Alias Analysis and Calculus based on Segmentation Address Memory Model

Igor Parfenov
Innopolis University
Innopolis, Russia
parfenov_2001@mail.ru

***Abstract*—We present a straightforward implementation of a simplified imperative programming language with direct memory access and address arithmetic, and a simple static analyzer for memory leaks. Our study continues a line of research attempted (in Innopolis University in years 2016-2022) on alias calculi for imperative programming languages with decidable pointer arithmetic but differs by memory address model — we study segmented memory model instead linear one.**

***Index Terms*—Imperative programming, memory address model, memory safety, memory leaks, static analysis**

I. Introduction

There are various different instruments for program code development. One of the areas that has to be improved for programming languages is the safety and correctness of successfully compiled programs. The C programming language, like some others, has pointers and direct memory access, which is a powerful and, at the same time, uncontrollable instrument, whose safety depends only on the programmer. Those programming languages need some validation techniques for checking the safety and correctness of using those features.

The alias calculus is the mathematical model which operates on an abstract, simplified programming language with dynamic memory (accessible by explicit or implicit pointers) and some rules, using which it can validate if the program is memory safe. This theory can be expanded to real programming languages.

In this paper, firstly, a new variant of the alias calculus is suggested and studied up to some extent. Then, based on the this theory, we present a compiler for a simple model C-like programming language with direct memory access (via pointers). The compiler has been implemented from scratches. For this programming language, a set of programs has been written, and some metrics and statistics of their executions have been collected and studied. Finally, we present a static validator for memory leak safety for programs (with pointers) written in our model language. We hope that this instrument can be successfully used in real programming.

II. Literature Review

A. Anderson's Model

Andersen's pointer analysis model [1] is the most close to alias calculus among commonly used static analysis models. However, it describes a little different, more simplified, pointer-to model. Nevertheless, definitions and properties introduced there are necessary for this scope's analysis. Roughly speaking, Andersen' pointer analysis is based on theory of equality for uninterpreted functional symbols. The algorithm traverses the program by statements and calculates for every pointer set of other pointers, to which it can be equal. Such a set is called points-to set. During traversing, once an assignment is met, a constraint "point-to set of source is subset of point-to set of destination" is created. After the constraints are collected, they are solved. The content of the work is overcomplicated, though.

B. Alias Calculus

A simplified description of alias calculus and some other information out of this scope is described in [2]. Informally, alias relation is a structure, which specifies for every variable, to which pointer variable does it belong. The cited paper presents a set of simple operations: assignment, allocation and deallocation, if-statements and loops, which affect the alias relation. The purpose of calculus rules is static over-approximation Q of actual alias relation after execution of a program S for a given alias relation P before the program execution, i.e., such relation Q that Hoare triple $[P]S[Q]$ to be true. For example, assignment statement copies aliases to destination replaced by source and removes aliases, which contain source; if-statements calculate relations for all branches and unite them.

The algorithm from [2] was implemented in the Eiffel Verification Environment and can be used through the AutoProof module. The approach used in the algorithm, presented in our is, however, different and will be explained in detail in the corresponding section. One of the main differences is the memory model used: in [2] memory consists of abstract addresses while in our model, for every state and for every variable, alias relation describes the meta-identifier of allocated space and shift relative to the meta-identifier. This allows swapping allocated space for variables without triggering the validator.

C. Separation Logic

Separation logic [3] and [4] is an extension of the first-order logic for specification of the programs over dynamic memory (heap) in Hoare assertions $[P]S[Q]$. It operates on a heap, which is addressed, using a "separating conjunction" operator, which checks if objects hold different parts of the heap. There were proposed ways to handle unrestricted memory access with not only static arrays, but also dynamic arrays and recursive functions. The concept of separation logic is widely used in different proof assistants and frameworks; hence, it can also be used for validating programs in this scope. Separation logic semantics is based on a model comprising stores (to represent static memory) and heaps (to represent dynamic memory), which are finite-domain maps from variables' identifiers and, respectively, locations (accessible via pointers or addresses which are particular numbers), to data values (e.g. integers). There are two major heap models in use: linear or flat (where each location is capable of storing simple data values), and segmented (where locations can store compound data like arrays with static size).

D. MoRe Language

[5] presents and describes the MoRe language, which allows more flexible actions on pointers' addresses in comparison to Andersen's one. The cited paper describes the target theory in the clearest and most understandable way, so this was the starting point for our research. MoRe language presents the linear address arithmetic and has a separate stack and heap address spaces. The language has direct memory access and address arithmetic, hence its memory model fully represents C programming language address memory model. There are only integer and pointer data types in MoRe. The algorithm traverses the program and calculates a set of configurations at every moment. The configuration consists of three objects: a set of address variables, a set of address expressions and a set of pairs of "synonyms" – variables, which point to one cell in current configuration. For recalculation the state an operator "aft" was introduced, which for every possible state and statement properly defines a new state after execution of the statement. The syntax grammar of this language is given in Fig. 1. Bachelor Thesis [6] presents simple implementation and analysis of MoRe language. Bachelor Thesis [7] implements simplified C language with MoRe language interface, which can be compiled using LLVM. The syntax and semantics of this thesis' implemented language is close to MoRe's.

$$\begin{aligned} P ::= \quad & skip \mid var\ V = C \mid V := T \mid V ::= cons(C*) \mid \\ & \mid [V] := V \mid V := [V] \mid dispose(V) \mid (P\,; P) \mid \\ & \mid (if\ then\ P\ else\ P) \mid (while\ do\ P) \end{aligned}$$

Fig. 1. The syntax grammar of MoRe language: start variable is P, C is constant integer, and C* is list of integers with "," character between them

III. Methodology

In this section we informally introduce and overview a simplified model language Alias. Though the real implemented language has same syntax as presented here, it's semantic is developed more practically oriented and proposes new instruments.

A. Alias Programming Language Overview

The implemented version of the Alias language has/offers

- Two types — integer and pointer (to be tracked in analysis)
- Variable definitions, assignments, and annotations (assumptions)
- Blocks, If- and While-statements, Procedures.

Program may be split on multiple files. BNF syntax definition of language is given in Figure 2. However, the semantics are very restricted.

- *Type* ::= int | ptr
- *Program* ::= *Block*
- *Block* ::= {[*Statement*]}
- *Statement* ::=

– *Block*	Block
– def *Ident Type*	Definition
– *Ident* := *Expression*	Assignment
– *Ident* <- *Expression*	Movement
– free(*Ident*)	Deallocation operator
– if (*Expression*) *Block*	If statement
– if (*Expression*) *Block* else *Block*	If/Else statement
– while (*Expression*) *Block*	While statement
– func([int *Ident* | ptr *Ident Integer Integer*] *Block*	Function definition
– call *Ident*([*Ident*])	Function call

- *Expression* ::=

– *Ident*	
– *Integer*	
– $*Ident*	Dereference
– *Expression* + *Expression*	
– alloc(*Int*)	Allocation operator

Fig. 2. The syntax grammar of implemented language

B. Outlines of validation algorithm (static semantics)

Memory safety validation is done using the following method (algorithm).

- Program (text) is parsed line by line maintaining (in form of states) a set of known relations "pointer points to cell in heap" but ignoring any information about integer variables.
  - At some stages the states (known relations "pointer points to cell in heap") can split, as there is no information about integer variables.
  - If in a current line there is no pointer variable, which points to any heap cell, then it means memory leak happened.
  - if there is a dereference of a pointer variable, which at some state points out of allocated area, then access violation happened.

C. Configuration

Every configuration contains

I: A set of local variables/identifiers, which have pointer type.

A: a set of allocated cells in the heap (each cell in the form "Meta-variable + Integer-phase")

S: For every identifier in I appointed cell in A, or an exceptional value "OUT".

D. Legal Types of Assignments

There are three types of assignments:

1) int := int – i.e., an integer expression is assigned to an integer variable
2) [ptr -> ptr] := ptr – i.e., a pointer expression is assigned to a pointer
3) [ptr -> int] := int – i.e., an indirect assignment to integer variable

Only the second type affects on configuration.

Note, that storage pointer variables on heap doesn't affect on configuration. Hence validation of multidimensional arrays of structures, for example, is not supported by our analysis.

E. Some optimizing assumptions

We make the following (informal) assumptions about programs (for boosting of validator).

- The number of local variables is not very big, while the number of heap cells can be very big, but (as now) is assumed constant.
- Since the number of configurations grows exponentially, we implement 'assume' statement, which filters the configurations which pass given condition (but programmer is responsible for the correctness of this assumption).
- The current number of configurations is counted, so the programmer can get number of configurations in real time in IDE.

F. Static semantics for pointers

Program traversed recursively. For the following statements corresponding actions made:

- Block affects only on visibility scopes of variables. It doesn't change state.
- Definition affects only on visibility scopes of variables. It doesn't change state.
- Assignment depending on types does following:
  - Destination has pointer type, and source has pointer type. If assignment has form $a := b + x$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$ then in new state this configuration has $a = b_v + (b_p + x)$. For example, if there was configuration with $(b = \_0 + 3)$ and statement $a := b + (-1)$ was executed, the next configuration will be with $(b = \_0 + 2)$. If after this in some configuration there is no a_v, then memory leak happened.
  - Destination has pointer type, and source has integer type. For every configuration and every allocated cell new configuration created where destination points to such cell.
  - If destination has integer type, the state is not changed.
- Assumption works as a guard, i.e., it removes configurations, where the assumption condition is false. If assumption has form $assume(a = b + x)$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$, then if $a_v \neq b_v$ or $a_p \neq b_p + x$, then condition is false. If assumption has form $assume(a < b + x)$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$, then if $a_v \neq b_v$ or $a_p \geq b_p + x$, then condition is false.
- If-statement is traversed in the following steps. Firstly, the first branch is validated. Then the sizes of all lists, which were allocated during this are saved and set to 0. Finally, the second branch is validated, and finally the sizes of lists are restored. *If there is allocation in one branch, then the list will be added to states, but it won't appear in any configuration in second branch, hence it is guaranteed, that an alert will be shown.* (Probably it is a solvable problem, we can force to allocate to the variable the same size at the last assignment in both branches.)
- While-statement is traversed in the following steps. The body is validated, and if state has been changed, the body validated again. There is a threshold (set in validator) for number of these iterations, after exceeding which, it is assumed that the loop is infinite. The variables declared in a loop are scoped in the loop.
- A function actually is a procedures, its definition contains set of formal parameters (as arguments) in its signature. Each pointer parameter has two associated integer values, which guarantee the minimal number of sells before and after a call. The function doesn't return values, but can change its pointer arguments (i.e., actual arguments are passed name to function).
- Function call contains parameters as actual arguments of function. If in some configuration some pointer variable variable (passed to the function as an actual argument) doesn't satisfy the minimal size of allocated space, then it causes a run-time error.

IV. Implementation

This section describes implemented language, which is based on model language described previously, but mostly oriented on practical usage.

A. Overview

By default, the whole process of building and execution consists of the following sequential stages.

Parsing `calias` parses input files and builds abstract syntax tree;
Validation `calias` traverses tree and does static analysis;
Compilation `calias` traverses tree and writes equivalent x86 assembly code;
Assembly `nasm` builds object file;
Linking `gcc` links object file and provides its `malloc` and `free` functions.

B. Tool-chain for the Alias Language

The compiler is implemented using language *C++* for *GNU G++* compiler and preferably uses *C++17* standard. The implementation can be found in corresponding GitHub repository. The output executable is called `calias`.

For front-end no lexical and parser generating tools, or a framework for development of domain specific languages were used, both lexer and syntax parser were implemented from scratches.

C. Validation

The validation is done as traversing the abstract syntax tree with passing and modification a *context*.

A context consists of the following components (though its implementation is a bit more complicated):

- stack of variables;
- stack of functions;
- vector of sizes of packets;
- set of states.

A state is a vector, which for every declared variable contains

- either the pair consisting of a packet, in which it lays, and a phase (i.e, a shift relative to the beginning of packet, to which the variable points);
- or a special value `OUT`.

Note, that here we use a terminology that differs from terminology in the section III: context here is used instead of state, and state here is used instead of configuration (since these terminology is commonly adopted in program languages implementation community).

D. Rules definitions

This is a formal description of working process of validator. It omits some non-important cases, for more clear understanding.

The rule is described in two lines. Conclusions are written in the bottom line $A \vdash B \to C$ and premises — in the top line $D \vdash E \to F$. This means, that if we have to traverse node B of abstract syntax tree and the current context is A, then we have to create new context D, do recursive call on node E, which will return context F and then return context C. If a rule has no premises it is an axiom (i.e., no further recursive calls).

Let us introduce some notation conventions. Meta-variable VS stands for variable stack, FS — for function stack, PS — for vector of packet sizes, and SS — for set of states. If the actual value of some of the listed meta-variables does not change in a rule, then it is presented implicitly, while any change of actual value must be specified in the rule explicitly. For example, if there is a line $C[FS] \vdash statement \to C[FS : foo]$, then it means that the output context is almost the same as input, but the value of FS (to which foo is appended to the end of the function stack).

Operation ":" appends the value to the end of the stack; it is also used to denote, that the element has instances in the structure. Operation "::" concatenates two stacks or vectors; it is also used to denote, that the elements of second list are presented in the first list (neglecting the order). As usual, (x, y) stays for a pair of two elements and $x := y$ denotes an update assigning the value of y to variable x.

There are following additional operators:

- $packet(x, S)$ returns the identifier of the packet, to which the variable x is bound in state S;
- $phase(x, S)$ returns the phase with respect to the beginning of the packet, to which variable x is bounded in state S;
- $value(x, S)$ returns a pair consisting of $packet(x, S)$ and $phase(x, S)$;
- $packet\_size(x)$ returns the size of packet x (remark that it is unique in all states);

The `CHECK` operator works as a guard, i.e., it is used to evaluate the expression (after `CHECK`), and if it is false, stops validation with corresponding error message.

E. List of Rules

1) Block: Remember the size of stack of variables. Traverse all statements in body, and crop stack of variables to previous size.

$$\frac{C \vdash S_1 \to C_1; \ldots; C_{n-1} \vdash S_n \to C_n}{C[VS, FS] \vdash \{S_1, \ldots S_n\} \to C_n[VS, FS]}$$

2) Definition: Append the variable name to the stack of variables.

$$\frac{}{C[VS] \vdash \text{def a type} \to C_2[VS : a]}$$

3) Assignment: Different behaviour for integer and pointer types.

For integer we just need to check the right part is a valid expression.

$$\frac{C[VS : a] \vdash \text{expr} \to C}{C \vdash \text{a := expr} \to C}$$

There are three options for assignments with pointers — alloc, shift by a constant, and more complicated expressions in the right-hand side..

Alloc expression creates an additional packet.

$$\frac{}{C[VS : a, PS, SS] \vdash \text{a := alloc (x)} \to C[[PS : x], \forall S \in SS \to value(a, S) := (size(PS), 0)]}$$

Shift (i.e., pointer + constant integer) assigns the corresponding value.

$$\frac{}{C[VS :: [a, b], SS] \vdash \text{a := b + x} \to C[\forall S \in SS \to value(a, S) := (packet(b, S), phase(b, S) + x)]}$$

For all other cases the state into states, where the variable points to one of all possible allocated cells, and check right part expression.

$$\frac{C \vdash \text{expr} \to C}{C[VS : a, SS] \vdash \text{a := expr} \to C[\forall S \in SS \to \forall packet\ p, x \in [0, packet\_size(p)) \to value(a, S) := (p, x)]}$$

4) Movement: Check, that pointer at left size if correct, and check the right part expression.

$$\frac{C \vdash expr \to C}{\begin{array}{c} C \vdash \text{a <- expr} \to C \\ \text{CHECK}\forall S \in SS\ phase(a, S) \in \\ [0, packet\_size(packet(a))) \end{array}}$$

5) Free: Check, that the pointer has phase zero, and have same packet in all states. Assign packets of all pointers, which point to this packet, to *OUT*.

$$\frac{}{\begin{array}{c} C[PS] \vdash \text{free(a)} \to C[PS \to packet(p) := 0, \forall S \in SS \to \\ \forall x, packet(x, S) = p \to value(x, S) := (OUT, 0)] \\ \text{CHECK}\forall S \in SS : packet(a, S) = p\ and\ phase(a, S) = 0 \end{array}}$$

6) Function definition: Flush all variables and append argument variables. Each pointer variable which has nonzero pre size lays in own packet with size equal to pre size. Check body. At the end check that all pointer variables lays in different packets with at least post size distance from end of packet and have same packet in all states. Restore variables and append function.

$$\frac{[[a, b], [foo], [in_a, in_b], a := (\_a, 0), b := (\_b, 0)] \vdash block \to C_2}{\begin{array}{c} C[FS] \vdash \\ \text{func foo (def a ptr in\_a out\_a, def b ptr in\_b out\_b) block)} \to \\ C[FS : foo] \\ \text{CHECK}\forall var\ x, S \in SS\ packet(x, S) = \\ \_x\ and\ phase(x, S) \in [0, packet\_size(\_x) - \\ out_x)\ and\ \forall var\ x \neq var\ y\ packet(x, S) \neq packet(y, S) \end{array}}$$

7) Function call: Check, that all pre conditions are satisfied: all argument variable lay in different packets with at least pre size distance from end of packet and have same packet in all states. Remove all passed packets, as if they were freed, and create new packet for each argument variable.

$$\frac{}{\begin{array}{c} C[FS : foo, PS, SS] \vdash call\ foo(args) \to C[PS :: \\ [out_a, out_b], \forall S \in SS \to value(a, S) := (new\_a, 0), b := \\ (new\_b, 0)] \\ \text{CHECK}\forall x \in args\ S \in SS\ phase(x, S) \in \\ [0, packet\_size(\_x) - in_x)\ and\ \forall x \neq y \in \\ args\ packet(x, S) \neq packet(y, S) \end{array}}$$

F. Compilation

The compilation is done into *x86 intel Assembly*. The compiler using almost same structure as validator. But its context is adapted for compilation. The compilation is done as traversing abstract syntax tree and building assembly code, which is the assembled using *nasm* and linked using *gcc*, which provides implementations of functions *malloc* and *free*.

G. Assembly structure

There are rules for compilation, which are defined the same way, as for validation. Though, they are not interested in scope of this thesis.

- There is an enter point of the program;
- There is declarations of functions malloc and free, their implementations have to be provided;
- The System V ABI[8] is used, which makes this file compatible with programs written is `C` language;
- The 32 bit assembly is used, thus the only data types have same size of four bytes;
- Only the simple general purpose instructions are used;
- The expressions push calculated result on the top of current stack. The binary operators do recursive call of one operand, then pushes stack and do recursive call of the other operand;
- There are no optimizations.

H. The IDE

The IDE is implemented from scratches in language *C++* for *GNU G++* compiler and preferably uses *C++17* standard. It widely uses *NCurses* library for implementation text editor. The implementation can be found in corresponding GitHub repository. The output executable is called `ideal`.

V. Evaluation: examples of memory errors

A. Detected Errors with one Configuration

In this section will be presented examples of programs (each with a simple error) that have only one configuration on every state.

```
def a ptr
a := alloc (3)
a := alloc (2)
```

Listing 1. Example of memory leak

After the second assignment, there is a configuration (this is the only one configuration in this state), where there is no page, which was allocated first. The validator will show corresponding error on the third line.

```
def a ptr
a := alloc (3)
a := a + 4
def b int
b := $a
```

Listing 2. Example of access violation while dereference

In the fifth statement there an attempt to dereference the pointer, while there is a configuration, where this pointer points out of page (this is the only one configuration in this state). The validator will show corresponding error on the fifth line.

```
def a ptr
a <- 4
```

Listing 3. Example of access violation while movement

In the second statement there is an attempt to move value by pointer, while there is a configuration, where this pointer out of page (this is the only one configuration in this state). The validator will show corresponding error on the second line.

```
def a ptr
a := alloc (3)
a := a + 1
free (a)
```

Listing 4. Example of access violation while free

In the forth statement there is an attempt to free page by pointer, while there is a configuration, where this pointer is not at the beginning of page (this is the only one configuration in this state). The validator will show corresponding error on the forth line.

B. Detected Errors with multiple Configurations

Next let us discuss examples of programs which have multiple configurations in every state.

```
def a ptr
if (1) {
    a := alloc (3)
}
```

Listing 5. Example of memory leak on branching

In the end of body of *if* statement there is memory leak, since there is a configuration, where there is no allocated page (there are two configurations: with *if* and without). In general, the allocations can only be places at root blocks in function bodies.

```
def a ptr
def b ptr
a := alloc (3)
b := alloc (4)
def c ptr
if (1) {
    c := a + 0
}
else {
    c := b + 0
}
free (c)
```

Listing 6. Example of unpredictible free

In the free statement there are two configurations, but in these configurations the variable c points to different pages. It is restricted, as there is no way to continue validation.

VI. Conclusion

In this work in progress paper, firstly we briefly review some approaches to memory safety analysis. Then we proceed to a new variant of alias calculus and propose several changes, stemmed from the the C programming language memory model. Finally, we describe our implementation of a model language, our static analysis tool, and present several experiments showing analysis' potential (as we believe).

Still we need to try validator on a "large" source code file containing more than 100 lines of code. Right now we foresee a problem with scaling our analysis to "large" programs and on programs in a programming language from the real world. Additionally, a crucial missing piece in the theory is the handling of dynamic arrays and recursive functions.

Bibliography cited

[1] L. O. Andersen, "Program analysis and specialization for c programming language," in *DICU*, [Online]. Available: http://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf, May 1994.

[2] S. V. A. Kogtenkov B. Meyer, "Alias calculus, change calculus and frame inference," in *Science of Computer Programming*, [Online]. Available: http://is.ifmo.ru/articles_en/2013/meyer-calculus-2013.pdf, Nov. 2013.

[3] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Carnegie Mellon University*, [Online]. Available: https://www.cs.cmu.edu/~jcr/seplogic.pdf, Jul. 2022.

[4] P. O'Hearn, "Communications of the acm," in *Carnegie Mellon University*, [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3211968, Feb. 2019.

[5] A. V. N.V. Shilov A. Satekbayeva, "Alias calculus for a simple imperative language with decidable pointer arithmetic," in *Novosibirsk Computing Center*, [Online]. Available: https://nccbulletin.ru/files/article/shilov_satekbayeva_vorontsov.pdf, 2014.

[6] L. I. Lygin, "Alias calculus in c-like languages," 2021.

[7] G. Dolgov, "Implementing alias calculus for c programming language using llvm," 2022.

[8] A. J. Michael Matz Jan Hubicka, *System v application binary interface*, [Online]. Available: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf, Jul. 2012.