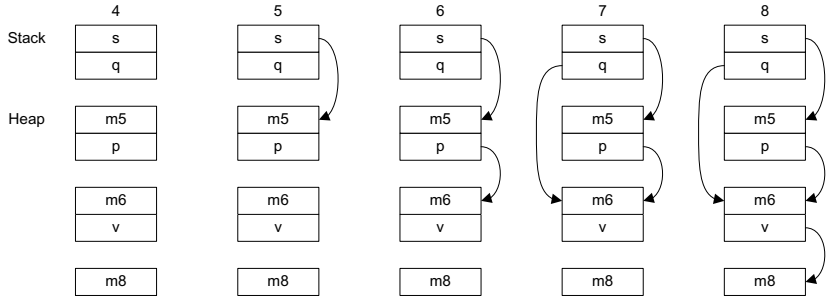


Fig. 4. Structure of stack and heap during execution `perform()`.

Encoding the nested aliasing using the algorithm described in Sect. 3.3 leads to the equation system depicted in Fig. 5 for the procedure `perform()`. For clarity, variables as well as memory locations introduced in a statement are annotated with their corresponding line numbers. The equation $\mathtt{s}_5 \cong m_5$ is generated by applying condition (ii) from Sect. 3.3. In contrast, the updated reference in the equation $\mathtt{q}_8 \cong \overrightarrow{\mathtt{s->p}_8}$ is generated due to the application of condition (iv). Note, that at this stage no connection between `s->p->v` and `q->v` exists, and hence, no information can be passed from `q->v` to `s->p->v`. This implies that the allocation of memory using `q->v` is not passed to `s->p->v`, although they alias the same memory location.

Reducing the equation system shown in Fig. 5 produces the equation system given in Fig. 6. The equations for the variables `s` and `s->p` remain unchanged by the reduction algorithm, and hence, they are omitted here. The reference in the equation $\mathtt{q->v}_7 \cong \overrightarrow{\mathtt{s->p->v}_7}$ causes the memory location $m_8$ to be passed to $\mathtt{s->p->v}_8$. The least fixed point at the end of the analyzed procedure `perform()` produces the exact points-to sets, namely $\mathtt{s}_8 = \{m_5\}$, $\mathtt{s->p}_8 = \{m_6\}$, $\mathtt{s->p->v}_8 = \{m_8\}$, $\mathtt{q}_8 = \{m_6\}$, and $\mathtt{q->v}_8 = \{m_8\}$.

| | | | | |
|---|---|---|---|---|
| $\mathtt{s}_4 \cong \bot$ | $\mathtt{s->p}_4 \cong \bot$ | $\mathtt{s->p->v}_4 \cong \bot$ | $\mathtt{q}_4 \cong \bot$ | $\mathtt{q->v}_4 \cong \bot$ |
| $\mathbf{s_5 \cong m_5}$ | $\mathtt{s->p}_5 \cong \mathtt{s->p}_4$ | $\mathtt{s->p->v}_5 \cong \mathtt{s->p->v}_4$ | $\mathtt{q}_5 \cong \mathtt{q}_4$ | $\mathtt{q->v}_5 \cong \mathtt{q->v}_4$ |
| $\mathtt{s}_6 \cong \mathtt{s}_5$ | $\mathtt{s->p}_6 \cong m_6$ | $\mathtt{s->p->v}_6 \cong \mathtt{s->p->v}_5$ | $\mathtt{q}_6 \cong \mathtt{q}_5$ | $\mathtt{q->v}_6 \cong \mathtt{q->v}_5$ |
| $\mathtt{s}_7 \cong \mathtt{s}_6$ | $\mathtt{s->p}_7 \cong \mathtt{s->p}_6$ | $\mathtt{s->p->v}_7 \cong \mathtt{s->p->v}_6$ | $\mathtt{q}_7 \cong \overrightarrow{\mathtt{s->p}_7}$ | $\mathtt{q->v}_7 \cong \overrightarrow{\mathtt{s->p->v}_7}$ |
| $\mathtt{s}_8 \cong \mathtt{s}_7$ | $\mathtt{s->p}_8 \cong \mathtt{s->p}_7$ | $\mathtt{s->p->v}_8 \cong \mathtt{s->p->v}_7$ | $\mathbf{q_8 \cong \overrightarrow{s->p}_8}$ | $\mathtt{q->v}_8 \cong m_8$ |

Fig. 5. Equation system generated from procedure `perform()` in Fig. 3 before reduction.

| | | |
|---|---|---|
| $\mathtt{s->p->v}_4 \cong \bot$ | $\mathtt{q}_4 \cong \bot$ | $\mathtt{q->v}_4 \cong \bot$ |
| $\mathtt{s->p->v}_5 \cong \mathtt{s->p->v}_4$ | $\mathtt{q}_5 \cong \mathtt{q}_4$ | $\mathtt{q->v}_5 \cong \mathtt{q->v}_4$ |
| $\mathtt{s->p->v}_6 \cong \mathtt{s->p->v}_5$ | $\mathtt{q}_6 \cong \mathtt{q}_5$ | $\mathtt{q->v}_6 \cong \mathtt{q->v}_5$ |
| $\mathtt{s->p->v}_7 \cong \mathtt{s->p->v}_6$ | $\mathtt{q}_7 \cong \mathtt{s->p}_7$ | $\mathbf{q->v_7 \cong s->p->v_7}$ |
| $\mathbf{s->p->v_8 \cong m_8}$ | $\mathtt{q}_8 \cong \mathtt{s->p}_8$ | $\mathtt{q->v}_8 \cong m_8$ |

Fig. 6. Reduced equation system generated from Fig. 5.

# 4 Interprocedural Analysis

This section details how to capture the behavior of functions with respect to pointer analysis using procedure summaries based on the intraprocedural points-to sets. Moreover, it describes our approach of combining aliasing information with procedure summaries in the intraprocedural analysis framework of Goanna. This allows us to detect invalid memory accesses that result from procedure calls by extending the existing intraprocedural model described in Sect. 2.

## *4.1 Procedure Summaries*

In Goanna, procedure summaries are represented as sets of variables. Each procedure summary describes a single property of interest and contains variables for which the respective property is fulfilled after termination of the procedure. Due to the call-by-value semantics of C, three kinds of summaries are required to detect invalid memory accesses across boundaries of procedure scopes. These summaries for a function $f$ state whether a parameter is dereferenced, validated, or invalidated.

An invalid pointer directly passed to a function cannot be validated by the callee because a copy of the aliased address is passed and not the address of

the pointer itself. This is only true in the absence of nested pointers such as `int**`. In contrast, uninitialized fields in a structure can be validated because their addresses may be accessed through the structure stored on the heap. The set of parameters does not only contain explicitly declared formal parameters, but also parameters hidden in structures. This is called a *transitive attribute closure*. Given a parameter $p$ in $f$ and $m \in \mathbb{M}^f$ with $p_0 := \theta(m)$, we have the following summaries:

**Memory Dereference** $p \in \mathcal{D}^f$ iff $m$ is dereferenced in $f$ or passed to a procedure called from $f$, where it is dereferenced.

**Memory Invalidation** $p \in \mathcal{I}^f$ iff $m$ is invalidated in $f$, for instance, by calling `free()`, or passed to a procedure $g$ called by $f$ and invalidated in $g$.

**Memory Validation** $p \notin \mathcal{V}^f$ if $p$ is of simple type for the reasons described before. That is, pointers passed by value cannot be validated in a called function. Memory for fields in structures, however, can be validated. Detection of validation is based on the intraprocedural points-to sets.

A call graph is a directed graph where each node represents one procedure in the analyzed program. It contains an edge $(f, g)$ if $g$ is called by $f$. Procedure summaries describe the behavior of $f$. Moreover, the summaries depend on all summaries of functions called from $f$. If the call graph is acyclic, the summaries are generated by visiting all procedures in reverse topological order. In contrast, recursive dependencies between procedures require computing the least fixed point in all strongly connected components.

### *4.2 Model Generation*

In this section, we describe how a NuSMV model for the interprocedural analysis of pointers is generated based on procedure summaries and the intraprocedural pointer analysis. The underlying model is an extension of the intraprocedural model given in Sect. 2.3. Given a procedure $f$ and its intraprocedural model $NuSMV^f = (var^f, \Delta^f, Def^f, CTL^f)$, the interprocedural model for pointer analysis is a quadruple $NuSMV^f_{PA} = (var^f_{PA}, \Delta^f_{PA}, Def^f_{PA}, CTL^f_{PA})$ consisting of:

**Variables** $var^f_{PA}$ For each memory location $m \in \mathbb{M}^f_+$, we introduce a state variable with possible values *valid* and *invalid* to track state changes during the execution of $f$. Note that memory allocations indicated by the validation summary $\mathcal{V}^f$ increase the size of $\mathbb{M}^f_+$, and hence, the number of state variables. The coarse memory abstraction leads to a symbolic memory address $\star$ for unknown memory locations. To avoid producing a multitude of spurious warnings, we exclude this label, which leads to an under-approximation.

Hence, it is $var^f_{PA} = var^f \cup \mathbb{M}^f_+$.

**Transitions** $\Delta^f_{PA}$ Each $m \in \mathbb{M}^f_+$ is initialized with *valid*. Furthermore, for each invocation of a function $g$ in a statement $l \in L^f$ to which $m$ is potentially passed, the effect of the function call is encoded based on the memory invalidation summary $\mathcal{I}^g$. Hence, we add a transition $(statement = l \wedge m = valid) \rightarrow m = invalid$ to the model if we have $m \in \mathcal{I}^g$. That is, if the corresponding summary states that $m$ may be invalidated by $g$, then the status of $m$ is changed to *invalid*.

**Labels** $Def^f_{PA}$ For each dereference of a memory location $m \in \mathbb{M}^f$ in a program location $l \in L^f$ based on the dereference summary, a label $deref_m$ is added to the state representing $l$.

**Specifications** $CTL^f_{PA}$ For each memory location $m \in \mathbb{M}^f_+$, we add an invariant specification $AG(deref_m \Rightarrow m = valid)$. That is, if a variable aliasing $m$ is derefenced, then $m$ is required to be valid.

In the memory abstraction used, memory locations are introduced through parameters or return values of functions such as `malloc()`. Pointers accessed in $f$ may be invalid in both cases: A programmer may pass an invalid pointer to $f$ or `malloc()` may fail. In the first case, the defect in $f$ is detected during the analysis of the caller of $f$. The second case is dealt with by an intraprocedural check that requires all allocated memory to be checked before it is dereferenced. To detect dereferences of `NULL` pointers, a specification $\mathsf{AG}(\neg deref_{\texttt{NULL}})$ is added.

The encoding of the pointer analysis in the NuSMV model could be conducted in a different manner. It would, for instance, also be possible to distribute the detected memory statuses to all program locations using a forward data flow analysis and label the corresponding locations with atomic propositions. In this case, an invariant specification of the form $\mathsf{AG}\neg(deref_m \wedge invalid_m)$ would be sufficient and state variables are not required.

### *4.3 Example*

This section describes the application of the analysis to parts of the Linux 2.6 kernel, namely the `sound` module. The code contains a defect caused by false pointer deallocation nested in two function calls. The procedure `snd_hwdep_release()` in Fig. 7 is implemented in `sound/core/hwdep.c`. In line 10, it calls `snd_card_file_remove()` (see Fig. 8) and passes `hw->card`, which is then dereferenced in line 12.

The function `snd_card_file_remove()` calls `snd_card_do_free()`, which

```
1 static int snd_hwdep_release(struct inode *inode,
2                               struct file *file) {
3   int err = -ENXIO;
4   struct snd_hwdep *hw = file->private_data;
5   mutex_lock(&hw->open_mutex);
6   if (hw->ops.release) {
7     err = hw->ops.release(hw, file);
8     hw->used--;
9   }
10  snd_card_file_remove(hw->card, file); // free
11  mutex_unlock(&hw->open_mutex);
12  module_put(hw->card->module);          // deref
13  return err;
14 }
```

Fig. 7. Function `snd_hwdep_release` from Linux 2.6 kernel.

invalidates the parameter `hw->card` by calling `kfree()`. This means, when `hw->card` is dereferenced in line 12 of `snd_hwdep_release()`, it has possibly been freed. The summaries generated for `snd_card_file_remove()` are:

$$\mathcal{D}^{\texttt{snd\_card\_file\_remove}} = \{\texttt{card}\}$$
$$\mathcal{I}^{\texttt{snd\_card\_file\_remove}} = \{\texttt{card}\}$$

In the model of `snd_hwdep_release()`, the variable `hw->card` is mapped to `card` in `snd_card_file_remove()` and the invalidation summary is applied. This means, the state of `hw->card` in the NuSMV model is set to *invalid* when the function `snd_card_file_remove()` is called. NuSMV reports a violation of the specification $\mathsf{AG}(\mathit{deref}_{\texttt{hw->card}} \Rightarrow \texttt{hw->card} = \mathit{valid})$ in line 12.

```
1 static int snd_card_do_free(struct snd_card *card,
2                              struct file *file) {
3   ...
4   if (last_close) {
5     wake_up(&card->shutdown_sleep);
6     if (card->free_on_last_close)
7       snd_card_do_free(card);
8   }
9 }
```

Fig. 8. Functions `snd_card_file_remove()` and `snd_card_do_free()` from Linux 2.6 kernel.

# 5 Case Study

We have evaluated the performance of the intraprocedural pointer analysis and the summary-based interprocedural analysis by analyzing some source directories of OpenSSL 0.9.8d. These directories contain between 1,633 and 28,916 lines of C code. This section first describes the applied checks and then presents the analysis results.

## *5.1 Evaluation Principles*

The hardware platform used for the experiments is a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4 GHz, 2 MiB L2 cache and 1.5 GiB DDR-2 400 Mhz ECC memory. We compare the performance of Goanna running with the following three configurations: (i) intraprocedural analysis with all standard checks enabled, but array bounds checking and false path elimination disabled, (ii) with intraprocedural pointer analysis, and (iii) with summary-based interprocedural analysis. The standard checks include checks for uninitialized variables, unused values, unreachable code and simple memory checks, which do not consider aliasing. For the other configurations, these checks are performed as well. Moreover, the runtimes also include the time required for parsing the respective programs. For completeness, we compare the results with the time needed by GCC required for compiling these source directories.

## *5.2 Analysis Performance*

The analysis performance is depicted in Tab. 1. The intraprocedural pointer analysis usually requires 3 to 6 times more time than the analysis with standard checks. The runtimes show that the interprocedural analysis scales with the code size. The modeling of implicit aliasing through the transitive attribute closure introduces large numbers of auxiliary variables in the equation system, and in consequence, slows down the pointer analysis.

About 90% of the slowdown caused by the pointer analysis is spent on the generation of the equation system. Fehnker et al. [12] already noticed that one particular performance bottleneck of Goanna is the currently used tree matching algorithm for the AST, which is based on an XML representation and XPath in order to detect statements and expressions of interest. This mechanism is extensively used when the equation system is generated and causes most of the slowdown. The fixed point iteration for computing the points-to sets themselves is barely noticeable in terms of runtime. The same applies for model checking the extended interprocedural model. During

Table 1
Analysis performance of Goanna for OpenSSL 0.9.8d.

| Directory | LoC | GCC | intra | intra+pointer | inter+pointer |
|---|---|---|---|---|---|
| `crypto/des` | 6,112 | 4.204 | 14.166 | 52.961 | 53.294 |
| `crypto/engine` | 4,991 | 3.618 | 11.253 | 80.449 | 82.841 |
| `crypto/pkcs12` | 1,633 | 1.476 | 4.237 | 17.673 | 18.934 |
| `engine` | 7,244 | 5.152 | 16.913 | 56.796 | 60.371 |
| `ssl` | 28,916 | 22.733 | 58.372 | 242.149 | 253.313 |

the evaluation, we found some files for which the poor performance results was caused by extensive use of preprocessor macros, which introduced large numbers of auxiliary variables. The analysis of the files in the `crypto/engine` directory, for instance, was slowed down by one file `eng_padlock.c`, on which 70% of the overall runtime was spent.

Despite these downsides, we showed that the developed approach can be successfully applied to large code bases. We have also applied the approach to parts of the Firefox codebase, for which the slowdown was also linear. Applying Goanna to the complete source code of Firefox did not produce meaningful results due to the extensive use of C++ features such as templates. The performance drawback could be minimized by optimizing the XPath algorithm, which would lead to competitive runtimes. In practice, runtimes can be significantly decreased using incremental analysis. Only those program fragments affected by a modified summary have to be reanalyzed. Typically, this involves only small parts of the program.

## 6 Related Work

In Steensgard's flow-insensitive algorithm based on unification [23], the pointer analysis problem is reduced to finding a well-typed environment using set constraints, which allows a points-to analysis in almost linear time. Andersen's subtyping-based approach [1] relies on constraint solving over inclusion constraints. The algorithm is slower than Steengard's approach, but produces more precise results. The approach by Das [8] is an extension of Steensgard's algorithm based on a restricted form of subtyping. Unification of symbols at top levels of pointer chains in the points-to graph is avoided. Shapiro and Horwitz [22] developed an extension of Steensgard's algorithm in which the points-to set of a variable is partitioned into multiple categories. All approaches described so far are flow-insensitive.

Flow-sensitive analyses produce more precise results than flow-insensitive approaches. In practice, however, Hind and Pioli [15] observed little benefit on most benchmarks. Nevertheless, a flow-sensitive approach promises a lower rate of spurious warnings. An interprocedural algorithm for conditional may-aliasing based on abstract interpretation was described by Landi and Ryder [17]. Choi et al. [4] presented an interprocedural algorithm, which combines flow-sensitive and -insensitive techniques. A similar approach was described by Emami et al. [10], who proposed a technique for context-sensitive analysis of stack-allocated data structures, which is specifically suitable in the presence of function pointers. In contrast, our approach focuses on heap-allocated data. An interprocedural analysis algorithm for recursive data structures was described by Deutsch [9]. Another context-sensitive points-to analysis, which uses partial transfer functions for procedure summaries, was developed by Wilson and Lam [24]. Incomplete transfer functions are used, which only cover input conditions that exist in a program. Their points-to analysis uses an iterative data flow approach to find potential pointer values. Cheng and Hwu [3] described an approach using accesses paths for interprocedural pointer analysis of C programs based on context-sensitive transfer functions. Access paths are used to distinguish non-recursive heap structures.

Closely related to our approach is the context-insensitive but flow-sensitive points-to analysis for Java programs described by Ma and Foster [18]. Their abstraction of physical memory using memory labels is similar to our approach. In contrast to our algorithm, their approach lacks support for nested data structures. While most approaches rely on equation solving in order to resolve points-to information, Ma and Foster use constraint solving.

## 7 Conclusion and Future Work

Many defects found in real software are related to false handling of pointers. This is especially true for languages such as C, which allow arbitrary pointer arithmetic and have no built-in mechanisms for pointer safety. It turned out that static analysis based on purely syntactic properties of a program allows an efficient analysis but also generates large numbers of false warnings. We have combined the syntactic analysis framework in Goanna with a memory abstraction. This paper describes an approach to interprocedural pointer analysis that is integrated into the intraprocedural analysis framework. This approach consists of a two-pass algorithm. First, an intraprocedural pointer analysis is performed, based on fixed point iteration over an equation system. The obtained intraprocedural points-to information is then integrated into a NuSMV model for interprocedural analysis, which is used by Goanna to

conduct static analysis.

The implementation smoothly integrates into the existing framework. The intraprocedural analysis alone improves the existing intraprocedural analysis framework. It integrates well with incremental analyses. The interprocedural NuSMV model is an extension of the original model. One of the advantages of using model checking for static analysis is the automatic generation of counterexamples. With interprocedural analysis using summaries, this cannot be achieved easily. During the analysis of a function, no information about the intrinsics of called functions is present. Hence, different techniques have to be developed to tackle this problem.

The pointer analysis is based on a coarse abstraction of the physical memory, which has proven to be powerful enough to detect bugs in real software. The discussed approach has some disadvantages. One obvious flaw of the current implementation is the assumption that parameters passed to a function are separated, that is, they do not alias the same memory locations. This can be fixed with minor extensions of the described algorithm. Only the results of the points-to analysis have to be updated, that is, aliasing of parameters can be expressed by replacing memory labels introduced through two parameters by a single one. The corresponding procedure does not have to be reanalyzed.

Another improvement would be to integrate of the interval solving techniques and the memory model to gain further precision. A challenge for static code checkers is to detect inobvious defects while not producing vast amounts of spurious warnings, which is sometimes contradictory. The average defect density in Goanna is around 0.3 to 2 bugs per 1,000 lines of code, which is comparable with commercial static analyzers [11]. The number of false warnings strongly depends on the analyzed code. Function pointers, for instance, often cause spurious warnings. Combining different techniques promises the detection of more classes of defects while at the same time reducing the number of false warnings.

## References

[1] Andersen, L. O., "Program Analysis and Specialization for the C Programming Language," Dissertation, DIKU, University of Copenhagen, Copenhagen, Denmark (1994).

[2] Ben-Ari, M., Z. Manna and A. Pnueli, *The temporal logic of branching time*, Acta Informatica **20** (1983), pp. 207–226.

[3] Cheng, B.-C. and W.-M. W. Hwu, *Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation*, in: *Programming Language Design and Implementation (PLDI 2000), Vancouver, Canada* (2000), pp. 57–69.

[4] Choi, J.-D., M. Burke and P. Carini, *Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects*, in: *Principles of Programming Languages (POPL 1993), Charleston, USA*, 1993.

[5] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement*, in: *12th International Conference on Computer Aided Verification (CAV 2000), Chicago, USA*, Lecture Notes in Computer Science **1855** (2000), pp. 154–169.

[6] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 1999.

[7] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Effciently computing static single assignment form and the control dependence graph*, ACM Transaction on Programming Languages and Systems (1991), pp. 451–590.

[8] Das, M., *Unification-based pointer analysis with directional assignments*, in: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), Vancouver, Canada* (2000), pp. 35–46.

[9] Deutsch, A., *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, in: *Programming Language Design and Implementation (PLDI 1994), New York, USA*, 1994, pp. 230–241.

[10] Emami, M., R. Ghiya and L. J. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, in: *Programming Language Design and Implementation (PLDI 1994), Orlando, USA* (1994), pp. 242–256.

[11] Emanuelsson, P. and U. Nilsson, *A comparative study of industrial static analysis tools*, in: *3rd International Workshop on Systems Software Verification (SSV 2008), Sydney, Australia*, Electronic Notes in Theoretical Computer Science **217** (2008), pp. 5–21.

[12] Fehnker, A., R. Huuck, P. Jayet, M. Lussenburg and F. Rauch, *Model checking software at compile-time*, in: *Theoretical Aspects of Software Engineering (TASE '07), Shanghai, China* (2007), pp. 45–56.

[13] Fehnker, A., R. Huuck and S. Seefried, *Counterexample guided path reduction for static program analysis*, in: *Correctness, Concurrency, Compositionality: Essays in honor of Willem-Paul de Roever*, Lecture Notes in Computer Science (2008).

[14] Gawlitza, T. and H. Seidl, *Precise fixpoint computation through strategy iteration*, in: *16th European Symposium on Programming (ESOP 2007), Braga, Portugal*, Lecture Notes in Computer Science **4421** (2007), pp. 300–315.

[15] Hind, M. and A. Pioli, *Assessing the effects of flow-sensitivity on pointer alias analyses*, in: *Symposium on Static Analysis (SAS 1998), Pisa, Italy*, Lecture Notes in Computer Science (1998), pp. 57–81.

[16] Huuck, R., A. Fehnker, S. Seefried and J. Brauer, *Goanna: Syntactic software model checking*, in: *Automated Technology for Verification and Analysis (ATVA 2008), Seoul, Korea*, Lecture Notes in Computer Science **5311** (2008), pp. 216–221.

[17] Landi, W. and B. G. Ryder, *A safe approximate algorithm for interprocedural pointer aliasing*, in: *Programming Language Design and Implementation (PLDI 1992), San Francisco, USA* (1992), pp. 473–489.

[18] Ma, K.-K. and J. S. Foster, *Inferring aliasing and encapsulation properties for Java*, in: *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007), Montreal, Canada* (2007), pp. 423–440.

[19] Nielson, F., H. R. Nielson and C. Hankin, "Principles of Program Analysis," Springer, 1999.

[20] Schmidt, D. A., *Data flow analysis is model checking of abstract interpretations*, in: *25th ACM Symposium on Principles of Programming Languages (POPL 1998), San Diego, USA* (1998), pp. 38–48.

[21] Schmidt, D. A. and B. Steffen, *Program analysis as model checking of abstract interpretations*, in: *5th International Symposium on Static Analysis (SAS 1998), Pisa, Italy*, Lecture Notes in Computer Science **1503** (1998), pp. 351–380.

[22] Shapiro, M. and S. Horwitz, *Fast and accurate flow-insensitive points-to analysis*, in: *24th Symposium on Principles of Programming Languages (POPL 1997), Paris, France* (1997), pp. 1–14.

[23] Steensgard, B., *Points-to analysis in almost linear time*, in: *23th ACM Symposium on Principles of Programm Language (POPL 1996), St. Petersburg Beach, USA* (1996), pp. 32–41.

[24] Wilson, R. P. and M. Lam, *Efficient context-sensitive pointer analysis for C programs*, in: *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1995), La Jolla, USA* (1995), pp. 1–12.