A Memory Model for Static Analysis of C Programs

Zhongxing Xu[1], Ted Kremenek[2], and Jian Zhang[1]

[1] State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
xzx@ios.ac.cn,
zj@ios.ac.cn
[2] Apple Inc.
kremenek@apple.com

Abstract. Automatic bug finding with static analysis requires precise tracking of different memory object values. This paper describes a memory modeling method for static analysis of C programs. It is particularly suitable for precise path-sensitive analyses, e.g., symbolic execution. It can handle almost all kinds of C expressions, including arbitrary levels of pointer dereferences, pointer arithmetic, composite array and struct data types, arbitrary type casts, dynamic memory allocation, etc. It maps aliased lvalue expressions to the identical object without extra alias analysis. The model has been implemented in the Clang static analyzer and enhanced the analyzer a lot by enabling it to have precise value tracking ability.

1 Introduction

Recently there has been a large number of works on bug finding with symbolic execution technique. In these works, tracking values of different memory objects along a single path is a common requirement. Some works get the run-time addresses of memory objects by actually compiling and running the program [1][3]. These are dynamic techniques. Programs being checked must be instrumented and linked with an auxiliary library and run. Other works solve the memory object identifying problem through various static ways. The simplest approach is to only track simple variables with names, and ignore multi-level pointers, array elements, and struct fields. This would surely sacrifice much analysis power.

This paper proposes a memory modeling method that is particularly suitable for symbolic execution of C programs. It enables the symbolic execution to identify and track each memory object precisely. We give algorithms that enable the mapping from C l-value expressions to memory objects during the analysis. Thus no separate alias analysis is required.

Memory model is the way that the analysis tool models the storage of the underlying machine on which the code runs. It is the basis of language semantics simulation and a key component of static code analysis tools.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part I, LNCS 6415, pp. 535–548, 2010.

Memory model determines how accurately the tool can simulate the language semantics and thus affects the ability of the tool to detect bugs. Surprisingly, few papers addressed the memory modeling issue in the static analysis field.

Unlike other high level programming languages, the C programming language assumes a rather low level memory model and provides extensive kinds of memory operations, such as multi-level pointers, arbitrary pointer arithmetic, built-in array and struct data types. Pointers in C can point to arbitrary locations of the memory, and can be cast freely between different types. All these make it difficult to simulate the C semantics accurately.

In Section 2 and 3, we introduce two basic memory models which are commonly used but have some limitations. In Section 4 we describe our novel memory model, which can precisely map each l-value expressions to the corresponding memory object. In Section 5 we describe how to simulate the C language semantics with the new memory model. We give some examples and implementation in Section 6 and 7. We compare with related works in Section 8 and conclude in Section 9.

2 Name Binding Model

The name-binding model is the most basic storage model present in the semantics textbooks. In this model, the computer memory is seen as name-value pairs. When an assignment expression is evaluated, we bind the name of the variable on the left-hand-side to the value of the expression on the right-hand-side. While this model is very common, it is not powerful enough to be used for simulating the C semantics. Consider the following example:

```
int x, y;
int *p = &x;
x = 3;
*p = 4;
y = x;
```

The C language has pointers. In this example, `p` is a pointer variable that points to the variable `x`. The presence of pointers brings the aliasing problem. The aliasing problem is that two or more names can represent the same storage location. In this example, `*p` and `x` are aliases.

The name-binding model cannot deal with the aliasing problem. When we modify the value of a name, we should also modify the value of all names aliased to it accordingly. But in no way can we know the aliases of a name in the name-binding model.

3 Array Simulation Model

One of the deficiencies of the name-binding model is that it lacks the concept of memory locations. The pointers in C, however, are invented to manipulate memory locations.

Zhang proposed an array model for the memory [11]. The motivation of the array model is simple: intuitively the memory can be seen as a large array. If we allocate all variables on an array, all operations on variables can be transformed into operations on corresponding array elements, and most importantly, the array element indices can be taken as the memory locations for the variables.

The example in Section 2 can be transformed using the array memory model:

```
Assume mem[] is the memory simulation array.
Memory allocation:
  x: mem[1], y: mem[2], p: mem[3]

mem[3] = 1; // p = &x; mem[3] is 'p',
            // 1 is x's simulation location.
mem[1] = 3; // x = 3;
mem[mem[3]] = 4; // dereference '*' means
                 // we have to nest mem[]'s
mem[2] = mem[1]; // y = x;
```

The array model is somewhat more powerful than the name-binding model. It can solve some problems in program analysis [10]. But its disadvantages are also obvious.

The array model requires that every variable has a fixed position. This can be achieved as long as we know the exact size of every memory object. Once we have a memory object of unknown size, such as a variable-length array or a heap object of unknown size, the array model is unusable.

A slight improvement for the array model is that instead of using a single large array, we use multiple arrays. Each memory object has a corresponding array for it. But composite memory objects are difficult to represent in this model. For example, for object struct array `struct s { int d } sa[2];`, if we use a single array to represent it, it still has the same weakness as the single array model. If we use multiple arrays to represent it, we will lose the hierarchy relation among memory objects.

4 Region Based Ternary Model

Formal semantics models program state with two mappings [8]: a variable environment that associates a location with each variable and a store that associates a value with each location. Formally, we define a variable environment Env as a mapping of

$$Env = Var \rightarrow Loc$$

where Loc is a set of locations. A store is the mapping of

$$Store = Loc \rightarrow Value$$

The name-binding model in Section 2 lacks the concept of locations. The array model in Section 3 does have the concept of locations. It uses concrete integers

to represent locations. This concretization of location limits the applicability of the array model.

In this section we develop a new representation of locations: regions. A *region* is an abstract chunk of memory corresponding to an lvalue expression in the C programming language.

According to the C standard [5], an *lvalue* is an expression with an object type. That is, an lvalue expression has an associated memory object. We use an abstract region to represent this memory object.

Thus in our region based memory model, every lvalue expression should have a corresponding region. Furthermore, lvalue expressions indicating the same memory object should have the same region corresponding to them. Next we describe the way to get the region associated with an lvalue expression.

4.1 Region Hierarchy

We define several kinds of regions. For explicitly declared variables, we have *VarRegion* identified by the variable declarations. Every variable has a unique VarRegion associated with it.

If the variable is of array or struct type, it has subobjects called element or field. To represent this hierarchy between memory objects, we introduce the concept of subregions. A region can be the subregion of another region. There is a super region pointer pointing to the super region of a subregion.

For array elements, we have *ElementRegion* with its super region pointer pointing to its array region. Likewise, for struct fields, we have *FieldRegion* with its super region pointer pointing to its struct region. ElementRegions are identified by their array regions and the indices. FieldRegions are identified by their struct regions and the field declaration.

In C, there are three kinds of storage classes: local (stack), global (static), dynamically allocated (heap). We also have a *MemSpaceRegion* for this concept. There are three MemSpaceRegions for stack, heap, and static storage respectively. All local variables have the stack MemSpaceRegion as their super region. All global variables have the static MemSpaceRegion as their super region. All dynamically allocated objects (mostly by malloc()) have the heap MemSpaceRegion as their super region.

In static analysis, we sometimes would have symbolic values. For example, we assume function arguments and global variables holds symbolic values at the entry of the function. If the symbolic variable is a pointer, it may point to some unknown memory block. For this case, we have *SymbolicRegion* for representing the memory block pointed to by the symbolic pointer. SymbolicRegions are identified by the symbolic pointer values that point to them.

4.2 Region Properties

Besides storage classes, memory objects have extents, or sizes. Some objects' extents are explicit. For example, a char variable has the extent of one byte. But dynamically allocated objects can have various extents. We record this

information with a region-extent mapping from the object's associated region to its extent. The extents can be in various forms: concrete integer value or symbolic unknown value.

Memory state is modeled by the bindings of the regions. There are two kinds of bindings: direct binding and default binding. After an assignment expression, like `x = 3;`, we set the direct binding of region of `x` to `3`. Default binding is usually set on super regions. If a subregion has no binding, then it could use its super region's default binding as its value. For example, after function call `bzero(buf)`, we can set the default binding of the region pointed to by `buf` to 0. Without default binding, we have to set each element of that region to 0, which is prohibitively expensive for large arrays.

4.3 Region Views

The C programming language permits arbitrary conversions between types of pointers. This poses great difficulty to static analysis tools.

Consider the following code snippet:

```
void *p = malloc(10);
char *buf1 = (char *) p;
buf1[0] = 'a';

int *buf2 = (int *) p;
buf2[0] = 0;

char c = buf1[0];
```

This is a contrived example. But the code pattern is fairly common in system programs. Programmers often allocate a generic block of memory, then cast it to different types for different uses. How do we deal with such (ab)uses of the C type system?

The essence of this problem is that in C we can have typeless generic chunk of memory. In the code example above the dynamically allocated memory pointed to by `p` is such a chunk of memory. The later casts from it to `char*` and `int*` can be interpreted as installing "views" to it. When it is operated by pointer `buf1`, it is viewed as a memory block of type `char`. When it is operated by pointer `buf2`, it is viewed as a memory block of type `int`.

We set up a region view mapping from a region to its various views: some anonymous typed region. Casting a generic block of memory to some type is equivalent to adding a new view to the block of memory. We create an anonymous typed region to represent the view. Later when the generic memory block is cast to another type, a new anonymous typed region is created to represent this new view. The idea is illustrated in Figure 1.

A region can have multiple views simultaneously. But only one view can be in effect at a time. When the region is operated on by one view region of it, the information associated with its other view regions must be invalidated.

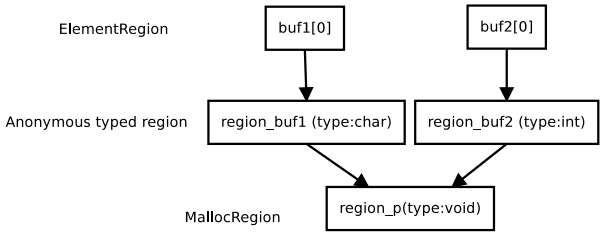

Fig. 1. Anonymous typed region illustration. Region region_p is the generic memory block pointed to by p. Region region_buf1 is the same memory block with type char. Region region_buf2 is the same memory block with type int. Regions buf1[0] and buf2[0] are the ElementRegions on regions region_buf1 and region_buf2.

Returning to the example, after expression `buf1[0] = 'a';`, `buf1[0]` has value `'a'`. But after expression `buf2[0] = 0`, we not only set the value of `buf2[0]` to `0`, but also removes the binding for `buf1[0]`. When `c` is assigned `buf1[0]`, we can spot that `buf1[0]` is an undefined value.

5 Simulation of C Semantics

With the region based memory model, we can simulate the C semantics precisely. We still use the semantics model in Section 4. But we extend it as follows.

The program state is modeled by two mappings: Environment and Store. We define Environment as a mapping from expressions to values:

$$Env : Expr \rightarrow SVal$$

where Expr is an element of C expressions. SVal is some kind of abstract values, which we will describe in Section 5.1. The Store is defined as a mapping from Regions to values:

$$Store : Region \rightarrow SVal$$

where Region is the abstract representation of memory objects.

5.1 Abstract Values

We divide the values occurring in the symbolic simulation into two classes: locations *loc* and non-locations *non-loc*. A *loc* can be a region or unknown. *loc* represents an abstract memory location. Pointers in C all have *loc* values. The rests are non-location values *non-loc*. A *non-loc* can be a concrete integer, a symbolic integer, etc.

5.2 l-Value and r-Value

The C programming language standard [5] classifies expressions into lvalue and rvalue. Expressions referring to objects are lvalue expressions. The rests are

rvalue expressions. To evaluate expressions, we have to distinguish between an expression's l-value and r-value. We define an expression's l-value to be the memory location of its associated memory object. An expression's r-value is the value associated with the memory object if the expression is an lvalue expression or the semantic value if the expression is not an lvalue expression.

Note that lvalue expressions have both l-value and r-value. Non-lvalue expressions only have r-value, i.e., their semantic value. l-value can only be *loc*s, while r-value can be both *loc*s and *non-loc*s. For example, `*p`'s r-value is a *loc*.

5.3 Evaluation Rules

In this section we give some rules of evaluation of C expressions. We define some notations: l-value(e) returns expression e's l-value, which is a loc. r-value(e) returns expression e's r-value, which is a *loc* or *non-loc*. VarRegion(v) return the unique region associated with variable declaration v. ElementRegion(r, i) returns the unique region associated with the i'th element of array region r. Similarly, FieldRegion(r, f) returns the unique region associated with the f field of struct r. $Store(loc)$ returns the value associated with location *loc*.

Evaluation rules for main kinds of expressions of C are specified in Table 1.

Table 1. Evaluation rules for C expressions

	l-value	r-value
constant n	N/A	n
variable x	VarRegion(x)	Store(l-value(x))
array a	VarRegion(a)	N/A
`a[e]`	ElementRegion(l-value(a), r-value(e))	Store(l-value(a[e]))
`s.d`	FieldRegion(l-value(s), d)	Store(l-value(s.d))
`p->d`	FieldRegion(r-value(p), d)	Store(l-value(`p->d`))
`&expr`	N/A	l-value(expr)
`*expr`	r-value(expr)	Store(r-value(expr))
`alloca(n)`	N/A	AllocaRegion(n)
`malloc(n)`	N/A	MallocRegion(n)

The region-base memory model supports almost all kinds of C expression semantics: arbitrary level pointer, composite struct and array data types, dynamic memory allocation, and symbolic pointer.

For pointer arithmetic, we assume that it only happens to element regions. We get the index of the element region, apply the offset to it, and get an element region with the new index.

For dynamically allocated memory, we create a MallocRegion with the size and an execution counter to differentiate memory chunks allocated in the same statement.

Let us look at an example showing how the evaluation rules are applied. Consider code snippet:

```
struct s1 {
  int d;
};

struct s2 {
  struct s1 *p;
};

void foo(void) {
  struct s1 data;
  struct s2 *sp;
  int a[2];

  sp = malloc(sizeof(struct s2));
  sp->p = &data;
  sp->p->d = 3;
  a[1] = data.d;
}
```

After we have processed the three variable declarations in function `foo`, we have program state shown as follows:

Expression	Region	Value
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	undefined
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined

When processing the malloc statement, we will have a new region in the program state. The updated program state is shown as follows, where $MallocReg$ represents the memory region allocated by `malloc()`.

Expression	Region	Value
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	**region 7**
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
$MallocReg_0$	**region 7**	**N/A**
$MallocReg_0.p$	**region 8**	**undefined**

Next we process statement `sp->p = &data;`. Evaluating expression `&data` requires the l-value of `data`, which is region 1. Then we get the r-value of `sp`, which is region 7. getFieldRegion(region7,p) returns region 8. The updated program state is shown as follows:

Expression	Region	Value
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
$MallocReg_0$	region 7	N/A
$MallocReg_0.p$	region 8	**region 1**

Next we process statement `sp->p->d = 3;` Similarly we get the r-value of `sp->p`, which is region1, and the l-value of `sp->p->d`, which is region2. The updated program state is shown as follows:

Expression	Region	Value
data	region 1	N/A
data.d	region 2	**3**
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
$MallocReg_0$	region 7	N/A
$MallocReg_0.p$	region 8	region 1

Then we process statement `a[1] = data.d;` The l-value of `data.d` is region2, then we get its r-value 3. The l-value of `a[1]` is region 6. The updated program state is shown as follows:

Expression	Regions	Values
data	region 1	N/A
data.d	region 2	3
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	**3**
$MallocReg_0$	region 7	N/A
$MallocReg_0.p$	region 8	region 1

From the above example we can see that all memory objects are represented unambiguously, and their corresponding regions are computed on the fly with little overhead. The region hierarchy is shown in Figure 2.

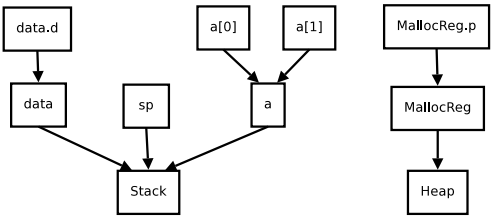

Fig. 2. The final memory region hierarchy. The arrow points to the super region.

6 An Example

In this section we use an example to show the power of the region-base memory model. Consider the following code snippet.

```
struct s { int data[2]; }
void f(struct s buf) {
1  int i = 3, *q = NULL;
2  struct s* p;

3  if (buf.data[1] == 1)
4    q = &i;

5  p = &buf;

6  if (p->data[1] == 1)
7    p->data[0] = *q;

8  return;
}
```

In this code, if we do not track the value of the struct field `buf.data[1]`, we would have a path leading to the NULL pointer dereference of `q` at line 7: 1,2,3,5,6,7,8.

With the region memory model, we can precisely know that `buf.data[1]` and `p->data[1]` refer to the identical memory object. Thus the path condition along the previous path cannot be satisfied: `buf.data[1]` cannot be simultaneously equal and unequal to 1.

The region hierarchy and storage mapping is shown in Figure 3. When `buf.data[1]` is evaluated, we get the region associated with `buf`, `buf.data`, `buf.data[1]` respectively. Then the value of `buf.data[1]` is retrieved, which is a symbolic value \$1. When `p->data[1]` is evaluated, we first get `p`'s rvalue, which is the memory region of `buf`. Then along the same way, we get the value for `buf.data[1]`, which is the symbolic value \$1.

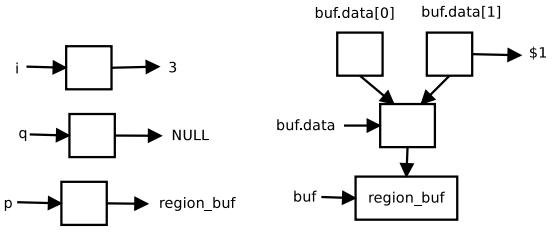

Fig. 3. The region hierarchy and storage mapping. Squares represent memory regions. $1 is the symbolic value of buf.data[1]

The for path 1,2,3,5,6,7,8, we get path condition:

$$\$1! = 1 \wedge \$1 == 1,$$

which cannot be satisfied. There will be no false alarm for the NULL pointer dereference of `q` at line 7.

7 Implementation

The region based memory model is fully implemented in the Clang analyzer [2]. Clang is a new C, C++ and Objective C front-end for the LLVM [7] compiler. The static analyzer is an official part of Clang that find bugs in C and Objective-C programs. The second author of this paper is the original architect of the static analyzer core [6]. The analyzer has very good modularity. It is designed in such a way that main components can be swapped in and out. Basically it has the following components:

- The core engine, driving the analysis through the program in some order.
- State manager, managing the simulated program states.
- Constraint manager, recording and solving path conditions collected along a program path.
- Store manager, modeling the program storage.

The region based memory model is implemented as the region store manager in the tool. It adds the full field sensitivity to the analysis.

An implementation feature that is worth mentioning is the lazy binding technique we employed. It can be best illustrated by an example. Consider code snippet:

```
for (...) {
  ...
  if (...) {
L:    int buf[8096];
  }
}
```

As we are using the region based memory model, theoretically we have a separate region for each of the array elements on line `L`. Since `buf` is a local array, we have to initialize all of its elements to have value *undefined*. Because we could have multiple paths leading to the definition of `buf`, we have to initialize its 8096 elements multiple times. This proved very time consuming. Moreover, few of the elements are actually used during later analysis.

To solve this problem, we switched to an implementation that employed lazy initialization. We do not initialize any variables. Instead, we compute their initial value the first time it is used according to their storage classes. This optimization reduced the analysis cost dramatically.

8 Related Work

Hampapuram et al. used a region-based memory model in their work [4]. Our model is different than theirs in various ways. The main feature of our model is the region hierarchy, which plays an important role in tracking and distinguishing memory objects and reasoning about their relations.

Saturn [9] is a general framework for building precise and scalable static error detection systems. It models program operations with Boolean satisfiability (SAT) techniques down to the bit level. Pointers in Saturn are modeled with Guarded Location Sets (GLS). The GLS method essentially gives each location an explicit name and records the set of locations a pointer could reference at a particular program point. This memory representation is not as rich as our region-based memory model, which treats memory objects as first-class values with properties such as extents and relationships to other memory regions (e.g., region views to capture type-casting relationships). One consequence of these differences is that Saturn's strictly name-based memory model does not amend itself well to reasoning about pointer arithmetic nor array operations, both of which are handled naturally and precisely in our region-based model (i.e., by reasoning about the indices of ElementRegions).

Other bug finding tools like EXE [1] and DART [3] circumvent the memory modeling problem by actually compiling and running the program being checked. They use the run-time addresses of memory objects to distinguish them.

The closest related work to ours in the traditional program analysis field is alias analysis. We both try to solve the similar problem: the correspondence between expressions and memory objects. The requirements, however, are different.

Alias analysis for compiler optimization aims for a conservative result. It usually computes "may" aliases. We aim to get more precise alias relation. Due to the path sensitive character of our application, we do have more precise program state information that enables us to get more precise alias information.

On the usage aspect, alias analysis is often used as a separate analysis pass on the program. Then its results are stored for later use. Our memory model is used in combination with the whole symbolic analysis of the program. The corresponding memory regions are computed on the fly during the symbolic analysis. There is no separate "region analysis" pass for our analysis model.

9 Conclusion

We designed a region-based memory model for path-sensitive symbolic program analysis. In summary, the memory model has the following features which make it more powerful and suitable for symbolic execution:

- It can handle almost all kinds C expressions, including arbitrary levels of pointer dereferences, pointer arithmetic, composite array and struct data types, arbitrary type casts, dynamic memory allocation, etc.
- It maps aliased lvalue expressions to the identical object without extra alias analysis.
- It can represent various properties of memory objects: known and unknown extent, storage class, hierarchy relation, concrete and symbolic values, etc.

The memory model has been implemented in the Clang analyzer [2]. Our preliminary experimentation showed that it is effective for adding field sensitivity to our static analysis tool.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant No. 60633010, 60903049) and the High-Tech (863) program of China (Grant No. 2009AA01Z148).

References

1. Cadar, C., Twohey, P., Ganesh, V., Engler, D.: EXE: A system for automatically generating inputs of death using symbolic execution. Technical report, Computer System Laboratory, Stanford University (2006)
2. Clang static analyzer, `http://clang-analyzer.llvm.org/`
3. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213–223 (2005)
4. Hampapuram, H., Yang, Y., Das, M.: Symbolic path simulation in path-sensitive dataflow analysis. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 52–58 (2005)
5. WG14 ISO/IEC 9899:201x, editor. Programming Languages - C. ISO (1999), `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1336.pdf`
6. Kremenek, T.: Finding bugs with the clang static analyzer, `http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf`
7. The LLVM compiler infrastructure, `http://llvm.org/`
8. Nielson, H.R., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons Inc., Chichester (1992)

9. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using Boolean satisfiability. ACM Transactions on Programming Languages and Systems 29(3) (2007)
10. Xu, Z., Zhang, J.: A test data generation tool for unit testing of C programs. In: Proceedings of the International Conference on Quality Software, pp. 107–116 (2006)
11. Zhang, J.: Symbolic execution of program paths involving pointers and structure variables. In: Proceedings of the Fourth International Conference on Quality Software, pp. 87–92 (2004)