Universitatea "Politehnica" din Timișoara

Facultatea de Automatică și Calculatoare

Verificarea acceselor la memorie în programe C folosind constrângeri simbolice și slicing

- Lucrare de diplomă -

Edvin Török

Coordonator: conf. dr. Marius Minea

iunie 2009

Rezumat

Lucrarea prezintă un instrument pentru detectarea depășirilor de limite în programe C, care poate fi folosit pentru a analiza programe cu sute de mii de linii de cod într-un timp comparabil cu timpul de compilare. O analiză interprocedurală marchează variabilele care influențează validitatea acceselor, iar folosind tehnica de slicing reducem programul doar la codul aferent acestora. Ulterior, printr-o varietate de tehnici de optimizare, obținem un cod pe baza căruia creem expresii simbolice și verificam dacă accesul e în interiorul limitelor. În cazul în care corectitudinea accesului e nedecidabilă doar pe baza expresiilor simbolice luăm în considerare și predicatele care domină accesele de memorie, și interogăm un rezolvitor de constrângeri pentru a determina validitatea accesului. Eliminăm accesele a căror validitate am demonstrat-o, și repetăm pașii de mai sus până la demonstrarea tuturor acceselor sau atingerea unor limite de inlining. Instrumentul detectează cu success erori simple, întâlnite frecvent pe parcursul dezvoltării unei aplicații, de exemplu erori “off-by-one”.

"Politehnica" University of Timisoara
Faculty of Automation and Computers

Interprocedural bounds checker for C programs using symbolic constraints and slicing

- final year thesis -

Edvin Török

Advisor: Assoc. Prof. dr. Marius Minea

June 2009

Abstract

We describe a tool for finding out-of-bounds memory access bugs in C programs, which can be used to check programs with more than 100 000 lines of source code in time comparable to compilation time. Our tool doesn't search for out-of-bounds pointers, but only for dereferences of out-of-bounds pointers (not including NULL dereferences). An interprocedural analysis marks the variables that influence the validity of the accesses, and by using slicing we reduce the program to only the code corresponding to these. Afterwards, by using a variety of code optimization techniques, we obtain a code based on which we create symbolic expressions, and using them we check whether accesses are within bounds using them. If this is undecidable based purely on the symbolic expressions, we take into account the predicates that dominate these memory accesses, and query a solver to determine the validity of the access. We eliminate the accesses we could prove, and repeat the above steps until we either prove all accesses, or we reach the inlining limit. The tool is successful in catching simple cases of common bugs encountered during application development such as off-by-one bugs.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Goal

In C programs one of the most frequently encountered errors are memory errors: `NULL` pointer dereferences, memory leaks, wraparound in pointer arithmetic, bounds overflow. For example, failing to check for memory allocation failures can cause `NULL` dereferences, failing to free resources on error paths can lead to memory leaks, adding a large unsigned value to a pointer can cause it to overflow, and performing pointer arithmetic without checking bounds can cause bounds overflow. These can sometimes be latent for a long time, being discovered only after the program is already in production.

We focus on the latter category of errors: bounds overflow. Although the other errors are serious too (NULL dereferences crash the program, memory leaks can cause Out-of-Memory conditions, pointer overflow is undefined behavior), bounds overflow bugs (buffer overflow, heap overflow, stack overflow, ...) are the source of security vulnerabilities in programs, and thus particularly serious/dangerous.

1.2 Motivation

We aim for an instrument that:

- looks for bounds overflow bugs, even if the allocations and dereferences are in different functions
- is easy to integrate into the compilation process
- runs fast, comparable to compilation time
- has low false positive rate
- is preferably open-source

There are many tools that could satisfy part of these constraints, but:

- Some of them are commercial (Coverity, Polyspace, ...)
- Other do a very thorough check, and don't fit our time constraints, (KLEE, ...)
- Others don't detect dereferences if allocations and dereferences are in different functions, or have high false positive rate (splint, ...)

This thesis describes a tool to detect these errors, with the primary goal of a short analysis time, comparable to compilation time. Our tool is a bug-finder, we do not propose to prove the correctness of a program, or absence of bugs. This is not possible in general, and verification tools generally have difficulty in handling large-scale programs. To achieve speed and scalability, we will have to sacrifice accuracy as a tradeoff.

1.3 Related Work

There has been a lot of research done on static analysis, and detecting bounds overflows:

- ASTRÉE [BCC+02] is a tool used for checking avionics code. It has a sound analysis, able to analyze floating point operations too. It doesn't analyze dynamically allocated memory.
- Archer [XCE03] is a symbolic, path-sensitive analysis used to detect memory access errors. It is interprocedural and prunes impossible execution paths.
- SAFEcode / Data Structure Analysis [DKA06] uses a combined approach: it does static analysis, and where it cannot prove the safety of the code it inserts runtime checks. It works on LLVM bitcode.
- Calysto [BH07] has an interprocedural analysis, having both NULL pointer and bounds analysis. It works on LLVM bitcode.
- KLEE [CC08] is a tool for symbolic execution: it executes the program symbolically, taking alternate execution paths, to achieve high coverage. It generates test cases, and also finds memory access bugs. It works on LLVM bitcode.
- Clang static analyzer [cla] is integrated into the Clang compiler, and does the analysis based on information from the Abstract Syntax Tree. It has intraprocedural analysis to detect various bugs (null dereference, division by zero, API violation). It doesn't have an interprocedural analysis yet.

All of these tools (except clang) provide in-depth verification, they are very useful for performing a security audit of a project for example. However they are less useful in the day-to-day compile-test-debug cycle.

Our tool differs from the above by setting different goals: verification speed should be fast, thus easy to integrate into the test process of a program. Obviously we need to sacrifice the accuracy of the analysis to achieve this, we only attempt to analyze a certain class of buffer overflows, but do so interprocedurally. This class essentially includes memory accesses that have a predicate guaranteeing their safety , and both the index and bounds of the access are part of such a predicate; for example if there is an explicit bounds check ensuring that the pointer dereference instruction is never executed if the pointer is out of bounds.

Chapter 2

Theoretical overview

2.1 Intermediate representation

2.1.1 SSA form. Basic blocks. Def-use chains

Static Single Assignment form is a representation where each variable is assigned exactly once. A single variable from the original source code can be split into multiple SSA values.

Let's consider as an example Algorithm 2.1.

Algorithm 2.1 Code to be transformed into SSA form

```
$x \leftarrow 42$
$x \leftarrow x - 4$
**if** $x \leq 30$ **then**
    $y \leftarrow x * 4$
    $w \leftarrow y$
**else**
    $y \leftarrow x - 5$
**end if**
$w \leftarrow x - y$
$z \leftarrow x + y$
```

Transformed to SSA form the algorithm becomes in Algorithm 2.2.

Algorithm 2.2 Code in SSA form

```
$x_1 \leftarrow 42$
$x_2 \leftarrow x_1 - 3$
**if** $x_2 \leq 3$ **then**
    $y_1 \leftarrow x_2 * 4$
    $w_1 \leftarrow y_1$
**else**
    $y_2 \leftarrow x_2 - 5$
**end if**
$y_3 \leftarrow \phi(y_1, y_2)$
$w_2 \leftarrow x_2 - y_3$
$z_1 \leftarrow x_2 + y_3$
```

The ϕ instruction joins multiple SSA values, in our example if control flow comes from the basic block where y_1 exists, then the result shall be y_1. Otherwise if control flow comes from the basic block where y_2 exists, the result shall be y_2.

Temporary variables (like y_1, y_2) are added when a variable that already has a value has assigned a new value, or when the value assigned to a variable depends on control flow.

To understand where ϕ functions need to be placed we need the concept of *dominance frontiers*: a node B is in the dominance frontier of a node A if A does not strictly dominate B, but does dominate some immediate predecessor of B (possibly itself if A is the immediate predecessor of B).

Dominance frontiers are the exact places at which we need ϕ functions: if node A defines a certain variable, then that definition alone (or redefinitions) will reach every node A dominates. Only when we leave these nodes and enter the dominance frontier must we account for other flows bringing in other definitions of the same variable. Moreover, no other ϕ functions are needed in the control flow graph to deal with A's definitions, and we can do with no less.

Def-use chains are explicit in SSA form: definitions are the only assignment of a variable, uses are all the expressions where the variable occurs.

An intermediate representation in SSA form is also split into basic blocks: a basic block is a sequence of instructions that has one entry point, and one exit point with no other jumps/branches in between. A basic block can be terminated by one of:

- a branch instruction (conditional or unconditional). The target of a branch instruction is another basic block.
- return instruction (to caller)
- an instruction that may throw an exception
- noreturn function calls (`exit()`, `abort()`, `assert_fail()`, ...)

An important property of SSA form is that definitions dominate all uses of the variable: the basic block containing the definition of the variable must dominate all the basic blocks where the variable is used. Basic block A dominates basic block B, if and only if all paths from the root of the CFG to basic block B contain basic block A.

One can easily build a control flow graph using this intermediate representation: nodes are basic blocks, children of nodes are successors of the basic-block, parents of nodes are predecessors of a basic-block.

2.2 Chains of Recurrences. Scalar Evolution

Consider the **for** loop in Algorithm 2.3.

Algorithm 2.3 Evolution of variables in a for loop

```
$j \leftarrow 4$
for $i = 1$ to 100 step 3 do
    $j \leftarrow i + 8 + j$
end for
```

We observe that i and j take the following values: $i = 1, 4, 7, 10, 13, \ldots, 97$
$j = 4, 13, 25, 40, 58, \ldots, 1885$

We now use Newton's formula for interpolation to get a formula for j, the coefficients are taken from Table 2.1.

Table 2.1: Differentiation table for j

k	0	1	2	3	4
Δ_k^0	4	13	25	40	58
Δ_k^1	9	12	15	18	
Δ_k^2	3	3	3		
Δ_k^3	0	0			

The formula is $j(n) = \sum_{i=0}^{p} \Delta_0^i \binom{n}{i}$, Δ_0^i are coefficients of a chain of recurrence. Chains of recurrences can be used to represent scalar functions [BWZ94, JBB02, Pop04]. For Algorithm 2.3 on the facing page we have:

$$\begin{aligned} f(x) &= 4\binom{x}{0} + 9\binom{x}{1} + 3\binom{x}{2} \\ &= 4 + 9x + 3\frac{x(x-1)}{2} \\ &= \frac{3}{2}x^2 + \frac{15}{2}x + 4 \end{aligned}$$

x represents the number of iterations, thus on entry to the loop we have $j = f(0)$, after the first iteration $j = f(1)$, and on exit: $j = f(33)$.

The syntax for the chains of recurrence of j is $\{4, +, 9, +, 3\}$: the numbers are the coefficients from the interpolation formula, and $+$ is the operation between the elements (a sum in this case).

Some example of other chains of recurrences are:

$$\begin{aligned} \{5, +, 2\} &\rightarrow f(x) = 2x + 5 \\ \{1, *, 2\} &\rightarrow f(x) = 2^x \\ \{1, *, 1, +1\} &\rightarrow f(x) = x! \\ \{init, +, update\} &\rightarrow f(x) = init + \sum_{j=0}^{x-1} update \\ \{init, +, b_0, +, \ldots, +, b_{n-1}\} &\rightarrow init + \sum_{j=0}^{x-1} b_0 \sum_{k=0}^{n-1} b_k * x^k \end{aligned}$$

Once we have a chain of recurrence expression for a variable, we can do operations directly on this form to find out the chain of recurrence expression for other variables.

Extracting the chains of recurrence expressions for the example in Algorithm 2.3 on the preceding page can be done if its transformed into SSA form as shown in Algorithm 2.4 on the following page

The expression for i is $i = \{1, +, 3\}$, because it is of the form $i = phi(init, i + update)$. The expression for j is $j = \{4, +, 8 + i\} = \{4, +, 8 + \{1, +, 3\}\} = \{4, +, 9, +, 3\}$.

The variables in the Program 1 on the next page have the following expressions as chains of recurrences: $i = \{0, +, 1\}$, $j = \{0, +, 1, +, 1\}$ Using binomial coefficients we

Algorithm 2.4 Evolution of variables in a loop SSA form

```
$j \leftarrow 4$
for do
    $i \leftarrow \phi(1, i.0)$
    $j \leftarrow \phi(4, j.0)$
    $i.0 \leftarrow i + 3$
    $j.0 \leftarrow j + 8 + i$
end for
```

Program 1 Scalar Evolution example

```c
int foo(int *a, unsigned n)
{
    int *p = a;
    unsigned s = 0, j=0;
    for (unsigned i=0; i<n;i++) {
        j += i;
        s += a[j];
    }

    int *r = a + j;
    return *r;
}
```

can calculate the exit value of j: $\left[\frac{n_{33}\cdot(n-1)_{33}}{2}\right]_{32}$. In the previous formula $_32$ and $_33$ are the number of bits used to represent the respective values (because multiplication can overflow 32 bits, 33 bits are needed to correctly calculate the result).

2.3 Global Value Numbering

Introduced in [AWZ88], Global Value Numbering is an algorithm to determine equivalence of variables in a program. It is different from local value numbering, because it finds equivalences beyond the limits of a single basic-block. The algorithm works by assigning a value number to variables and expressions; variables and expressions that are equivalent get the same value number assigned. A value graph is built, and congruent nodes are found and assigned the same value number.

Algorithm 2.5 Global Value Numbering example

```
$a \leftarrow c \cdot d$
$e \leftarrow c$
$f \leftarrow e \cdot d$
```

In example Algorithm 2.5 Global Value Numbering will assign the same value number to c and e, determine that the nodes for a and f are equivalent (they perform the same operation, and their operands have same value number) and assign a and f same value number too. An example value numbering would be: $a \to 0, c \to 1, d \to 2, e \to 1, f \to 0$.

2.4 Alias Analysis

Pointers in a program can point to the same region of memory. If they do we can't interchange loads/stores to different pointers because they may store/read from the same location.

Thus it is important to be able to analyze whether two pointers may alias or not. There can be three situations: the pointers never alias (they point to distinct memory locations at all times), the pointers always alias (they point to the same memory location at all times), the pointer may alias (in some situations they may point to same location, in others they may not). The *may alias* is given if the memory region pointed by the 2 pointers overlap, but don't have same starting point (for example portions of an array with different starting index).

Also if our analysis doesn't have enough information to prove a *must alias* or *no alias* situation, then a conservative answer is *may alias* between two pointers.

There are many algorithms for alias analysis, with various degrees of compromise between speed and accuracy.

2.5 Scalar Replacement of Aggregates

The purpose of this transform is to replace structures and structure fields with individual SSA variables, if we can prove they don't alias any other field.

2.6 Dominator Tree

In Section 2.1.1 we've seen that a fundamental property of SSA is that value definition dominate all of their uses. To check this we can build the control flow graph out of the program's basic blocks, and we can establish the dominance property as follows: basic block A dominates basic block B, if and only if all paths from the root of the CFG to basic block B contain basic block A.

Dominance can be computed by traversing the graph depth-first, and numbering nodes with DFSin, and DFSout numbers representing the moment we started traversing a node, and the moment we finished traversing the node.

Some related concepts:

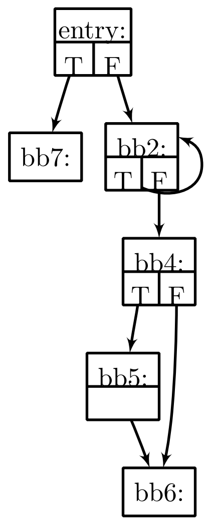
- a node always dominates itself by definition
- A strictly dominates B, if A dominates B, and A != B
- the immediate dominator of a node A that strictly dominates A, but doesn't strictly dominate any other node that strictly dominates A
- A dominator tree is a tree where each node's children are those nodes that it immediately dominates

An algorithm to compute dominators is given by [LT79].

For the example in Figure 2.6 on the next page the dominator tree is shown in Figure 2.1 on the following page.

```
struct bar { char *p; unsigned size; };
struct bar *allocate(unsigned siz)
{
    struct bar *b = malloc(sizeof(*b));
    if (b) {
        b->size = siz;
        b->p = malloc(siz);
        if (b->p) return b;
        free(b);
    }
    return NULL;
}
int foo(struct bar *b)
{
    if (!b->size) return -1;
    unsigned s=0;
    for (int i=0;i<b->size;i++)
        s += foobar(i);
    process(b->p, b->size, s);
    b->p[b->size - 1]++;
    unsigned pos = foobar(0);
    if (pos < b->size)
      b->p[pos] = 0;
    process(b->p, b->size, 0);
    return 0;
}
int main(int argc, char *argv[]) {
    struct bar *b = allocate(argc);
    if (!b) return -1;
    int ret =  foo(b);
    free(b);
    return ret;
}
```

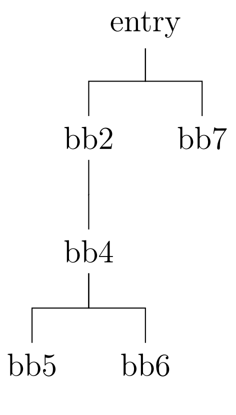
(a) Example



CFG for 'foo' function

(b) Control flow graph

Figure 2.1: Dominator-tree



2.7 Inlining

Inlining copies the code of a function into the caller, replacing arguments with actual parameters. There are some situations where inlining is not possible:

- vararg functions: theoretically this would be possible, but requires lots of changes to the inlined function
- indirect function calls, if constant propagation hasn't determined what function is called

Inlining has the benefit that it optimizes away a function call, but more importantly allows constant-propagation for its arguments, and allows optimizers to discover more relations between its arguments.

Inlining is done bottom-up on the call graph, usually by doing a cost-analysis to determine whether inlining is feasible. Small functions are always candidates for inlining, however too much inlining can increase code size, increase compilation time, and have no benefit at runtime.

2.8 Slicing

Slicing was introduced by [Wei81]. A static slice S consists of all the statements that may influence the slicing criterion. The slicing criterion is a point in the program and a set of variables that we want to keep.

A slice is created by analyzing the data- and control-flow dependencies in the program. The slice will preserve the variables in the slicing criterion, and all the instructions that can affect those variables.

Chapter 3

Algorithm design

3.1 Algorithm

3.1.1 Clarifying requirements

In order to clarify the goal of our tool let's the consider the example in Program 2 on the next page.

First of all let's see what errors might occur in such a program. We observe that the program does dynamic memory allocation and deallocation (lines 4, 7 and 10), a function dedicated for allocating `struct bar` (`allocate()`, lines 2 - 13), a for loop (lines 21 - 24), pointer arithmetic, and memory accesses using these pointers (lines 26, 29).

Such a program could have the following category of errors:

- NULL pointer dereference. If there hadn't been the check at line 5, a failed allocation that returned *NULL* could have caused such a bug.

- Memory leak. A dedicated constructor-like allocation function like `allocate()` could cause memory leaks, if there hadn't been the deallocation at line 10.

- Infinite loop. The counter at line 21 is a signed type, however the `b->size` is an unsigned number. If `b->size` has its MSB set, then we have an infinite loop. However it is not possible to determine for all loops whether they are finite, since it would be equivalent to solving the Turing halting problem.

- Pointer arithmetic wraparound. Not the case in this program, but compilers can assume that $pointer + unsignedvalue > pointer$ is always true. If we add a high enough number (with MSB set) the resulting pointer can be lower than the original one. The ISO C99 language standard [ISO07] considers this "undefined behavior", and allow compilers to consider it as if it never happens. Such constructs should be avoided in C code as described in rules ARR38-C and MSC15-CPP in [Sea08].

- Bounds overflow when dereferencing a pointer. At lines 26, and 29 we must ensure that after the pointer arithmetic the resulting pointer is still within bounds of a valid memory zone (the same as the original pointer).

The goal of this thesis is to detect the latter category of errors, bounds overflow. We don't aim to detect the other categories of errors. Pointer arithmetic wraparound, or computing an out-of-bounds pointer is not considered a bug, unless the resulting pointer is dereferenced.

Program 2 Interprocedural example

```
struct bar { char *p; unsigned size; };
struct bar *allocate(unsigned siz)
{
    struct bar *b = malloc(sizeof(*b));
    if (b) {
        b->size = siz;
        b->p = malloc(siz);
        if (b->p)
            return b;
        free(b);
    }
    return NULL;
}
int process(char *p, unsigned s, unsigned v);
int foobar(int a);
int foo(struct bar *b)
{
    if (!b->size)
        return -1;
    unsigned s=0;
    for (int i=0;i<b->size;i++)
    {
        s += foobar(i);
    }
    process(b->p, b->size, s);
    b->p[b->size - 1]++;
    unsigned pos = foobar(0);
    if (pos < b->size)
      b->p[pos] = 0;
    process(b->p, b->size, 0);
    return 0;
}
int main(int argc, char *argv[]) {
    struct bar *b = allocate(argc);
    if (!b)
        return -1;
    int ret =  foo(b);
    free(b);
    return ret;
}
```

3.1.2 Dereferencing and reachability

Program 3 Out-of-bound pointers vs. dereferencing

```
   int foo(FILE* f, size_t size) {
 2     char *x, *p, *end;
       x = malloc(size);
 4     if (!x || fread(x, 1, size, f) != size)
           return -1;
 6     p = memchr(x, ':', size);
       if (!p)
 8         return -1;
       p += 4;
10     if (p >= x + size)
           return -1;
12     bar(*p);
       free(x);
14     return 0;
   }
```

To understand the distinction between comparing out-of-bounds pointers, and dereferencing them consider Program 3. If the check at line 10 wouldn't exist, then the pointer dereference at line 12 could be out-of-bounds resulting in a crash at runtime, or leaking of data from other variables. Our tool should report this as a bug.

However with the check at line 10 in place our tool reports no bounds violation at line 12, because when the code at line 12 is *reached* the pointer is known to be valid. The pointer arithmetic at lines 9 and 10 can result in an out-of-bounds pointer, or even pointer overflow. Our tool doesn't check for this per se, neither does it warn about it.

3.2 Analysis used

Because we want to find errors in other programs we need to ensure that we are treating the semantics of the program correctly. Rather than building a tool from scratch, we use frameworks and libraries developed for this purpose. We have to reason about array indices and pointers: we have to find memory allocations, reason about pointer offsets dereferencing a pointer that is out-of-bounds. One choice would to analyze the abstract syntax tree of the program, for example by using the CIL library [NMRW02] , or the Clang compiler [cla]. Another option would to use an existing compiler, and analyze its intermediate representation. We chose LLVM [LA04, Lat02] for this purpose due to many reasons:

- LLVM-GCC is a production quality compiler, able to compile many real world applications
- The intermediate representation is lower level than C, easier to analyze, having some well defined instructions and expressions. Also type conversions are explicit (such as floating point to integer, sign-, and zero-extension, truncation, pointer casts). Integers are not signed/unsigned, rather operations are, thus making it easier to analyze expressions that mix variables of different sign.

- The intermediate representation is in Static Single Assignment form (memory is not in SSA form, only variables)
- It has data- and control flow analysis
- It has transforms and optimizations that are usually available in a modern compiler
- There are other tools that use LLVM for analysis: Calysto [BH07], KLEE [CC08], etc.
- There is a friendly open-source community

Our work has as primary goal the detection of memory access bugs by static analysis. We do not aim to perform an exhaustive analysis of the program, or to do symbolic execution. We don't generate constraints and propagate them interprocedurally, but rather use slicing combined with compiler optimizations and transforms.

To be able to analyze the validity of memory accesses we should track pointers in the program to find out whether they point to a valid zone of memory. However there are many situations, commonly encountered in practice, when a pointer points to an invalid memory zone, but is never dereferenced. Our tool doesn't consider this a bug, we have to also look for *dereferences* of possibly-out-of-bounds pointers.

To achieve this we have to track:

- the size of allocated memory
- other pointers that can point to same memory zone (aliasing)
- the value of a pointer after pointer arithmetic
- the validity of a pointer after pointer arithmetic
- the places where a pointer is dereferenced
- the constraints that need to be satisfied for the dereference to be executed (reachability conditions).
- how values evolve within and outside of loops

To track the evolution of values inside loops (with regard to the loop's induction variable), and to track the result of pointer arithmetic we use the *Scalar Evolution* [Pop04] framework from LLVM. This obtains an expression for each value, represented as a chain of recurrences [BWZ94, JBB02]. For example a variable that starts at 3, and is incremented by 2 each cycle is represented as $\{3,+,2\}$. If another variable would sum this variable, its expression would be $\{0,+,3,+,2\}$. This expression can be transformed into a linear combination by using binomial coefficients (see Section 2.2).

With *Scalar Evolution* we solved the problem of tracking pointers intraprocedurally in common cases: we get a mathematical expression for each pointer that contains the base pointer on which pointer arithmetic was done, however the base pointer can be a ϕ node if *Scalar Evolution* was unable to determine how that value evolves in a loop (or if it not a ϕ node that evolves in the loop at all).

Frequently there is a predicate that ensures the validity of an access, such as the predicate on lines 18, and 28 in Program 2 on page 12. Without these the access could be invalid for certain values of the index, thus we have to take into account these predicates in our analysis. We don't take these predicates into account, only those that dominate the basic-block in which the memory access instruction is. We do lose some constraints this way that could possibly help us in proving a memory access, but a predicate that doesn't dominate the basic-block is useless: we need to know that no matter what execution path you choose, the access is always valid. The only predicates that are true regardless of the path are those that dominate the basic block.

To determine which basic blocks dominate another basic block we use the DominatorTree [LT79] built by LLVM.

To be able to check a memory allocation we need one more piece of information: the allocated size of the memory zone the pointer points to. We notice that pointer dereferences and allocations can be in different functions, such as in Program 2 on page 12. To know the size of pointers at the dereference point we need to do an interprocedural analysis.

First of all we find all memory allocations, calls to `malloc()`, `calloc()`, `realloc()` (memory can be allocated in other ways, such as by calling `mmap()`, `mmap2()`, `sbrk()` but we do not treat these). For example at line 7 `b->p` is allocated with size `siz` bytes. However the `siz` variable is not available at the moment of dereference (line 26), instead the size is stored to `b->size`. If we'd assume that the size of `b->p` is always stored in `b->size` we'd be wrong, because `b->size` could be changed without reallocating `b->p`. We can only assume that `b->size` still holds the size if there was no store to it (or call to a function that could store to it) between the time of its allocation, and use.

There is an analysis that does just that, but only intraprocedurally: Global Value Numbering [AWZ88]. To get something similar interprocedurally, we should either use an interprocedural Global Value Numbering analysis, or use inlining, such that the allocation and dereference end up in the same function.

Inlining would work fine on small programs, such as Program 2 on page 12, but for programs of medium and large size it would be impractical for multiple reasons:

- forced inlining can increase code size exponentially
- not all algorithms have linear complexity, and can't treat a huge number of basic blocks efficiently

Still inlining offers a simple solution for tracking pointer sizes. By using slicing [Wei81] we can reduce the size of a function, keeping only the instructions that influence the constraints we want to prove. As soon as we prove some more memory accesses, we can remove more memory accesses. Thus we don't need a full inlining step, but rather do inlining iteratively, mixed with a slicing & analysis step after each iteration.

3.3 Algorithm description

Having clarified what we need to achieve our goal, we can move on to the description of the algorithm itself. The architecture of the application is represented in Figure 3.1 on the following page.

We do an *on-demand* analysis, computing constraints only for memory accesses (load, store, memset, memcpy).

- We start from memory allocations, and mark all variables and structs that can contain pointers and their sizes. For example in Program 2 on page 12 we'd mark `b` in `allocate()`.

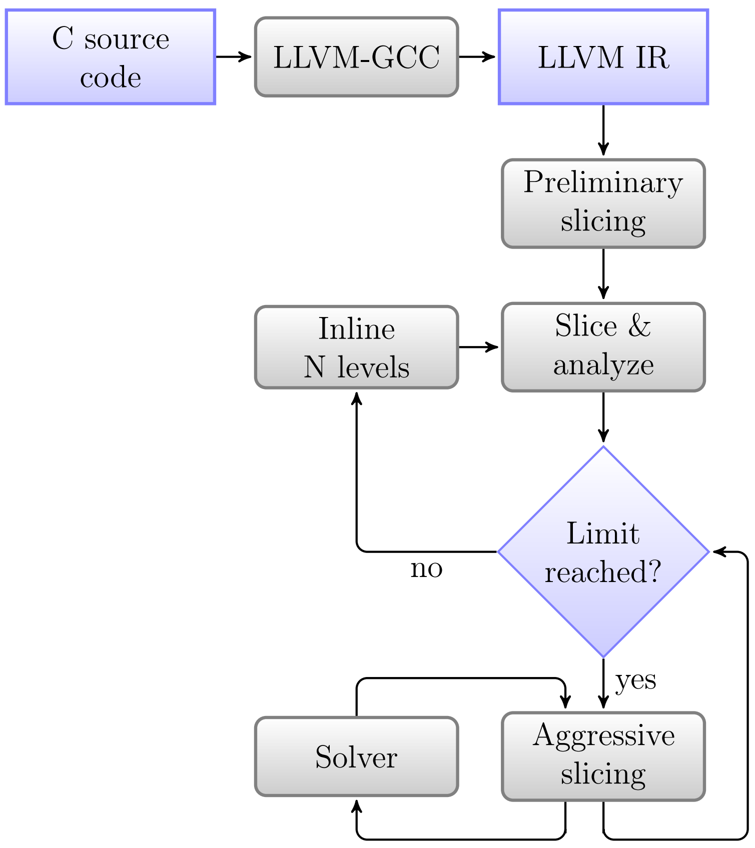


Figure 3.1: Application architecture

- We propagate these marks interprocedurally: if a function's parameter is marked in at least one call site, then we mark the function's argument too. This is done to ensure that the slicing step keeps all the variables needed for Global Value Numbering to ascertain the size of a pointer. In our example we mark the return value of `allocate()`, and `b` from `main()`.

- We obtain a *Scalar Evolution* expression for each memory dereference. A load/store would be deleted by Dead-Code Elimination. To prevent this we add a "dummy" user for each load, and we mark each store as volatile, because we haven't analyzed these yet, and we wouldn't want any of them to be deleted just yet. We do want to detect bounds overflow bugs in loads/stores that would otherwise be dead because they have no uses / they are invalidated by a later store. Of course we wouldn't want to detect bugs in code that is never reached.

- We perform a slicing step, with the slicing criterion to keep all the variables that are part of a *Scalar Evolution* expression obtained in the previous step, and all variables marked in the first interprocedural step. We replace calls to external functions with a call to a "dummy" function without parameters, and with a call to another "dummy" function for each of the parameters that are marked. Similarly for calls to internal functions we delete arguments that are not marked. In our example we replace the call to `foobar`, and the call to `process`. Through slicing we deleted the computation of `s` from the for loop, it didn't influence the dereference of `b->p`. Also we replace values stored to pointers that are not marked, with values obtained from dummy calls to free up the chain of computation for these values we aren't tracking.

- Simplify the result: the call to *dummy* in the for loop can be hoisted out of the loop (we don't care if its called once or more times since its a dummy function anyway). The loop itself can be deleted then, since it has no useful instructions anymore. We also run the Scalar Replacement of Aggregates transformation that *breaks up* a struct into multiple variables, one for each of its fields, this helps in analyzing each field of the struct independently, and to use Global Value Numbering on struct fields.

- Now we can attempt to verify the validity of some memory accesses: we obtain a *Scalar Evolution* expression for the allocation size, for the offset of the pointer inside the allocated zone, and for the length of the access (constant for a load/store, possibly variable for a memcpy/memset/...). For the access to be valid the following constraints must be satisfied: $offset < size$ && $offset+length <= size$ && $length < size$ (there is no need to check that the offset is positive: all values are unsigned, and negative values would be represented in 2's complement). We can rewrite the constraints as follows: **umax**(offset, size−1) == size−1 && **umax**(size, length)==size && **umax**(offset+length, size)== size) . If the constraint is decidable, we delete the dummy user / volatile marker, and if the result is invalid we show an error message. If the result is undecidable we move on to the next constraints that we want to prove. We are careful not to analyze the same access more than once if it's already been proven: we check that it has a dummy user / volatile attribute and check it only then. (such accesses shouldn't be in our slice at all, but they may be needed by memory accesses we haven't proved yet).

- We perform an inlining step: `foo->main`, `allocate->main`.

- We repeat the above steps until we reach the inlining limit.

- We look at the predicates that dominate the basic blocks that contain the memory access. We use a solver to refine the constraints obtained so far, by iteratively adding the predicates

- If we reach the inlining limit we perform a more aggressive slicing step, that might remove some values that could probably be useful in the future (but we don't have a choice, we have to reduce the size of the function somehow). We keep only undecided accesses, and ignore all values stored to them (even if the pointers are marked).

- Repeat the above steps until we reach inlining limit again.

- We stop, the accesses that are yet to be proven are either buggy in some situations (which we couldn't determine), or they are correct (but we simplified the code too much, and can't tell or we haven't reached the required inlining depth yet). We write a list of these into a log file, which the user can examine as low-priority possible bugs.

3.4 Slicing & simplification

3.4.1 LLVM intermediate representation

The result of compiling the `foo` function from Program 2 on page 12 to LLVM IR is presented below:

```
define i32 @foo(%struct.bar* %b) nounwind {
entry:
        tail call void @llvm.dbg.func.start({ }* bitcast (%llvm.dbg.
           subprogram.type* @llvm.dbg.subprogram23 to { }*))
        tail call void @llvm.dbg.stoppoint(i32 19, i32 0, { }*
           bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.
           compile_unit to { }*))
        %0 = getelementptr %struct.bar* %b, i64 0, i32 1
                    ; <i32*> [#uses=5]
        %1 = load i32* %0, align 8                    ; <i32> [#uses=1]
        %2 = icmp eq i32 %1, 0              ; <i1> [#uses=1]
        br i1 %2, label %bb7, label %bb2

bb2:                ; preds = %bb2, %entry
        %i.01 = phi i32 [ 0, %entry ], [ %tmp, %bb2 ]              ; <
           i32> [#uses=2]
        %s.02 = phi i32 [ 0, %entry ], [ %4, %bb2 ]                ; <
           i32> [#uses=1]
        %3 = tail call i32 @foobar(i32 %i.01) nounwind             ; <
           i32> [#uses=1]
        %4 = add i32 %3, %s.02              ; <i32> [#uses=2]
        %tmp = add i32 %i.01, 1             ; <i32> [#uses=2]
        %5 = load i32* %0, align 8                    ; <i32> [#uses=2]
        %6 = icmp ult i32 %tmp, %5                    ; <i1> [#uses=1]
        br i1 %6, label %bb2, label %bb4

bb4:                ; preds = %bb2
        tail call void @llvm.dbg.stoppoint(i32 26, i32 0, { }*
           bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.
           compile_unit to { }*))
        %7 = getelementptr %struct.bar* %b, i64 0, i32 0
                    ; <i8**> [#uses=4]
        %8 = load i8** %7, align 8                    ; <i8*> [#uses=1]
        %9 = tail call i32 @process(i8* %8, i32 %5, i32 %4) nounwind
                        ; <i32> [#uses=0]
        tail call void @llvm.dbg.stoppoint(i32 27, i32 0, { }*
           bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.
           compile_unit to { }*))
        %10 = load i8** %7, align 8                   ; <i8*> [#uses=1]
        %11 = load i32* %0, align 8                   ; <i32> [#uses=1]
        %12 = add i32 %11, -1               ; <i32> [#uses=1]
        %13 = zext i32 %12 to i64                     ; <i64> [#uses=1]
        %14 = getelementptr i8* %10, i64 %13                ; <i8*> [#
           uses=2]
        %15 = load i8* %14, align 1                   ; <i8> [#uses=1]
        %16 = add i8 %15, 1                ; <i8> [#uses=1]
        store i8 %16, i8* %14, align 1
        tail call void @llvm.dbg.stoppoint(i32 28, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        %17 = tail call i32 @foobar(i32 0) nounwind             ; <i32> [#uses=2]
        tail call void @llvm.dbg.stoppoint(i32 29, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        %18 = load i32* %0, align 8              ; <i32> [#uses=2]
        %19 = icmp ugt i32 %18, %17              ; <i1> [#uses=1]
        br i1 %19, label %bb5, label %bb6

bb5:            ; preds = %bb4
        tail call void @llvm.dbg.stoppoint(i32 30, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        %20 = load i8** %7, align 8              ; <i8*> [#uses=1]
        %21 = zext i32 %17 to i64                ; <i64> [#uses=1]
        %22 = getelementptr i8* %20, i64 %21             ; <i8*> [#uses=1]
        store i8 0, i8* %22, align 1
        %.pre = load i32* %0, align 8            ; <i32> [#uses=1]
        br label %bb6

bb6:            ; preds = %bb5, %bb4
        %23 = phi i32 [ %.pre, %bb5 ], [ %18, %bb4 ]            ; <i32> [#uses=1]
        tail call void @llvm.dbg.stoppoint(i32 31, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        %24 = load i8** %7, align 8              ; <i8*> [#uses=1]
        %25 = tail call i32 @process(i8* %24, i32 %23, i32 0) nounwind           ; <i32> [#uses=0]
        tail call void @llvm.dbg.stoppoint(i32 32, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        tail call void @llvm.dbg.region.end({ }* bitcast (%llvm.dbg.subprogram.type* @llvm.dbg.subprogram23 to { }*))
        ret i32 0

bb7:            ; preds = %entry
        tail call void @llvm.dbg.stoppoint(i32 32, i32 0, { }* bitcast (%llvm.dbg.compile_unit.type* @llvm.dbg.compile_unit to { }*))
        ret i32 -1
}
```

The LLVM IR above can be understood based on the following concepts:

- `%<number>` and `%<name>` are SSA values

- `i32` is a 32-bit integer
- the type of SSA values is written before their name when they are used, for example: `i32 %0`
- calls to `@llvm.dbg.*` represent debug information (source line, original variable names, function name) and are not real calls
- `getelementptr` is the instruction for pointer arithmetic: it's first operand is the pointer, and the subsequent operands are indexes into that pointer. For example `getelementptr %b, 0, 1` means: dereference b at offset 0, then give me second element (field of a struct).
- `load` will dereference its pointer operand, load the value, and create a new SSA value
- `icmp` is the integer compare instruction, result is a boolean (`i1`)
- `br` is a branch instruction, first operand is the condition, second and third operands are the branch targets
- `phi` is the ϕ function from the SSA representation, but it also specifies the predecessor basic block coresponding to incoming values.
- `call` is a function call
- `add` is integer addition
- `ult, ugt` are the integer compare predicates: unsigned less than, unsigned greater than
- `zext` is zero extension
- `store` will store the first operand to the pointer operand (second)
- `bitcast` is an explicit cast introduced, corresponding to implicit C casts
- `ret` will return from the current function with the specified value as a result

The values and calls to uninteresting functions have been replaced with calls to dummy functions.

3.4.2 Dummy functions

We use 3 types of dummy functions:

dummy.sideeffect a function without any parameters, to model unknown side-effects (like writing to global variables). This call can be hoisted out of loops though, and multiple calls to it merged into one call.

dummy.read a function with a read-only parameter. It is added just to keep loads live. To prevent the dummy call from being deleted by dead code elimination we don't mark the function as read-only (since then our dummy call could be removed), but rather tell the mod-ref analysis that the function doesn't modify any value. One call to this function per value is enough, others can be eliminated.

Program 4 Sliced program

```
int foo(struct bar *b)
{
    if (!b->size) return -1;

    for (int i=0;i<b->size;i++)
       dummysideeffect();
    dummywrite(b->p);
    b->p[b->size - 1]++;
    unsigned pos = dummysideeffect();
    if (pos < b->size)
      b->p[pos] = 0;
    dummywrite(b->p);
    return 0;
}
```

Program 5 Sliced program with loop eliminated

```
int foo(struct bar *b)
{
    if (!b->size) return -1;
    dummywrite(b->p);
    b->p[b->size - 1]++;
    unsigned pos = dummysideeffect();
    if (pos < b->size)
      b->p[pos] = 0;
    dummywrite(b->p);
    return 0;
}
```

dummy.write a function that writes to its parameter.

A function call can be replace by a call to *dummy.sideeffect*, and *dummy.write* for its pointer parameters.

3.4.3 Simplifying program transforms

We run standard transforms to simplify the code:

- Scalar Replacement of Aggregates
- Global Value Numbering
- Loop Invariant Code Motion
- Dead Code Elimination
- Loop Deletion

3.5 Solving constraints

Program 6 Access offset and length

```
1 int foo(unsigned x, unsigned y, unsigned n)
  {
3   int32_t *base = malloc(n);
    if (base) {
5     int32_t *p = base + x;
      bar(*p);
7     memset(p+1, 0, y);
    }
9 }
```

When we want to prove an access, we have to know the `offset` of the pointer into the allocated zone, the `size` of the allocated zone, and the `length` of the access.
Consider the example in Program 6. For the dereference of `*p` at line 6 we have:

- $\text{offset} = (\textbf{char}*)\text{p} - (\textbf{char}*)\text{base} = 4*\text{x}$
- $\mathit{length} = 4$
- $\mathit{size} = n$

For the `memset` at line 7 we have:

- $\text{offset} = (\textbf{char}*)(\text{p}+1) - (\textbf{char}*)\text{base} = 4*\text{x} + 4$
- $\mathit{length} = y$
- $\mathit{size} = n$

Generally let's consider the following:

- Our pointer at offset bytes inside an allocated zone, that starts at base:
  char *base = malloc(size); p = base + offset
- We have a memory access of length length bytes: memset(p, 0, length)

Then we know the following has to hold for a valid access: base[i]; $i \in [0, n), i \in [offset, offset + length)$, which means that the following must hold true: $offset < n, offset + length <= n, length < n$.

We know that the access is always valid if:

- **umax**(offset,n−1)==n−1
- **umax**(n, length)==n
- **umax**(offset+length, n)==n

We know that the access is always invalid if:

- **umax**(offset,n−1)==offset && n−1 != offset
- **umax**(n, length)==length && n != length
- **umax**(offset+length, n)==n && offset+length!=n

If neither holds true then we haven't got enough information to decide.

3.5.1 Intraprocedural analysis

The easiest situation is when both the allocation and the dereference are in the same function. In this situation it is easy to determine the size of the allocation, the offset of the pointer inside that, and the validity of the access.

This is the situation in Program 3 on page 13, which can be handled easily by just looking at the data flow graph, and computing what value p has at line 10 and 12. However we need to be able to handle more complicated situations.

First we need to be able to reason about values inside loops, arithmetic on values, and loop-exit values of variables. LLVM provides the Scalar Evolution framework that does this analysis. If the expression involves only pointer arithmetic with addition/subtraction and multiplication with constant, then having the SCEV representation of a pointer is enough to reason about some common cases.

We'll need a solver to prove the reachability of the memory access we are investigating, see Section 3.5.5. Consider the example in Program 7 on the next page. The SCEV expression for j is {0,+1,+,1} and the exit value of j is ((**trunc** i33 (((**zext** i32 (−1 + %n)**to** i33)∗ (**zext** i32 %n **to** i33)) /**u** 2) **to** i32) + %n) . That is equivalent to this mathematical formula $\frac{n \cdot (n-1)}{2}$ by doing the operations on 33-bits, then truncating to 32-bits. In the previous formula `zext` means zero extend (from 32 bits to 33 bits), `trunc` means truncate (from 33 bits to 32 bits), and `%n` is the variable `n`.

Now that we know how to reason about values inside and outside loops, we could be tempted to think that we're done with intraprocedural analysis. This is not the case. Consider the example in Program 8 on the following page.

We want to reason about the safety of the store on line 12. However we notice that we allocate `n` bytes, but we compare against `b.size`. The simple static analysis based solely on SCEV wouldn't be able to decide on the safety. There are two well-known

Program 7 Loop example

```
int foo(unsigned n, int *a) {
    int i;
    for (i=0;i < 100; i++, j+=i) {
        a[i]=0;
    }
    return j;
}
```

Program 8 Aggregates

```
struct bar {
    char *p;
    unsigned size;
};
int foo(int n) {
  struct bar b;
  b.size = n;
  if (!(b.p = malloc(n)))
      return -1;
  unsigned pos = foobar();
  if (pos < b.size)
      b.p[pos] = 0;
  ...
}
```

optimization techniques that can be applied in this case: Global Value Numbering, and Scalar Replacement of Aggregates. GVN will ascertain that `b.size` and `n` are the same values, because there is nothing that could modify `b.size` between the store at line 7 and the compare at line 11 (if `foobar()` would have taken b as a parameter this were no longer true, since `foobar()` could have modified `b.size`). This would work even if `struct bar b` was a parameter to `foo`, and not a local a variable. Another possibility is to use scalar replacement of aggregates, which will break up `b` into 2 values, one for each field. It would then realize that `bsize` is really the same as `n`. This doesn't work if `struct bar b` is a parameter to `foo`. We invoke both of these transforms prior to our static analyzer pass, hence we are able to reason about Program 8 on the preceding page too.

3.5.2 Interprocedural analysis

The wrong way: inlining as the solution for everything

We are now confident that we can handle simple cases intraprocedurally, and this analysis is probably very fast too. Once we stray away from our intraprocedural toy example Program 8 on the facing page, we realize that memory allocation is usually done in a different function (constructors in C++).

Consider Program 9 on the next page. All of our previous transformations and analysis fail to determine anything useful about the dereference at line 26. We don't even know the size of `b->p`, let alone whether `pos` is out-of-bounds or not.

It is time to think about interprocedural analysis. Being familiar with compilers you think you could take a shortcut by compiling the code at a higher optimization level that inlines functions. If you read Section 2.7 you know that it's not long before inlining is not enough to analyze a real world program.

But it is too tempting not to try inlining such a simple case as Program 9 on the following page. Indeed inlining creates similar code to the one in Program 8 on the preceding page, and the tool developed previously for Program 8 on the facing page works.

Why not just let the compiler do all the work, and inline the entire program into main, simplify it, and analyze the output? For *small* programs that might work (assuming they don't contain recursion), but for medium sized projects this becomes impractical, because of:

- memory usage: inlining easily leads to exponential memory usage: rather small, but often called functions are inlined in multiple cases, the deeper it is in the call stack, the more duplicated code you get when inlining. At some point fully inlining a program creates a huge file, in the order of hundreds of megabytes.

- complexity of algorithms: such a function obtained via excessive inlining would contain a huge amount of basic blocks, and a huge number of instructions. Compilers don't cope well with huge numbers of basic blocks, and neither with huge amounts of instructions inside one basic-block. It is very likely that analyzing such a program would cause out of memory conditions, and non-linear execution times. A tool developed and tested on small examples will not scale to real world programs.

- useless waste of time: Although analyzing a fully inlined program would theoretically not need interprocedural analysis, it would be a waste of time to analyze the same

Program 9 First interprocedural example

```
struct bar {
    char *p;
    unsigned size;
};
struct bar *allocate(unsigned siz)
{
    struct bar *b = malloc(sizeof(*b));
    if (b) {
        b->size = siz;
        b->p = malloc(siz);
        if (b->p)
            return b;
        free(b);
    }
    return NULL;
}
int main(int argc, char *argv[]) {
    struct bar *b = allocate(argc);
    int ret =  foo(b, argc);
    free(b);
    return ret;
}
int foo(struct bar *b, int n) {
  unsigned pos = foobar();
  if (pos < b->size)
      b->p[pos] = 0;
  ...
}
```

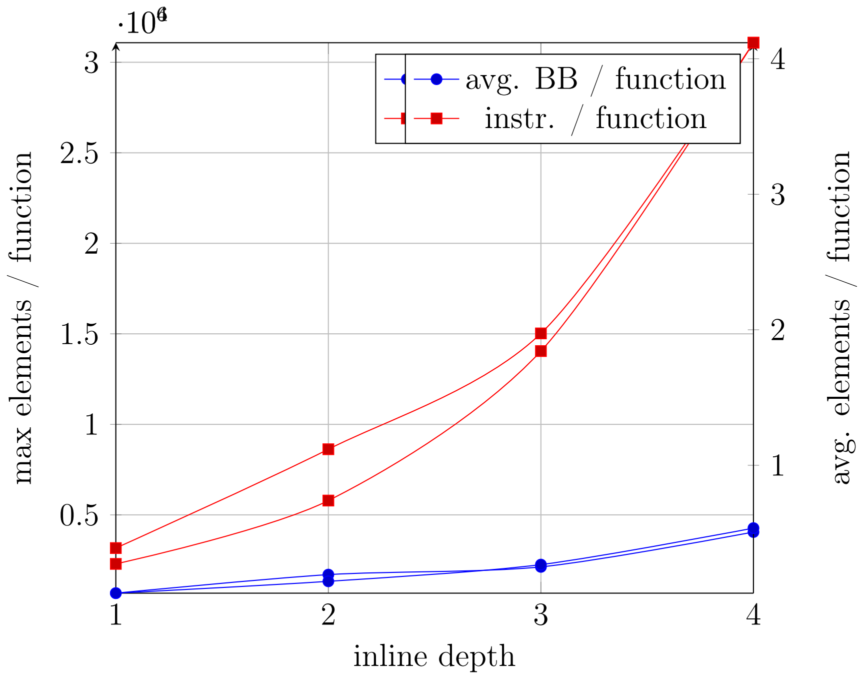


Figure 3.2: Inlined code statistics for depth-based inline limit

memory access in different contexts, if we have already proved it correct/incorrect once.

The graphs show three different inlining strategies: the default heuristics of LLVM (Figure 3.6 on page 29, Figure 3.7 on page 30), a simple basic-block-count limit based inlining (Figure 3.4 on the next page, Figure 3.5 on page 29), and a depth-based inlining (Figure 3.2, Figure 3.3 on the following page) applied to the ClamAV ® program.

The default heuristics of LLVM try to determine whether inlining would provide any benefits (such as propagating constants, etc.). The basic-block-count limit simply counts the number of basic block in the caller and callee, and prohibits inlining if limit is reached. The depth-based inlining forces inlining of all leaf calls (if possible) at N=1, the inlining of remaining leaf calls at N=2, and so on.

As the graphs show the depth based inlining results in exponential blowup in code size (Figure 3.2), and processing time (Figure 3.3 on the following page). Similarly the BB-count and heuristics based inlining methods are nonlinear (Figure 3.6 on page 29, Figure 3.7 on page 30, Figure 3.4 on the next page, Figure 3.5 on page 29). They also require huge amounts of memory (in excess of 4 GB) towards the end of the graph.

The execution time of transformations increase drastically on the resulting code, as shown in Figure 3.8 on page 30, Figure 3.9 on page 31, Figure 3.10 on page 31, Figure 3.11 on page 31.

It is obvious that inlining the entire program (or large portions thereof) is not a viable strategy for what we had in mind. We should reduce the size of the inlined functions for inlining to be practical. We can do this by a technique called slicing [Wei81].

Judging from the graphs we need to do slicing before inlining, and even after each inlining step to keep the code size manageable.

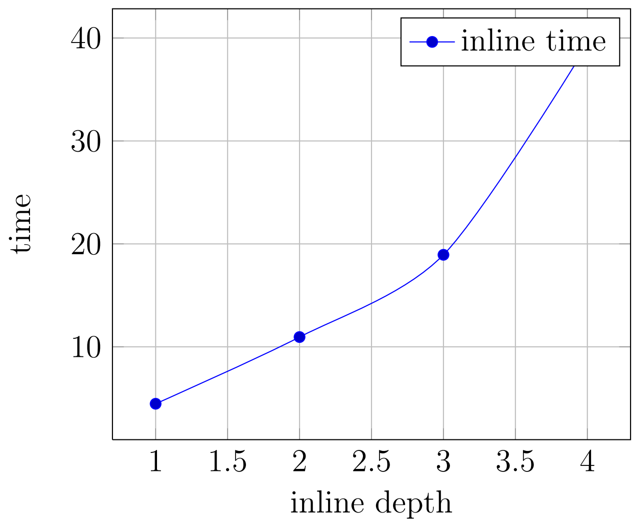


Figure 3.3: Time needed to inline the code with specified inline depth limit

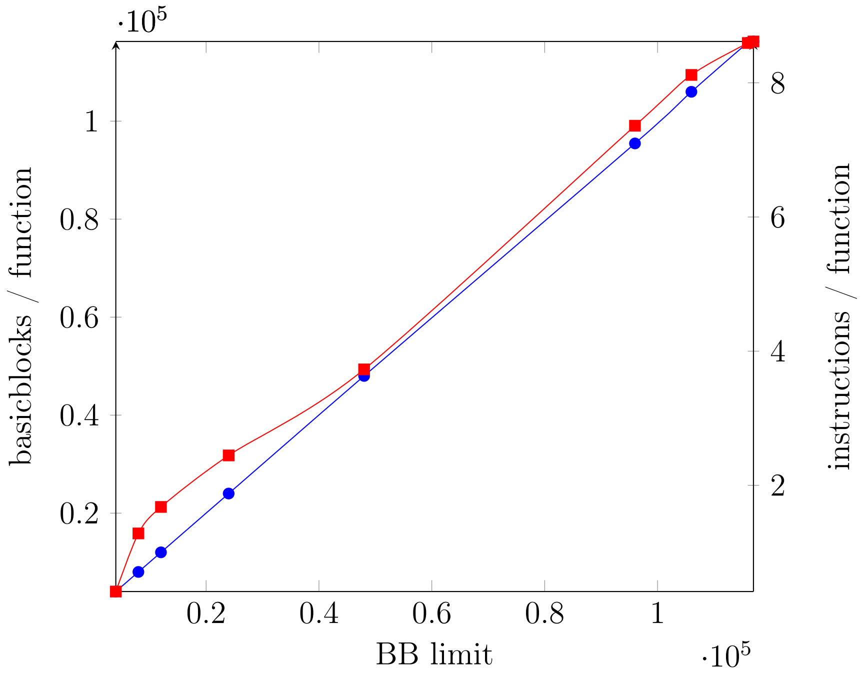


Figure 3.4: Inlined code statistics for BB based inline limits. ●: max BBs / function, ■: max instructions / function

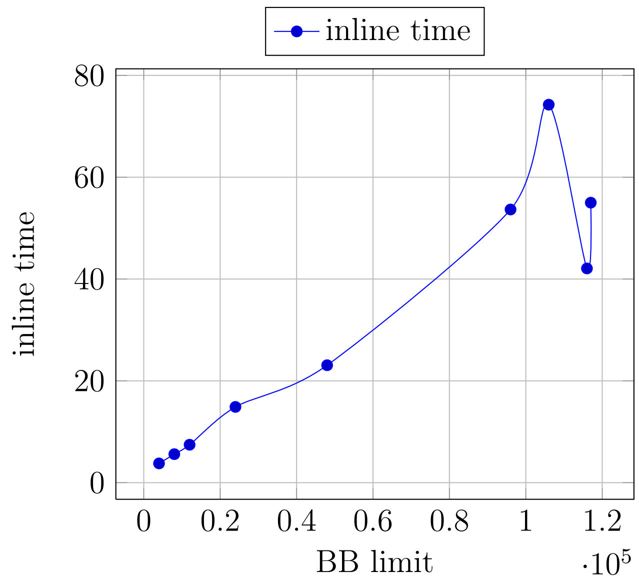


Figure 3.5: Time needed to inline code for BB based inline limits

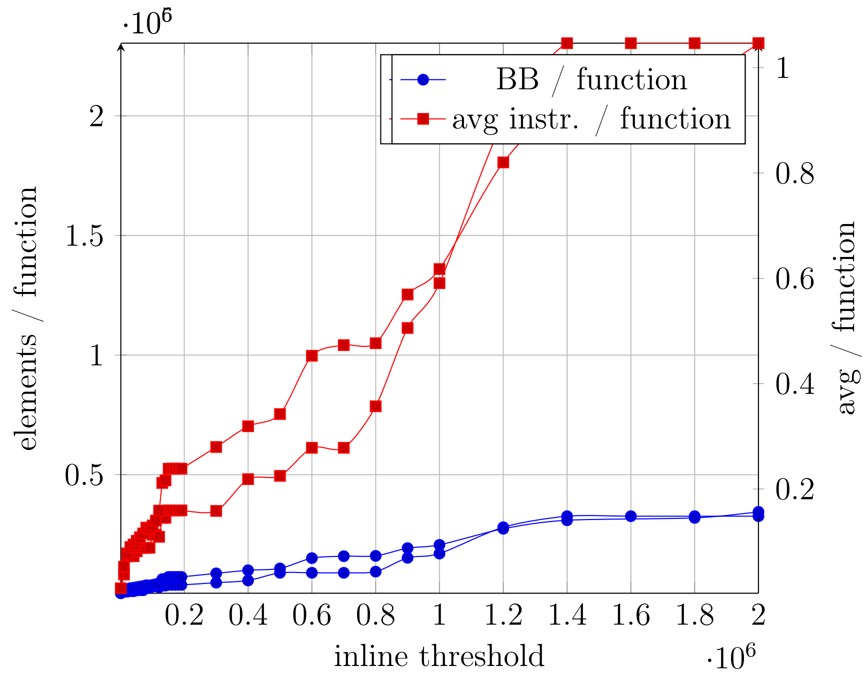


Figure 3.6: Code statistics for default heurstics based inline limits

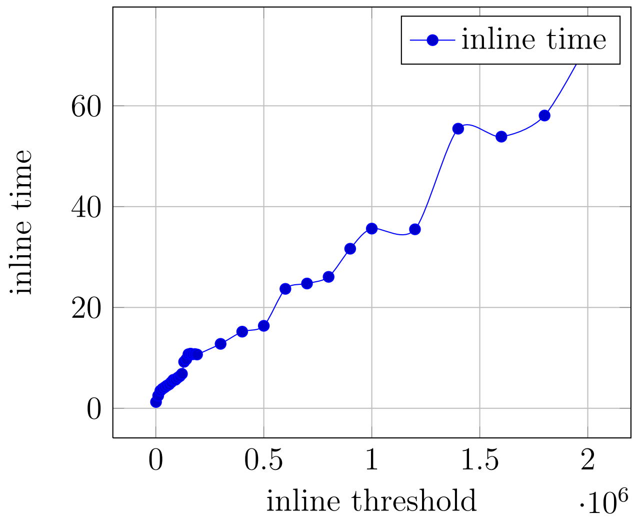


Figure 3.7: Time needed to inline code with default heuristic based limits

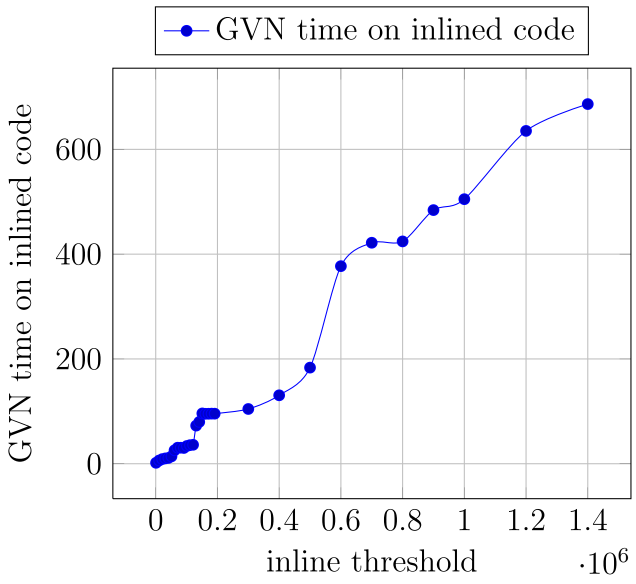


Figure 3.8: Time needed to run GVN on inlined code with specified heuristic limit

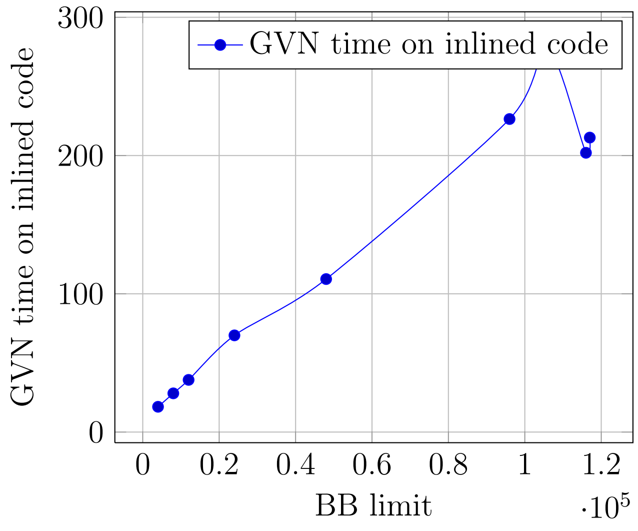


Figure 3.9: Time needed to run GVN on inlined code with specified BB limit

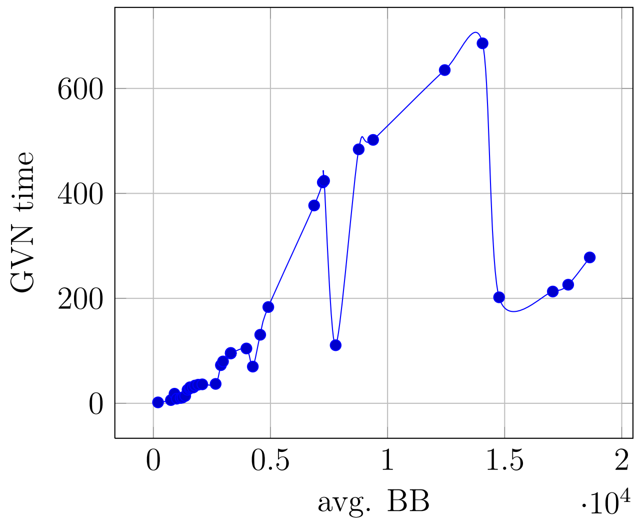


Figure 3.10: GVN time depending on average BB count

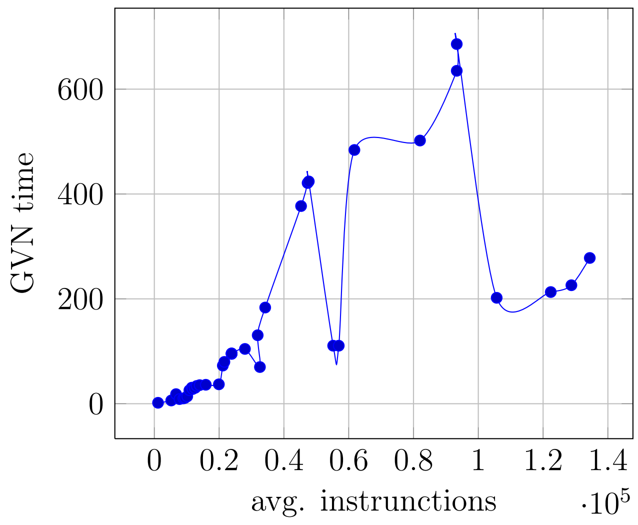


Figure 3.11: GVN time depending on average instruction count

3.5.3 Slicing

Let's think about example Program 8 on page 24 again. We were able to prove the validity of the access, yet we may have some other memory accesses we can't prove or disprove yet. We can tell the user to investigate those accesses, however it's often hard to figure out under what conditions the location in question is reached. Thinking about helping the user, we can create a slice (see Section 2.8) of the function containing the memory accesses we have proved invalid, and another slice containing the memory accesses we haven't proved yet.

To do this first we remove everything that is not interesting from the function, such as external function calls we don't know about, and computing parameters to these functions. We can also try to remove memory accesses we've already proved, unless their values are used by the memory accesses we haven't proved yet.

Program 10 Slicing example: C source

```
   struct bar {
 2     char *p;
       unsigned size;
 4 };
   int process(char *p, unsigned s, unsigned v);
 6 int foobar(int a);
   int foo(struct bar *b, unsigned n)
 8 {
       struct bar x;
10     if (!n)
           return -1;
12     x.size = n;
       x.p = malloc(n);
14     unsigned s=0;
       for (int i=0;i<n;i++)
16     {
           s += foobar(i);
18     }
       process(x.p, x.size, s);
20     x.p[x.size - 1]++;
       process(x.p, x.size, 0);
22     free(x.p);
   }
```

In Program 11 on the facing page the basic blocks are created from the following lines of the C source code:

bb1 line 10

bb2 line 13

bb2, bb3 lines 15 - 18

bb4 line 19 - end of function

Program 11 Slicing example: LLVM IR

```
        %struct.bar = type { i8*, i32 }

define i32 @foo(%struct.bar* %b, i32 %n) nounwind {
entry:
        %"alloca_point" = bitcast i32 0 to i32
        %0 = icmp eq i32 %n, 0
        br i1 %0, label %return, label %bb1

bb1:            ; preds = %entry
        %1 = zext i32 %n to i64
        %2 = call noalias i8* @malloc(i64 %1) nounwind
        br label %bb3

bb2:            ; preds = %bb3
        %3 = call i32 @foobar(i32 %i.0) nounwind
        %4 = add i32 %3, %s.0
        %5 = add i32 %i.0, 1
        br label %bb3

bb3:            ; preds = %bb2, %bb1
        %i.0 = phi i32 [ 0, %bb1 ], [ %5, %bb2 ]
        %s.0 = phi i32 [ 0, %bb1 ], [ %4, %bb2 ]
        %6 = icmp ult i32 %i.0, %n
        br i1 %6, label %bb2, label %bb4

bb4:            ; preds = %bb3
        %7 = call i32 @process(i8* %2, i32 %n, i32 %s.0)
           nounwind
        %8 = sub i32 %n, 1
        %9 = zext i32 %8 to i64
        %10 = getelementptr i8* %2, i64 %9
        %11 = load i8* %10, align 1
        %12 = add i8 %11, 1
        store i8 %12, i8* %10, align 1
        %13 = call i32 @process(i8* %2, i32 %n, i32 0) nounwind
        call void @free(i8* %2) nounwind
        ret i32 -1

return:         ; preds = %entry
        ret i32 -1
}
```

Program 12 Slicing example: first step

```
%struct.bar = type { i8*, i32 }

define i32 @foo(%struct.bar* %b, i32 %n) nounwind {
entry:
        %0 = icmp eq i32 %n, 0
        br i1 %0, label %return, label %bb1

bb1:
        %1 = zext i32 %n to i64
        %2 = call noalias i8* @malloc(i64 %1) nounwind
        br label %bb3

bb2:
        %3 = add i32 %6, %s.0
        %4 = add i32 %i.0, 1
        br label %bb3

bb3:
        %i.0 = phi i32 [ 0, %bb1 ], [ %4, %bb2 ]
        %s.0 = phi i32 [ 0, %bb1 ], [ %3, %bb2 ]
        %5 = icmp ult i32 %i.0, %n
        %6 = tail call i32 @.llvm.dummy.sideeffect()
        br i1 %5, label %bb2, label %bb4

bb4:
        call void @.llvm.dummy.write(i8* %2)
        %7 = sub i32 %n, 1
        %8 = zext i32 %7 to i64
        %9 = getelementptr i8* %2, i64 %8
        %10 = load i8* %9, align 1
        %11 = add i8 %10, 1
        volatile store i8 %11, i8* %9, align 1
        call void @.llvm.dummy.write(i8* %2)
        call void @free(i8* %2) nounwind
        call void @.llvm.dummy.read(i8 %10)
        ret i32 -1

return:
        ret i32 -1
}
```

Program 13 Slicing example: sliced LLVM IR

```
%struct.bar = type { i8*, i32 }

define i32 @foo(%struct.bar* %b, i32 %n) nounwind {
entry:
        %0 = icmp eq i32 %n, 0
        br i1 %0, label %return, label %bb1

bb1:                ; preds = %entry
        %1 = zext i32 %n to i64
        %2 = call noalias i8* @malloc(i64 %1) nounwind
        call void @.llvm.dummy.write(i8* %2)
        %3 = sub i32 %n, 1
        %4 = zext i32 %3 to i64
        %5 = getelementptr i8* %2, i64 %4
        %6 = load i8* %5, align 1
        %7 = add i8 %6, 1
        volatile store i8 %7, i8* %5, align 1
        %8 = tail call i32 @.llvm.dummy.sideeffect()
        call void @.llvm.dummy.write(i8* %2)
        call void @free(i8* %2) nounwind
        call void @.llvm.dummy.read(i8 %6)
        ret i32 -1

return:             ; preds = %entry
        ret i32 -1
}
```

In Program 13 on the previous page the only branch is the one checking for a nonzero `n`, the rest was merged into a single basic block `bb1`.

Keeping in mind that we want to do interprocedural analysis, we think about structs, and GVN again. We could eliminate stored values in our slice (by replacing with a dummy value), however that would prohibit GVN from finding out what the size of `b->p` is in Program 9 on page 26. We decide not to eliminate these yet.

Consider the example in Program 10 on page 32, which is compiled to the LLVM IR shown in Program 11 on page 33 (debug information has been removed for the sake of brevity. Normally debug info would be kept to allow nice messages to be printed with source:line references). We are only interested in the memory access at line 20. The loop between lines 15 and 18, and the call at line 19 doesn't affect it.

Instead of Program 11 on page 33 we'd want to see Program 13 on the preceding page. To do this transformation we need to:

- ascertain that the call to `foobar` doesn't influence the store at line 20
- ascertain that the call to `process` at line 19 doesn't influence the store
- remove the loop, and replace calls to `process` with dummy function call

That is too much to do in a single transformation step, so first we transform Program 11 on page 33 to Program 12 on page 34. To do this we have:

- replace the call to `foobar` with a call to .llvm.dummy.sideeffect
- replace the call to `process` with a call to .llvm.dummy.write, taking only the pointer parameter

Now we have to hoist the dummy call out of the loop. If this were a real function call we couldn't have done that, however for the dummy call it doesn't matter if we've got one or n calls to it.

With the dummy call hoisted out of the loop, we can delete the loop, because it does nothing useful anymore, LLVM provides a loop-deletion pass for this purpose. We have reached Program 13 on the previous page now.

You could be tempted to remove the call to dummy.write too, however in general you can't, because LLVM could think (rightfully so) that you are accessing uninitialized data, and "optimize away" your loads to undefined values.

3.5.4 Interprocedural propagation

Let's think about example Program 8 on page 24 again. We were able to prove the validity of the access, and we only needed to know that `b->size` is the size of `b->p`. We've also seen that a slice can help a user better understand why the code contains bugs. And of course we haven't completely forgotten about inlining yet. It turns out we can combine all these ideas into an algorithm that does interprocedural analysis, without the disadvantages of pure inlining described on 3.5.2 on page 25.

First we find all the memory allocations, this is easy: we just need to look for uses of the `malloc()`,`calloc()`,`realloc()` functions . . . literally. Function calls *use* the *function* object, so this is a simple def-use chain walking (see Section 2.1.1).

Next we could just propagate this information down the callgraph, and solve all memory access constraints. However these expressions become quite big soon (remember discussion about inlining, same happens with constraints), and showing the user a thousand line long constraint that is not satisfied won't help in solving the bug.

Recall the discussion about GVN/SROA, and realize that we can defer constructing such constraints. We only need to know which structs and values we have to *care about* in our analysis. By identifying the structs and values that can possibly store bounds information, we can use slicing to remove everything else.

So we have to iterate on the data flow graph starting from allocations, and mark structs, values, function parameters as *interesting* for our analysis. A better name would be *keep*, since we remove everything that is not *interesting*.

We start out by trying to prove accesses in a function, and mark the ones we've proved as not interesting, or not *keep*. For loads we do this by adding a dummy user: a call to a dummy function that only reads the value (careful that DCE doesn't delete it). For stores we do this by simply marking the store volatile.

Next we slice the function, and consider all loads/stores dead if:

- they are trivially dead: they have no users
- only user is debug info
- only users are dummysideeffect, and debug info: these are values that were previously parameters to functions we've transformed into dummy calls

Loads/stores are not dead if users are dummywrite/dummyread.

Next we inline the function n levels. Then we apply some LLVM transformation to simplify (GVN, SROA, SimplifyCFG, ...).

Next we do aggressive slicing: we remove the less important loads/stores, if needed we remove the important ones too (remember we have to reduce the code for inlining to be feasible). Important loads/stores are those we haven't proved yet, less important ones are those loads/stores we've proved, but influence the loads/stores we haven't proved yet. We replace them with dummy values / dummy calls. Then we do slicing as usual.

Then we look at code size, if possible inline again, and use the simple slicer. Why don't we just keep all stores, or at least all the structs so that SROA has something to work with? The resulting slice would still keep most of the program, and inlining would be infeasible once again.

3.5.5 Reachability

We are now able to identify loads/stores that are potentially buggy, and remove the loads/stores that were already proven as correct/incorrect. That still leaves us with most of the loads/stores in the program to be proved. A user would quickly give up on our tool due to the high false positive rate. What went wrong?

Let's consider the most recent example again: Program 10 on page 32. Our tool is able to create the following constraint for the access at line 20: $((\textbf{zext}\ \text{i32}\ (-1 + \%\text{n})\textbf{to}\ \text{i64})\textbf{umax}\ (\textbf{zext}\ \text{i32}\ \%\text{n}\ \textbf{to}\ \text{i64})\textbf{umax}\ (1 + (\textbf{zext}\ \text{i32}\ (-1 + \%\text{n})\textbf{to}\ \text{i64})))$ Why is the constraint so complicated for such a trivial situation? First of all `n-1` could overflow, and so could `n+1` (remember, we are only looking at the dataflow, and expressions, not predicates). Nevertheless the constraint should always evaluate to %n, the only possible bug this program has is a NULL dereference (the return value of malloc is not checked).

The check at line 10 guarantees that `n-1` doesn't overflow, and malloc(n) guarantees that `n+1` doesn't overflow (malloc would have returned a NULL pointer for such a large number, and we'd have a NULL dereference bug).

It is obvious that the constraint show above can be simplified by using the knowledge about the values %n can take, we can use a solver for this. Usual choices are an SMT solver or an integer linear programming library, such as Omega, or PPL [BHZ08]. See Section 3.5.6 on how the solver is used.

What constraints should we pass to the solver? We could pass all the constraints we can infer, however that would make queries very slow. Rather we only pass the solver constraints we infer about the values involved in the expression we want to prove, thus:

- the formula we want to prove
- ranges of values involved in the formula, as constants. We can get this information from loop trip-counts (if they're constant), and ranges of types. If we would have a Value Range Propagation (LLVM currently doesn't have one) pass we would use the more accurate value range information here.
- reachability predicates: from the terminator instruction in basic blocks that dominate the current basic block (Basic block A dominates basic block B if A is *always* executed when B is.). Current basic block is the BB that contains the load/store instruction we want to analyze.

In the example Program 12 on page 34, `bb1` dominates `bb4`. The predicate inferred from `bb1` is %n == 0, and its value has to be false for bb4 to be reached.

Let's look at Program 3 on page 13 again: here we have no other information about p, but the one from the predicate at line 10. It is however exactly the constraint we want to prove, so the (known to be true) constraint inferred from line 10 is actually the formula we want to prove. We don't even need a complicated solver for this.

In fact most of the constraints and queries are about linear combinations and integer comparisons.

We add constraints to the solver iteratively, it is very likely that we'll find the constraint we need very close to the actual memory access (in a carefully written system / security tool at least. Not that likely to find the constraints in the average userspace program which doesn't even check for the success/failure of malloc).

If we can't prove anything about a memory access , we leave it up to the next phase of inlining to find something.

3.5.6 The Solver

The solver receives a *Scalar Evolution* expression equality, and a set of inequalities and has to return an answer:

- Equality is always true
- Equality is known to have at least one true result
- Equality is known to have at least one false result
- Equality is known to be always false
- Undecidable

As described in Section 6.1 most of the time we have some constraints and inequalities but there is no relation between the size of allocation and the predicates. When there is, the relation is usually very simple.

We could use an SMT solver to obtain a precise answer, or implement a solver using PPL. However for the purpose of this tool a much simpler solver would suffice, and since we don't want to do in-depth analysis (to meet time requirements) it is even convenient to have simple and fast solver.

As seen in Section 3.5 the constraints we need to prove are in the form: $umax(a, b) == a$. Predicates in programs are usually of the form $a - b < C$, where C is a constant.

We can transform $umax(a, b) == a$ to `a` **ugt** `b`, also we can transform `a-b` **slt** `0` to `a` **slt** `b`.

Then using information we have about value ranges we can transform some *slt* to *ult*. To get the information about value ranges we:

- get information from *Scalar Evolution* about loop trip counts
- look at dominator predicates with constants that can restrict the range
- look at the type of the value

Once we have the transformed relations, we simply lookup our constraints in the predicates list, and see if we've got a match.

If we get an exact match, we can give an exact answer. If we get a match, except for the constant, and the constant makes the condition weaker we return that there is at least one false answer, otherwise we return always true. In the former case we can also show the constraint and the constant, this is useful to show off-by-one errors.

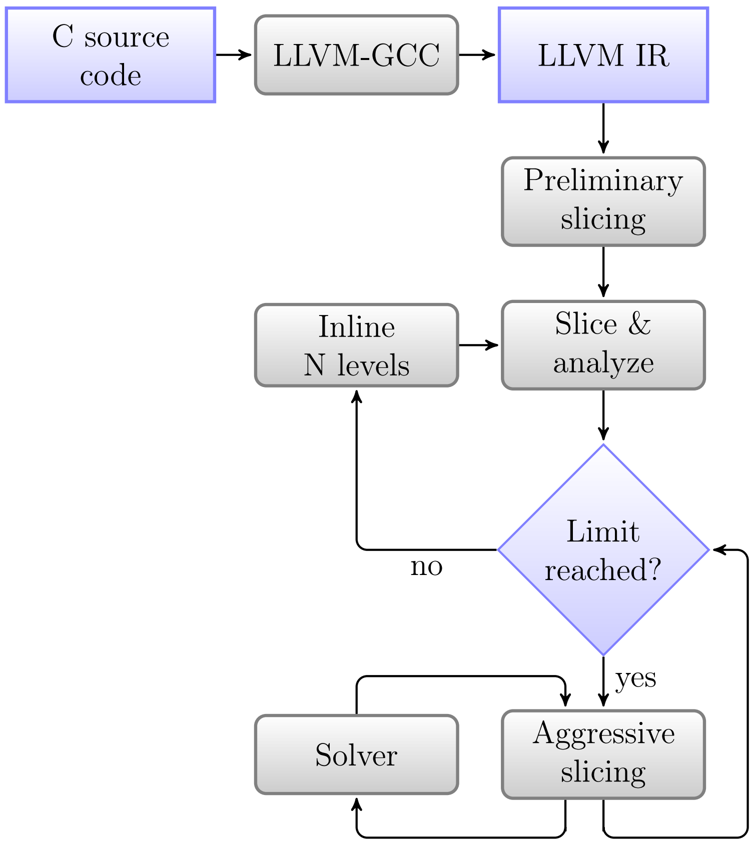
Of course this solver deals with only very few cases, but they are sufficient for the cases we do handle. For the memory accesses we can't handle, the problem is not the accuracy of the solver, but the lack of constraints that would make the system decidable.

Chapter 4

Implementation

4.1 Application architecture

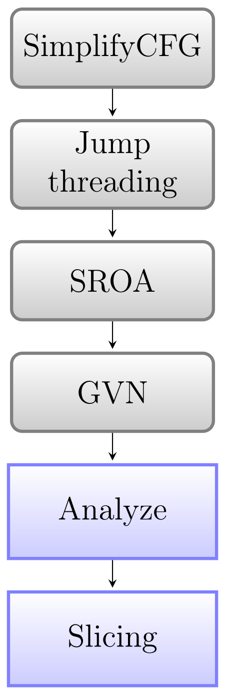
Figure 4.1: Overall architecture



4.2 Classes

4.2.1 ForceInliner

This is the class used for the experiments in Section 3.5.2. It implements two inlining strategies: depth-limit based inlining, and basic-block-limit based inlining. The basic-block limit is achieved by simply counting the number of basic blocks in the caller and



(a) Slice & analyze

Figure 4.2: Architecture

the callee. The depth-limit based inlining is done by recursively traversing the call-graph once on initialization depth-first. Leaves are considered level 0, when returning from recursion the depth is incremented. The depths are stored in a map. When this class is asked to decide whether a certain call should be inlined, it first checks the mode: if we are in basic block mode, it simply compares the basic block counts, otherwise it looks up the depth in the map, and compares to the depth-limit given on the command line. The inheritance diagram is represented in Figure 4.3a on the next page and the collaboration diagram for this class is represented in Figure 4.3b on the facing page.

4.2.2 Logger

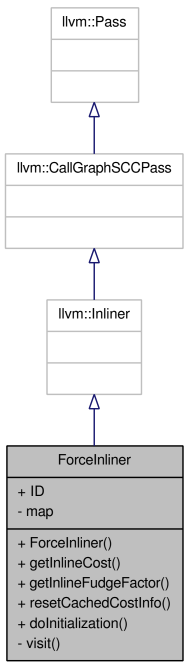
This is the class used to log the results of the analysis as errors/warnings. The messages are logged to standard error, using colors and highlighting to emphasize the error message. Less important messages are logged to standard output.

To make it easier for users to trace the origin of the bug the logger also prints the execution path from the entry point to the current basic block by traversing the immediate dominators of the basic block in reverse order, and printing location information for them. Since we also perform inlining this can actually be an interprocedural execution path.

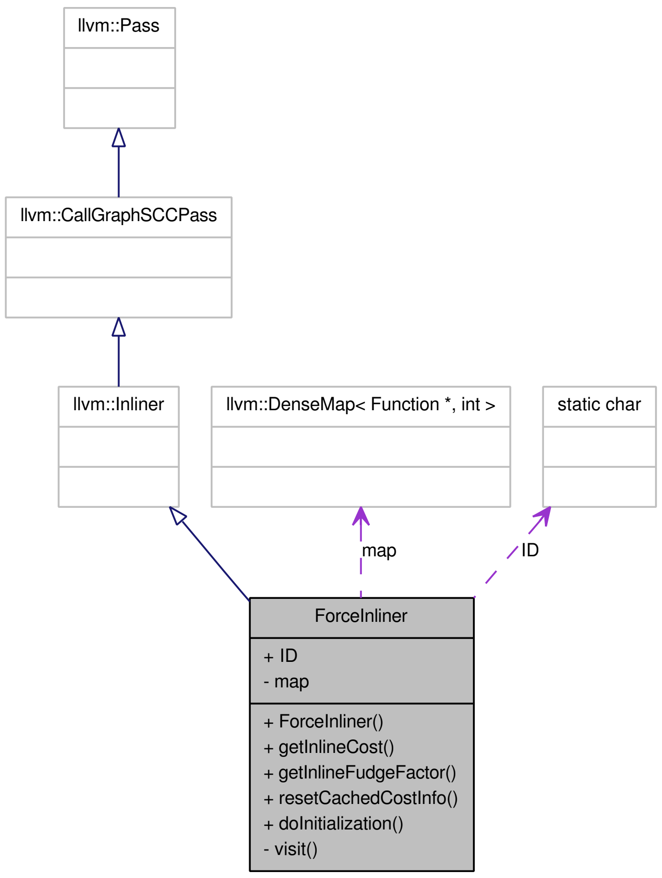
The inheritance diagram is represented in Figure 4.4a on page 44 and the collaboration diagram for this class is represented in Figure 4.4b on page 44.

4.2.3 PointerFinder

This is a helper class used to find the base pointer in a *Scalar Evolution* expression. It is actually a visitor that recursively visits the expression, and searches for pointers. When a pointer is encountered, it is stored in a class field, and the search is resumed. If more than one pointer is encountered the search is aborted. A client class has to call

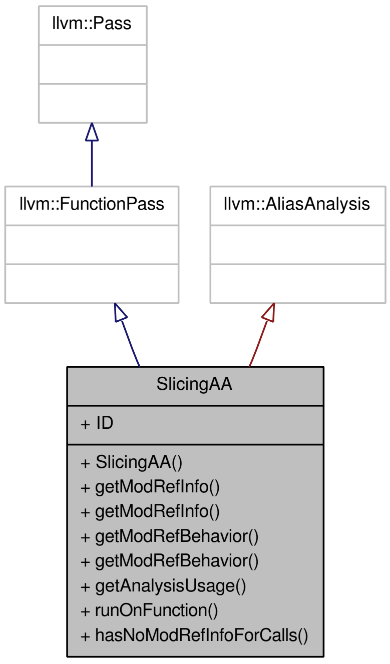


(a) Inheritance diagram

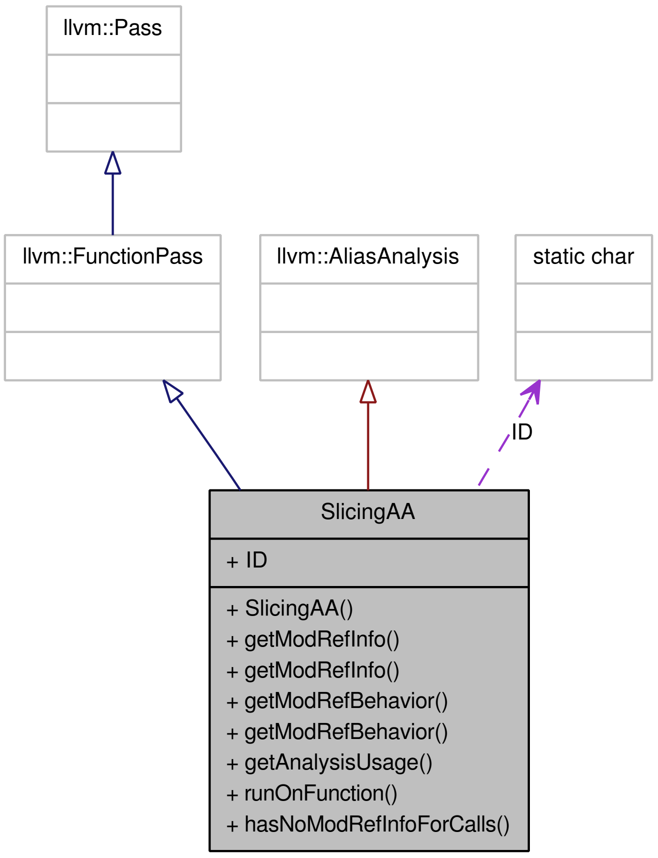


(b) Collaboration diagram

Figure 4.3: ForceInliner class



(a) Inheritance diagram

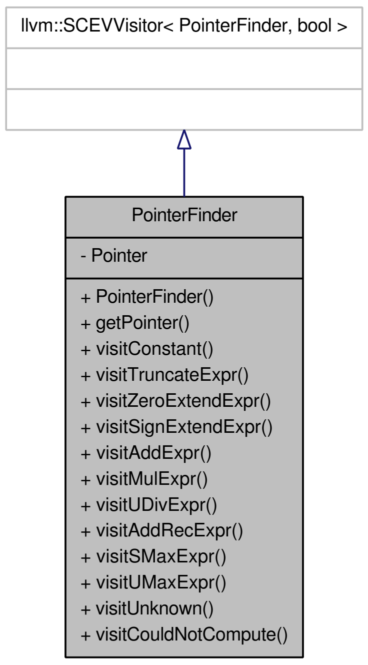


(b) Collaboration diagram

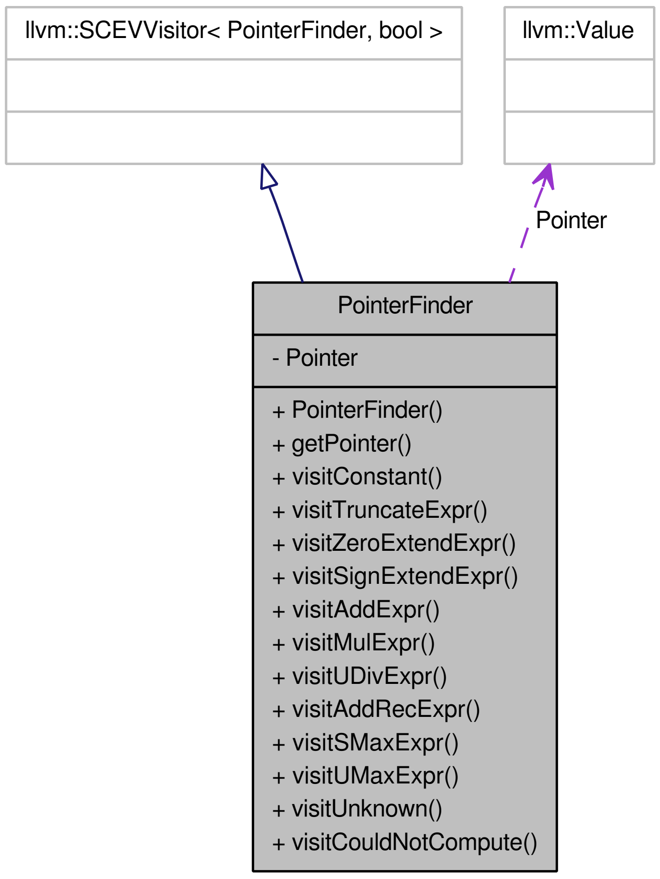
Figure 4.4: Logger class

`visit(SCEVHandle&)`, followed by `getPointer()` if the visit was successful (as shown by the boolean return value).

The inheritance diagram is represented in Figure 4.5a and the collaboration diagram for this class is represented in Figure 4.5b.



(a) Inheritance diagram



(b) Collaboration diagram

Figure 4.5: PointerFinder class

4.2.4 Solver

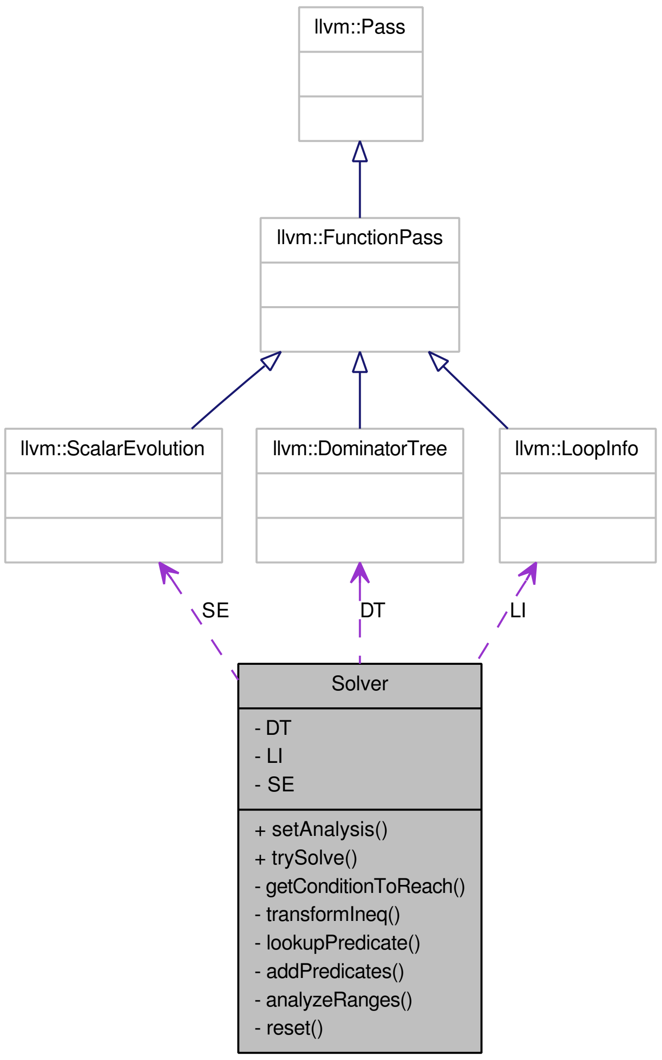
This is the class used to solve inequalities between ScalarEvolution expressions. The algorithm is described in Section 3.5.6.

The collaboration diagram for this class is represented in Figure 4.6 on the next page.

4.2.5 SlicingAA

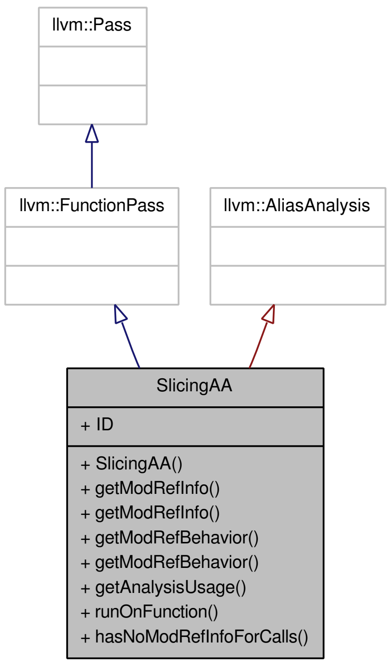
This is the Alias analyses used by the slicer. An alias analysis is needed to provide information about the behavior of the dummy calls (see Section 3.4.2) we introduced. It tells LLVM that calls to `dummyread` only read its argument, and doesn't access any globals or other variables. `dummywrite` calls only write to its argument, and doesn't access any other globals or local variables. `dummysideeffect` calls can read/write globals, but cannot read/write to local variables.

Figure 4.6: Collaboration diagram for Solver

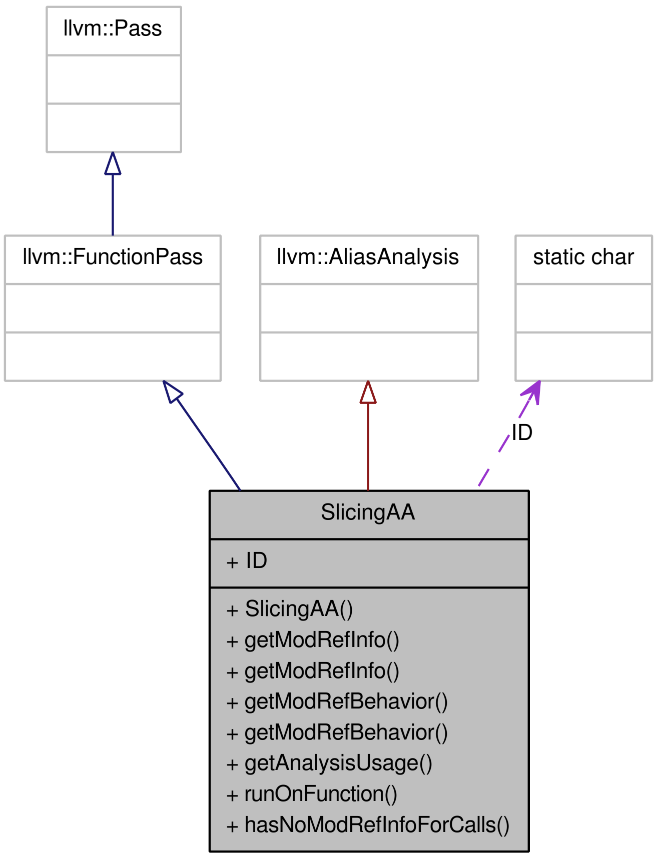


Alias Analysis are chained in LLVM, thus we register this Pass as part of the Alias-Analysis group.

The inheritance diagram is represented in Figure 4.7a and the collaboration diagram for this class is represented in Figure 4.7b.



(a) Inheritance diagram



(b) Collaboration diagram

Figure 4.7: SlicingAA class

4.2.6 SlicingInlinePrepare

This is the class used to do the initial slicing, it transforms one function a time. Thus this is registered as a FunctionPass in LLVM.

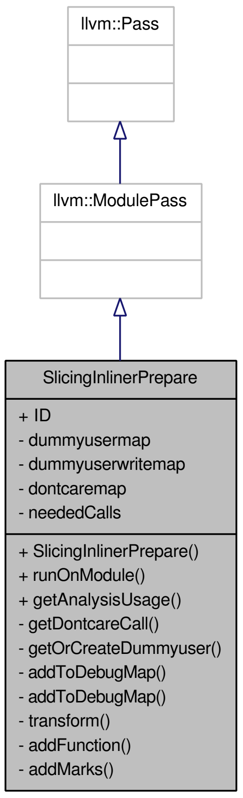
The algorithm of this class was described in Section 3.5.1, and the first 4 steps in Section 3.3.

The inheritance diagram is represented in Figure 4.8a on the following page and the collaboration diagram for this class is represented in Figure 4.8b on the next page.

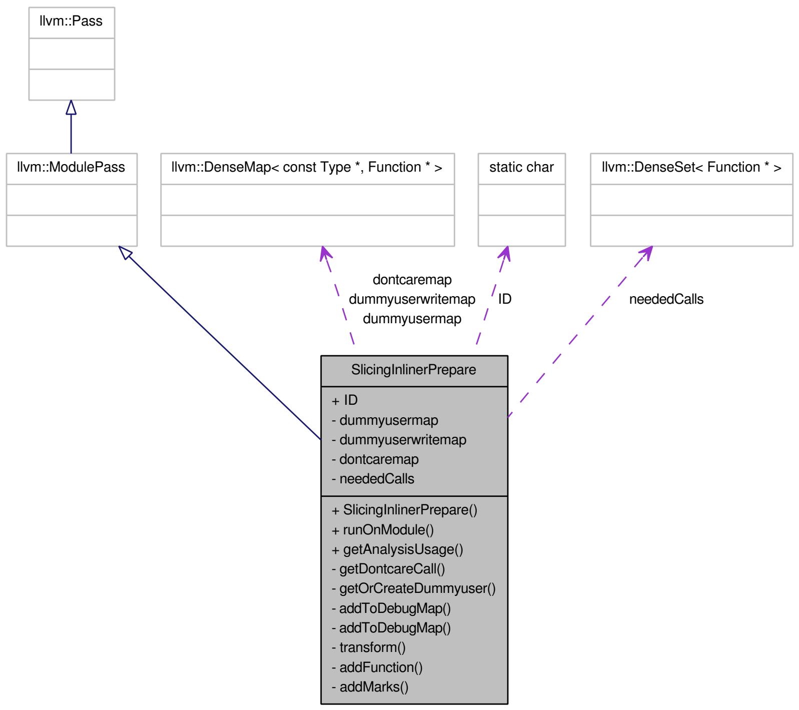
4.2.7 SlicingSimplifier<W>

The algorithm for this class was described in Section 3.5.3, and the last steps in Section 3.3. It also contains the bounds checker analysis algorithm.

It uses the Logger and Solver classes to perform its task, and also queries the `TargetData`, `LoopInfo`, `ScalarEvolution`, `DominatorTree` LLVM passes.



(a) Inheritance diagram



(b) Collaboration diagram

Figure 4.8: SlicingInlinePrepare class

The `<W>` template argument is a boolean which controls whether warnings for unprovable memory accesses are shown, it is instantiated as true only when it is the last simplifier pass to be run.

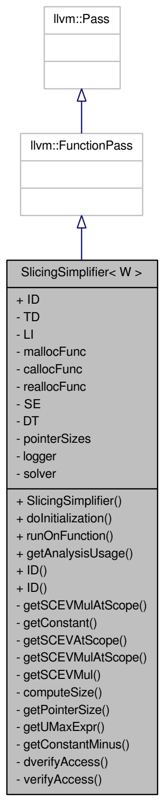
The inheritance diagram is represented in Figure 4.9a on the facing page and the collaboration diagram for this class is represented in Figure 4.9b on the next page.

4.2.8 Stats

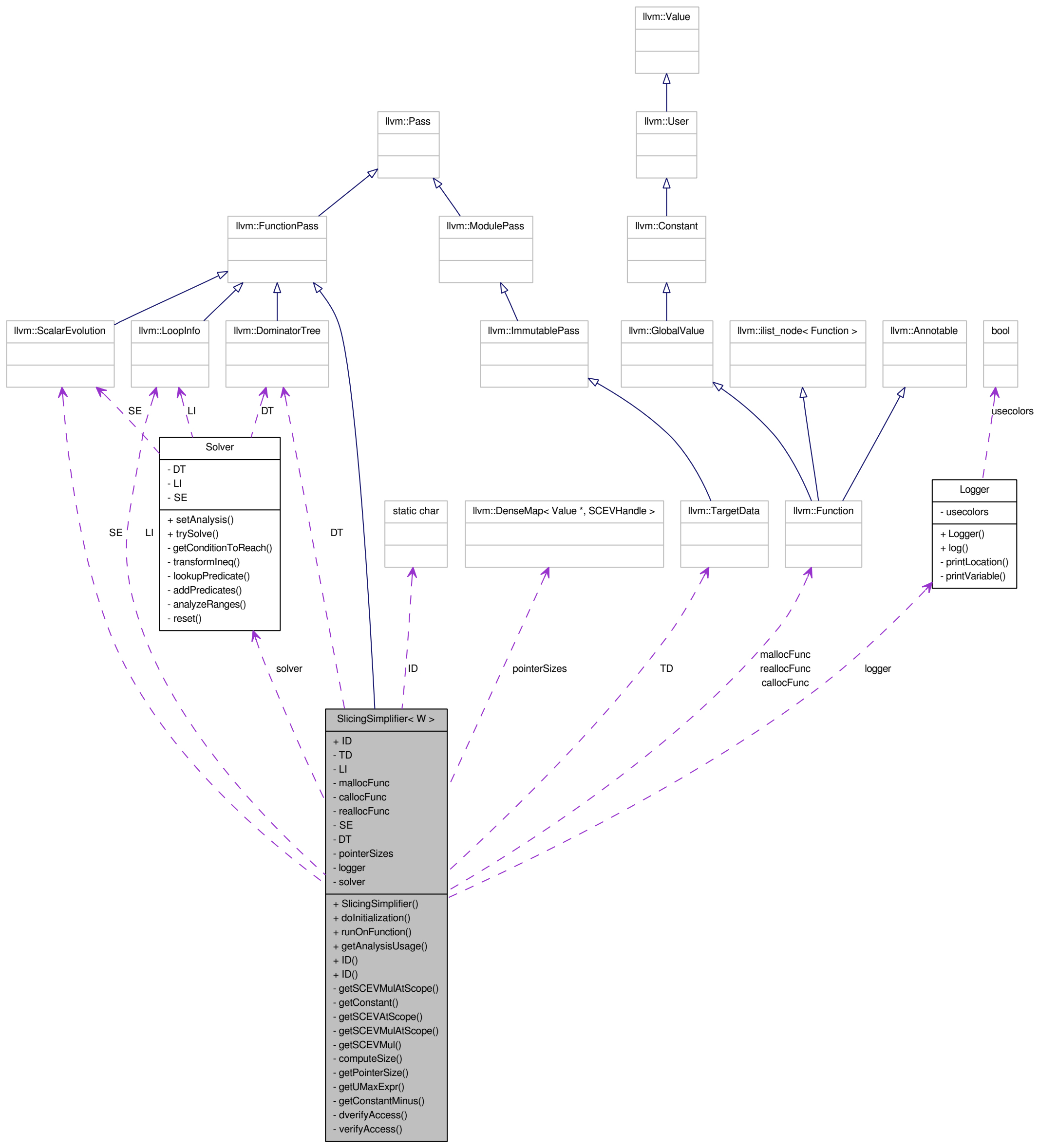
This is another class for the experiments in Section 3.5.2. It counts the number of basic blocks and instructions in each function, then it prints the maximum number of BBs/instructions in a function, and the average number of BBs/instructions per function.

This is a ModulePass because it needs to retain state between functions (the sum of BBs/instrs used to calculate the average).

The inheritance diagram is represented in Figure 4.10a on page 50 and the collaboration diagram for this class is represented in Figure 4.10b on page 50.

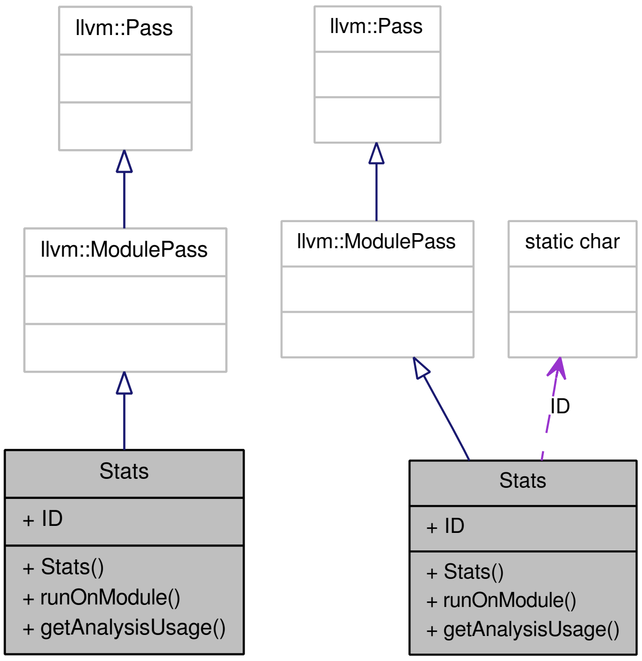


(a) Inheritance diagram



(b) Collaboration diagram

Figure 4.9: SlicingSimplifier<W>



(a) Inheritance diagram **(b)** Collaboration diagram

Figure 4.10: Stats class

Chapter 5

Using the application

Our tool takes as input LLVM bitcode files. For best results the entire program should be linked into a single bitcode file, so that we can do whole-program analysis.

To do more accurate analysis, libc could be compiled to bitcode, and linked into the application. uClibc should be suitable for this purpose, however building it is beyond the scope of this thesis.

5.1 Building the tool

Assuming you have unpacked the sources to `$SRCROOT`, you should have a directory structure as shown in Figure 5.1.

You should create an object directory where you will build everything, we'll refer to this as `$OBJROOT` from now on.

You should create the directory structure for `$OBJROOT` as shown in Figure 5.2 on the next page.

You configure `LLVM` as follows, `$PREFIX` is the prefix where you will install LLVM:

```
$ mkdir $PREFIX
$ cd $OBJROOT/llvm-obj
$ $SRCROOT/3rdparty/llvm/configure --enable-optimized --prefix=$PREFIX
```

Figure 5.1: Source directory layout

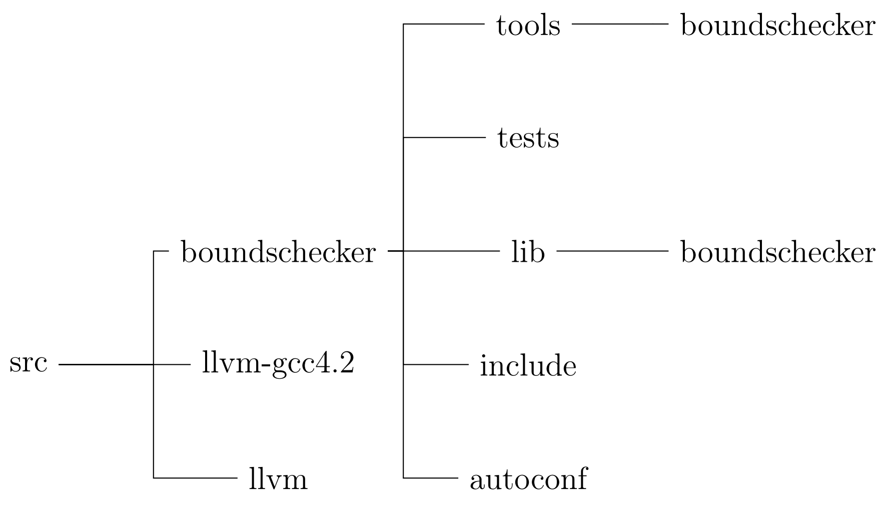


Figure 5.2: Object directory layout

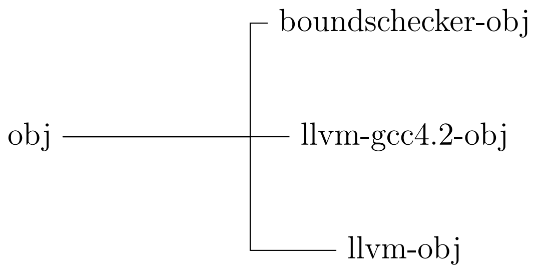
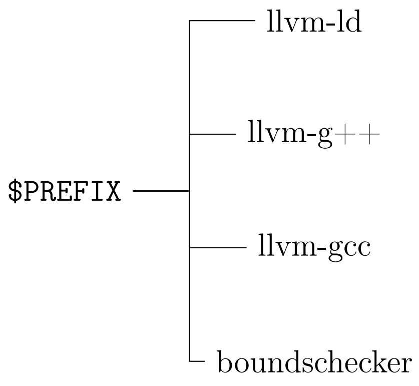


Figure 5.3: Tools installed



```
$ make -j4 && make check && make install
$ cd boundschecker
$ $SRCROOT/boundschecker/configure --prefix=$PREFIX --enable
$ make && make install
$ cd $OBJROOT/llvm-gcc4.2-obj
$ $SRCROOT/llvm-gcc4.2/configure --prefix=$PREFIX
--program-prefix=llvm- --enable-llvm=$OBJROOT/llvm
--enable-languages=c,c++
$ make -j4 && make install
```

The above build instructions are for `Linux`, for other systems see `README.LLVM` in `3rdparty/llvm-gcc4.2`. Alternatively you can download a pre-built llvm-gcc package from `llvm.org`, or from your OS distribution.

If the build was successful you should have the following tools in `$PREFIX` that we'll use, see Figure 5.3.

5.2 Compiling an application to LLVM bitcode

5.2.1 Single-source applications

To compile a single-source application to bitcode you can use the following command:

```
 $ llvm-gcc -O1 -g -c foo.c -o foo.bc -emit-llvm
```

You can use any additional flags that you would normally use with gcc (such as `-I` and `-D`). `foo.bc` is the file you need to provide as input to our tool.

5.2.2 Multi-source applications

An application that has multiple source files, without special build requirements can be built as follows:

```
 $ llvm-gcc -O1 -g -c foo1.c -o foo1.bc -emit-llvm
 $ llvm-gcc -O1 -g -c foo2.c -o foo2.bc -emit-llvm
 ...
 $ llvm-gcc -O1 -g -c fooN.c -o fooN.bc -emit-llvm
 $ llvm-ld -disable-opt foo1.bc foo2.bc ... fooN.bc
-link-as-library -o foo_all.bc
```

foo_all.bc is the file you need to provide as input to our tool.

5.2.3 Compiling an application that uses autoconf

If the application uses autoconf, and automake as the build environment, it can be compiled as follows:

```
 $ export CC=llvm-gcc
 $ export CXX=llvm-g++
 $ ./configure --disable-shared --enable-static
 $ make CPPFLAGS="-O1 -g -emit-llvm" CCLD="llvm-ld -disable-opt"
CFLAGS= CXXFLAGS= -j4
```

We need to set CPPFLAGS, instead of CFLAGS and CXXFLAGS, because llvm-ld doesn't understand -O1 and -g.

We could have built the application with dynamic libraries, but our tool cannot do interprocedural analysis across dynamic library calls, thus we needed to make sure that even if the application is composed of many libraries, the bitcode file we build is self-contained.

You will find an <applicationname>.bc file for each program built. These are the input to our tool.

5.3 Running the analysis

Following one of the steps in Section 5.2 you will get a bitcode file that can be used as input to our tool, replace foo.bc with the filename of the bitcode you got in the previous steps:

```
 $ boundschecker foo.bc >details.log
```

This command will print the bugs it finds to standard error, and more details about the bugs / low-priority bugs are printed only to standard out (redirected to details.log above).

Chapter 6

Conclusion

6.1 Experimental results

Our tool locates invalid memory accesses in simple cases (constant sizes, allocation/dereference in same function), and tries to find invalid accesses interprocedurally too. Unfortunately it is not possible to do both a fast and in-depth analysis for interprocedural cases, so we have traded accuracy for speed. If desired, inlining depth could be increased to gain more accuracy. Verification speed is comparable to compilation time, thus this tool could be easily integrated into the compilation process for medium sized programs. For example checking the *Clam Antivirus* program, which has 100 000 lines of code is done in less than a minute.

Even though we do interprocedural analysis, we handle only simple cases, and the cases encountered in real programs cannot be all handled by our approach: for example approximatively 1/3 of the memory accesses in ClamAV can be proved. Increasing inline depth won't solve this either. The fundamental problem is that our analysis can't determine how some of the values used in predicates relate to the size of the allocated memory.

For example if a value is read from a file at a certain position, and then later another value is read from the same position our analysis doesn't know that they are the same. Another problem is determining whether values stored to a place in memory and later read are the same or not. For simple cases this usually works (struct fields), however if there are stores that may alias the pointer between the original store and the later load we can't know for sure whether the value has been altered or not, and conservatively assume that the two values may not be equal.

6.2 Conclusions and future work

We have described a tool that can be used to detect bounds overflows, that handles simple intraprocedural and interprocedural cases. We achieved our speed target, being able to analyze medium sized program in time comparable to compilation time. By doing so we sacrificed the accuracy of our analysis, we are unable to prove/disprove the validity of more complex memory accesses. Still this tool is useful in catching common bugs during the development of an application (off-by-one, missing bounds checks).

To get better results we would need a better alias analysis, or to construct a model for each pointer and use a solver to determine whether the values are the same or not. However all this would cause the analysis time to increase, and we'd no longer be able to quickly verify a program. There is a way to increase the accuracy of the analysis without affecting performance too much, besides a better slicing algorithm that simplifies the program even more, and better propagation of bounds information: we could find out which variables store the size of a pointer, and then see whether there is a predicate checking the size against the offset for each access. If the pointer would be reallocated without changing the size, or if the size is not available at all when dereferencing a pointer we could log a warning too. This is based on the observation that bugs often appear where the code doesn't respect the behavior we observed in other parts of the program, as described in [ECH+01]. Our tool doesn't currently look for these kinds of bugs, but it could be a potential improvement.

Appendix A

Tools and libraries used

A.1 Versions

The following versions of tools/libraries were used:

- LLVM from SVN r72858:

  ```
  llvm version 2.6svn
  Optimized build with assertions.
  Built May 30 2009(11:03:05).
  ```

- LLVM-GCC from SVN r72774:

  ```
  Using built-in specs.
  Target: x86_64-unknown-linux-gnu
  Thread model: posix
  gcc version 4.2.1 (Based on Apple Inc. build 5646) (LLVM build)
  ```

- System compiler used was GCC 4.3.3:

  ```
  Using built-in specs.
  Target: x86_64-linux-gnu
  Thread model: posix
  gcc version 4.3.3 (Debian 4.3.3-10)
  ```

All tests were performed on a system with the following configuration:

Operating System Debian GNU/Linux (sid) `Linux 2.6.29.3 x86_64`

CPU Intel® Core™2 Quad CPU Q9550 @ 2.83GHz

Memory 4 GB DDR3

Storage 6 x WD Caviar Black 750 GB SATA II in software RAID10

This document was typeset using pdfLaTeX:
`pdfTeXk, Version 3.141592-1.40.3 (Web2C 7.5.6).`

A.2 Copyright and license

We use the LLVM compiler framework, which is:

We also use the `LLVM-GCC` compiler, which is:

References

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press. 6, 15

[BCC$^+$02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, October 2002. 2

[BH07] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *in Computer Aided Verification: 19th International Conference, CAV 2007, ser. LNCS*, pages 371–383. Springer, 2007. 2, 14

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. 38

[BWZ94] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *In International Symposium on Symbolic and Algebraic Computing*, pages 242–249. ACM Press, 1994. 5, 14

[CC08] Dawson Engler Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 12 2008. 2, 14

[cla] LLVM/Clang Static Analyzer. 2, 13

[DKA06] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM. 2

[ECH$^+$01] Dawson Engler, David Y. Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, December 2001. 56

[ISO07] ISO. International standard iso/iec 9899 - programming languages c. Technical Report WG14 N1256, ISO/IEC, 2007. 11

[JBB02] Name Johnnie, L. Birch, and Johnnie L. Birch. Using the chains of recurrences algebra for data dependence testing and induction variable substitution, 2002. 5, 14

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 13

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`. 13

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, July 1979. 7, 15

[NMRW02] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction*, volume 2304, pages 213–228, 2002. 13

[Pop04] Sebastian Pop. Scalar evolutions. slides, 2 2004. 5, 14

[Sea08] Robert C. Seacord. *The CERT C Secure Coding Standard.* Addison-Wesley Professional, 2008. 11

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. 9, 15, 27

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *SIGSOFT Softw. Eng. Notes*, pages 327–336. ACM Press, 2003. 2