HYBRID STATIC/DYNAMIC TYPE SAFETY FOR C/C++ PROGRAMS

BY

ARUSHI AGGARWAL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Vikram S. Adve

ABSTRACT

C/C++ are the languages of choice for development of many widely used system softwares. However, these languages do not provide the strong safety guarantees that safe languages such as Java do. Programming mistakes can introduce type errors that are not caught at compile time. These errors may subsequently be triggered at runtime, and their sources maybe hard to detect.

This thesis presents the design and implementation for a dynamic type checker for C/C++ programs. It is built using the LLVM compiler infrastructure and provides type safety for programs that have been compiled to the LLVM IR.

The thesis also discusses the design of static analysis to reduce the overhead of the dynamic type checker. We present an implementation of the static analysis and discuss how its results can be used to optimize the dynamic type checker.

We also present performance evaluation on various benchmarks and system software. We present results that show that we can catch most errors with relatively low overhead.

To all my teachers, specially my parents.

ACKNOWLEDGMENTS

I would like to thank my adviser, Vikram Adve, for his support and patience through the duration of this work. His drive and enthusiasm was a source of inspiration and motivation. Involved discussion with him have been the backbone of this thesis.

Brice Lin worked on the initial implementation of this tool. Andrew Lenharth was instrumental in building most of the type inference infrastructure. Working on this project was made fun and interesting by the brilliant people I worked with everyday, John Criswell and Will Dietz. Endless whiteboard deliberations with them have helped make this research better. From them I have learnt lessons in the art of writing good and robust tools.

Finally, the last two years have been great with the support of Nisha Somnath, and Ankit Singla who have great friends during this time in Urbana-Champaign.

This research was funded by AFRL contract number FA8650-10-C-7022 and by DoD MURI AF Subcontract UCB 00006769.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

C and C++ are popular in the field of system software development, as they provide support for low level control over the system thereby helping write more efficient tools [1]. They allow arbitrary pointer arithmetic, casts between pointers and integers, other unchecked type casts and explicit memory management. However, these advantages come at the cost of loss of the security guarantees, such as those provided by safe languages like Java. Programming bugs can introduce array out-of-bounds errors, null pointer dereferences, dangling pointer errors and several kinds of type errors. The static type system for C/C++ is weak and there are no runtime mechanisms to protect against such errors.

Many tools exist to detect the first two types of errors, i.e. array out-of-bound errors and null pointer dereferences, which are together broadly classified as *spatial memory safety* errors. These include, among others, Valgrind [1], Purify [2], Safe-C [3], and SAFECode [4]. These tools are also helpful in protecting against *temporal memory safety* errors, such as dangling pointers, double `free`'s and invalid `free`. In addition, garbage collection [5] can be used to prevent such errors. However, these tools are not useful in detecting type errors, which are caused due to the weak typing rules of C.

A *type error* occurs when the operands to an operation are not of compatible types. Passing the wrong number or type of arguments to functions, adding two incompatible types or performing operations on wrong operand types(shifts on `float`s or `bool`s, addition of `bool`s) are all examples of type errors. Type errors are not as clearly defined as the memory safety errors, making them hard to identify. Distinguishing incompatible casts from programmer intent is hard and enforcing a strict type system can lead to a large number of errors. The key, thus, is to find a balance between enforcing type safety and allowing programmers to take advantage of the language's

[1] `http://en.wikipedia.org/wiki/Fast_inverse_square_root`

weak typing features. We introduce the concept of *Primitive Typing*, which we believe is strong enough to be used for detecting type errors in realistic C programs while keeping the false positive rate low. We track and type check primitive type values but only when these are used in a type sensitive operation.

We describe the design and implementation of a dynamic type checking tool for C/C++ programs. It is designed to be a useful debugging tool, allowing programmers to catch problems before it is deployed. We instrument the program to track types for each memory location, updating on writes, and checking on uses. When a value is used in a type unsafe manner, the tool flags an *error*.

The tool has been built using the LLVM compiler infrastructure [6], and instruments the LLVM Intermediate Representation(IR). The LLVM IR uses SSA [7] form and is typed. We are thus able to statically type check all SSA values requiring runtime type checks only on the values that are stored in memory. We use a shadow memory to track the type information, without making any change to the pointer representation or data layout of the program.

Other tools that provide similiar functionality include Loginov et al. [8], Hobbes type checker [9] and Shen et al. [10]. Longinov et al. use source to source translation for instrumentation, and can handle only those programs that are written entirely in C. They also do not check for types on calls to variadic functions. The Hobbes type checker runs on binary programs and has the advantage of tracking through functions in compiled libraries. However, as it inserts checks in binary code, the overhead is rather high($\sim$141x). Our tool provides a good balance between the two tools, as we instrument the LLVM IR. We can thus type check functions in any libraries that can be linked in as LLVM bitcode. We can also take advantage of the static analyses and optimizations, that are available in a compiler infrastructure, both to reduce the number of checks inserted, and to make them more efficient. Shen et al. [10] provide a similiar tool. However they track types on all variables in a program. In comparison, we track a reduced set of variables consisting only of the values stored in memory. Furthermore, they also dynamically keep track of aliasing information, increasing the overhead on each update of type information; our runtime does not have this overhead.

We have also built a static type inference algorithm that uses our type

system. We have used its results to improve the performance of our dynamic tool. The static type inference helps us recognize objects which are always accessed in a type consistent manner and hence are *type safe*. The checks on these objects can be removed, reducing the runtime overhead significantly. Our type inference algorithm is built as a part of the Data Structure Analysis(DSA) [11] algorithm.

We have tested our tool on a wide set of benchmark programs and a few moderately sized system softwares. Our results indicate that our tool is effective in finding several types of errors, while keeping false positives to a minimum. We detected a format string error in Sqlite, which after we reported it was fixed.

The main contributions of this thesis are

1. Primitive Typing: An extension of the C type system that enforces type safety for uses of primitive types in C/C++ programs.

2. Dynamic Type Tracking Tool: A tool to enforce Primitive Typing dynamically. It is effective for real world programs, catching real problems while keeping spurious warnings to a minimum.

3. Static Type Inference: A static type inference algorithm based on primitive typing. We also discuss using static type inference results to reduce the dynamic overheads.

CHAPTER 2

RELATED WORK

Identifying errors in C/C++ programs has been the focus of many projects. Most of the effort has been concentrated on achieving memory safety for these programs.

Purify [2] is a widely used memory access checker. Valgrind [1] is a similiar tool for Linux binaries. They track status bits for each memory location and on every memory operation check the status information and ensure that it is appropriate for the operation. These are useful for detecting most memory safety errors, though at high overheads. However, they are not useful for detecting mosty type errors.

Loginov et al. [8] describe a debugging tool for type checking C programs dynamically. It uses source to source transformation to instrument the code with type checks. They track primitive types only similiar to our approach. However, they are not as complete and do not track types through variadic functions or indirect function calls. They, like us, enforce types only on some uses of values and not all like our tool. They do not differentiate between pointer types, similiar to our approach. They use a shadow memory approach to track type information. However, they have high overheads (6-130x) as they introduce complicated source code expressions to do the type tracking.

The Hobbes type checker [9] is a tool that type checks binary programs. They instrument binary programs and are not dependent on source code for the program or the libraries used. They use shadow memory to track the type information. As it works on binary programs it tracks only primitive types. Instrumenting binary code by using an interpreter is very high cost and their total overhead is around 140x. Their instrumentation also has to account for a lot of low level compiler optimizations which, though correct, can cause spurious type warnings.

Shen et al. [10] provide a type checker that is based on Physical subtyping [12]. They record a dynamic type for each program variable, including

pointers during execution. They also track alias sets at runtime and propogate type information to the aliasing pointers, increasing the overhead on each update of type information. They track structure and array types for objects and follow prefix matching to determine subtyping relations. They report a median overhead of 140x for their benchmarks.

Unlike the above approaches, we use a hybrid approach by using a static type inference to supplement the dynamic tracking tool. The static analysis helps reduce our overheads as compared to the techniques above. We are also able to take advantage of the compiler optimizations available in a compiler infrastructure like LLVM to reduce the number of checks inserted and making them more efficient. We track types on a reduced set of variables that are stored in memory. This is possible due to the typed nature of the LLVM IR.

CCured [13] uses static type inference to type check a program and reduce the number of type checks required. Their static inference is based on structural typing, making it more restrictive than our approach. Also, we restrict checks to operations that are type sensitive. As the representation of `WILD` pointers is different from the representation of pointers in the program, CCured requires that the programmer makes some changes to the source code to run correctly, while our tool is fully automatic. Their static type inference algorithm handles arrays much better than we do, helping to lower their overhead.

Using static type checking to identify type errors in C programs is also an option [12]. However, static analysis is flow insensitve and likely to cause warnings when applied to real world C programs. The advantage of using a hybrid approach is being able to make use of the static analysis results, but only flagging errors if they happen during a particular execution of the program.

CHAPTER 3

BACKGROUND

3.1 Motivating Examples

In this section we discuss some of the type errors that our tool will catch, but are not caught by C compilers. These are the motivation for a dynamic type checker like ours. We also present details of how we deal with each of these error.

3.1.1 Unions

In this example, borrowed from Loginov et al. [8], an error occurs because we write into one field of a union, and then read it as another.

```
union U {
  int u1;
  int *u2;
} u;

u.u1 = 10;
int *p = u.u2;
*p = 0;
```

The read of `u.u2` generates a type error, allowing us to identify the source of a potentially harmful pointer dereference later on. Since we track types of all memory locations, we would catch the mismatch when the same memory was read as an `int*` after having been written as an `int` earlier.

3.1.2 Simulating Inheritance

C programmers often use structures to simulate hierarchy of classes in an object oriented fashion. In this example, also borrowed from Loginov et al. [8], the following declarations are used to simulate a superclass `Sup` and subclass `Sub`.

```
struct Sup {int a1; int a2};
struct Sub {int b1; int b2; char b3};
```

Functions which operate on objects of type `Sup` and access only the first 2 fields and can thus take an argument of either type, e.g.

```
void f(struct Sup *s) {
  s->a1 = ...
  s->a2 = ...
}
```

If we were to check the type of the object stored in `s` and enforce it to be of type `struct Sup`, we would cause a lot of false positives. By checking types only on those fields of a structure that are actually accessed, we allow for such uses of structs, with fewer false positives.

Since the subtyping is done manually in C, it is quite possible for the programmer to do so incorrectly. If a programmer were to wrongly subclass `struct Sup` by not maintaining the same types as the super class at the offsets accessed, our tool would detect an error.

3.1.3 Uninitialized Memory

Our tool also detects uses of uninitialized memory locations, as shown in the following example.

```
void foo() {
  int a;
  ...
  if(flag)
     a = 5;
  i = a;
   ...
}
```

The initialization of `a` is dependent on `flag`. In certain executions it may be `false`, and `a` and `i` may not get initialized. Reading `a` is flagged as a type error by our tool since we use a special type to track uninitialized memory. Any subsequent uses of either `a` or `i`, will also get flagged, till these are initialized.

This is a simple example in which the compiler will generate a warning, flagging the use of an uninitialized variable. However, more complicated examples can be easily constructed where this is not the case.

3.2 Physical Subtyping

The concept of physical type checking for C was introduced by Chandra and Reps [12]. They argued that C with casts and pointers, needs stronger type checking than without it. They based their type inference on the physical layout of structs in memory, to help reduce the number of false positives. They defined that
"... to be physically type safe, each pointer dereference should point to "valid memory", and refer to a "valid type". By valid memory, we mean that the address computed for the load of the specified field from memory must be within the bounds of the allocation unit that the pointer currently points to. By valid type, we mean that the ground type being referred to must be the same as the one stored at the memory location."

They use the above concept to infer types statically for C programs by taking into consideration the layout of C `struct` fields. They allow structure types to be subclassed as long as they have a common prefix when their layout is considered. We use this idea as the basis of *Primitive Typing*, introduced in the next chapter.

3.3 LLVM IR

The instrumentation for the type tracking is done at the LLVM IR level. The IR is a typed SSA representation of the source level program. All scalar values are in SSA form, whereas aggregate and address-taken variables are left in non-SSA form. The LLVM IR is strictly typed, and each SSA value

has an associated type and all obey strict type rules. All instructions are typed too, and have restrictions on their operands. For example, an `add` instruction requires that both the operands be of the same type, which must be an arithmetic type, and it produces a value of that type [14].

`bitcast` instructions are the only way to get type unsafety in the LLVM IR. They can be used to cast values in virtual registers from one type to the other. Bitcasted pointers can be used to cause type unsafe behavior through memory, when a value is written using one pointer, and read using the casted one.

The LLVM types are source language independent data representations that are mapped from higher level language types [14]. The primitive types in the system are `void`, `bool`, integers of all sizes, and floating point values of varying precisions (`float, double, x86_fp80, fp128, ppc_fp128`). Arrays, vectors[1], structures, pointers and function types are constructed using other types. High-level source types are lowered to LLVM types by the front end, e.g classes in C++ are represented as LLVM structure types which show the layout of the fields and a function table for the functions.

[1]Our design does not presently handle vector types or operations on them.

CHAPTER 4

DESIGN

In this chapter, we introduce our type system for the LLVM IR that is enforced by the tool. The static typing rules are discussed first. The operational semantics detail the dynamic type checks required for a safe execution. The design of the runtime checks is discussed later in the section.

4.1 Primitive Typing

We introduce a new type system for C called *Primitive Typing*. It is an extension of the physical subtyping algorithm as it is based on the physical layout of types in memory. However, instead of tracking structure types we restrict tracking to primitive types. We only enforce checks on operations which are type sensitive. We do not restrict pointer arithmetic or casts as these are not affected by the operand types. Structure and array types (for an object) are not tracked. The program passes all type checks as long as the actual values used in operations are of the correct type.

Instead of enforcing subtyping via prefix matching of structures, we enforce types only on a specific field(at a specific offset) of the object. This provides more flexibility to the programmer while also ensuring that no type error is goes undetected. We do *not* enforce multiple pointer types, treating all pointers types as the same type. We believe that it is only the final use of a pointer to access a primitive type value that needs a type match with the used type.

The type system is both strong enough to catch real errors when they occur and permissive enough to allow most real world C programs. This is the key to reducing the number of false positivies. If any value is used in an operation with the wrong type, an error will be flagged. We do not flag errors on out-of-bounds array access or on casting an integer to a pointer,

$$\frac{\cdot}{T \leq \top}$$

$$\frac{\cdot}{\bot \leq T}$$

$$\frac{\cdot}{\bot \leq \top}$$

Figure 4.1: Subtyping relations

as long as the value being accessed has the correct type and alignment. A separate memory-safety checking tool like SAFECode [4], SoftBound [15], Baggy Bound Checking [16], etc., can be used to detect these errors.

4.2 Syntax

4.2.1 Types

The types in our system are the primitive LLVM types. These include integers of all sizes(`i`N, $1 \leq N$), float(`float, double, x86_fp80, fp128, ppc_fp128`) and pointer type(`ptr`). All LLVM pointer types are assumed to be of type `ptr`. These types are supplemented by the bottom($\bot$) and top($\top$) types which help form a lattice of types. $\bot$ indicates *uninitialized* memory which cannot be used in any type sensitive operation. $\top$ indicates *initialized* memory without a specific type. It is compatible with every type in the system and takes the type of its first use. $\top$ is used primarily for library functions, when the exact type of the value is not known, to specify that the memory has been written to. The subtyping relation is described in Figure 4.1.

4.2.2 Language Constructs

The type system is based on the LLVM IR [17]. The static type rules and the operational semantics operate on LLVM instructions. We have simplified some instructions to make the rules easier to read, without loss of generality. Some of the important details of the IR used in the rules are

1. All `gep` instructions take types as an argument. The size of the type before an index indicates the multiplier for that index.

2. We assume all `gep` indexes are `i64` for simplicity of the design. Only structure indexes are `i32`.

3. We store all aggregate values in memory. Thus, there are no `insertvalue` and `extractvalue` instructions in the semantics.

4. We also assume that all `phi` nodes have been eliminated by inserting copies into the predecessor blocks.

4.3 Static Type Rules

The typing rules are used for type checking programs statically. All programs that type check correctly statically, when instrumented with runtime checks, will be type safe dynamically. The static type checker ensures the program is well constructed and according to the LLVM IR. The dynamic type checker ensures that the type system we describe is being enforced and no type sensitive operation operates on operands of the wrong type.

The typing judgements used by the system are described below. These are similiar to the typing rules for LLVM instructions. The static type system is expressed by typing judgements as $\Gamma \vdash v : T$ where v is an LLVM value and T is a type in our lattice. Γ is the typing environment that maps each variable to its type. We also use int, $float$ and ptr to indicate the class of types. int_i denotes an integer of size i bytes. Similiarly, $float_i$ denotes a floating point value of i bytes.

Binary Operations

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash add_T\ v_1,\ v_2 : T}\ [\text{add}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash fadd_T\ v_1,\ v_2 : T}\ [\text{fadd}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash sub_T\ v_1,\ v_2 : T}\ [\text{sub}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash fsub_T\ v_1,\ v_2 : T}\ [\text{fsub}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash mul_T\ v_1,\ v_2 : T}\ [\text{mul}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash fmul\ T\ v_1,\ v_2 : T}\ [\text{fmul}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash udiv_T\ v_1,\ v_2 : T}\ [\text{udiv}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash sdiv_T\ v_1,\ v_2 : T}\ [\text{sdiv}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash fdiv_T\ v_1,\ v_2 : T}\ [\text{fdiv}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash urem_T\ v_1,\ v_2 : T}\ [\text{urem}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash srem_T\ v_1,\ v_2 : T}\ [\text{srem}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash frem_T\ v_1,\ v_2 : T}\ [\text{frem}]$$

Binary Bitwise Operations

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash shl_T\ v_1,\ v_2 : T}\ [\text{shl}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash lshr_T\ v_1,\ v_2 : T}\ [\text{lshr}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash ashr_T\ v_1,\ v_2 : T}\ [\text{ashr}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash and_T\ v_1,\ v_2 : T}\ [\text{and}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash or_T\ v_1,\ v_2 : T}\ [\text{or}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash xor_T\ v_1,\ v_2 : T}\ [\text{xor}]$$

Conversion Operations

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int_i \qquad T_2 : int_j, j < i}{\Gamma \vdash trunc\ T_1\ v\ to\ T_2 : T_2}\ \text{[trunc]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : float_i \qquad T_2 : float_j, j < i}{\Gamma \vdash fptrunc\ T_1\ v\ to\ T_2 : T_2}\ \text{[fptrunc]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int_i \qquad T_2 : int_j, j > i}{\Gamma \vdash zext\ T_1\ v\ to\ T_2 : T_2}\ \text{[zext]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int_i \qquad T_2 : int_j, j > i}{\Gamma \vdash sext\ T_1\ v\ to\ T_2 : T_2}\ \text{[sext]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : float_i \qquad T_2 : float_j, j > i}{\Gamma \vdash fpext\ T_1\ v\ to\ T_2 : T_2}\ \text{[fpext]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : float \qquad T_2 : int}{\Gamma \vdash fptoui\ T_1\ v\ to\ T_2 : T_2}\ \text{[fptoui]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : float \qquad T_2 : int}{\Gamma \vdash fptosi\ T_1\ v\ to\ T_2 : T_2}\ \text{[fptosi]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int \qquad T_2 : float}{\Gamma \vdash uitofp\ T_1\ v\ to\ T_2 : T_2}\ \text{[uitofp]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int \qquad T_2 : float}{\Gamma \vdash sitofp\ T_1\ v\ to\ T_2 : T_2}\ \text{[sitofp]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : ptr \qquad T_2 : int}{\Gamma \vdash ptrtoint\ T_1\ v\ to\ T_2 : T_2}\ \text{[ptrtoint]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : int \qquad T_2 : ptr}{\Gamma \vdash inttoptr\ T_1\ v\ to\ T_2 : T_2}\ \text{[inttoptr]}$$

$$\frac{\Gamma \vdash v : T_1 \qquad T_1 : ptr \qquad T_2 : ptr}{\Gamma \vdash bitcast\ T_1\ v\ to\ T_2 : T_2}\ \text{[bitcast]}$$

Control Flow Instructions

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash ret_T\ v : \bot}\ \text{[ret]}$$

$$\frac{\Gamma \vdash cond : i1}{\Gamma \vdash br\ cond\ l_t\ l_f : \bot}\ \text{[br]}$$

$$\frac{\Gamma \vdash v : T \qquad \Gamma \vdash v_k : T, 1 \leq k < n}{\Gamma \vdash switch_T\ v,\ v_1\ l_1,\ v_2\ l_2,\ ...,\ v_n\ l_n : \bot}\ \text{[switch]}$$

$$\frac{.}{\Gamma \vdash unreachable : \bot}\ \text{[unreachable]}$$

Memory Access and Indexing Operations

$$\frac{\cdot}{\Gamma \vdash v = alloca\ T : T^*}\ [\text{alloca}]$$

$$\frac{\Gamma \vdash p : T^*}{\Gamma \vdash load\ T^*\ p : T}\ [\text{load}]$$

$$\frac{\Gamma \vdash p : T^* \qquad \Gamma \vdash v : T}{\Gamma \vdash store\ T\ v,\ p : \bot}\ [\text{store}]$$

$$\frac{\Gamma \vdash p : T^* \qquad \Gamma \vdash i_k : i64 \qquad \Gamma \vdash 1 \leq k \leq n}{\Gamma \vdash gep\ T^*\ p,\ T_1\ i_1,\ T_2\ i_2,\ ...,\ T_n\ i_n : {T_n}^*}\ [\text{gep}]$$

Other Operations

$$\frac{\Gamma \vdash cond : i1 \qquad \Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T}{\Gamma \vdash select_T\ cond\ v_1,\ v_2 : T}\ [\text{select}]$$

$$\frac{\Gamma \vdash v_1 : T}{\Gamma \vdash v := v_1 : T}\ [\text{phi\_copy}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : int}{\Gamma \vdash icmp_T\ cond_o p\ v_1,\ v_2 : i1}\ [\text{icmp}]$$

$$\frac{\Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : T \qquad T : float}{\Gamma \vdash fcmp_T\ cond_o p\ v_1,\ v_2 : i1}\ [\text{fcmp}]$$

$$\frac{\Gamma \vdash v_k : T_k, 1 \leq k \leq N \qquad \Gamma \vdash f : T(T_1, T_2, ..., T_n)}{\Gamma \vdash call\ f(T_1\ v_1,\ T_2\ v_2,\ ...,\ T_n\ v_n) : T}\ [\text{call}]$$

4.4 Operational Semantics

We now describe the runtime checks needed in the form of an operational semantics for the LLVM IR. The execution environment consists of four parts.

1. S: A map from SSA variables to types
2. L: A map of memory locations to types.
3. M: A map of SSA variables to values.
4. H: A map of memory locations to values

The mappings S and L are provided externally and are similiar to the typing environment Γ used earlier. L is a map to track types of values stored at each memory location and $L[p] = T$ states that the data at the memory location p is of the type T. The map can be updated by $L[p \mapsto T]$ which sets the type for the memory location p to the type T. Similiarly for SSA values S can be updated as $S[v \mapsto T]$. Each access of the map reads/writes as many bytes as the size of the type being read/written. The design of the type maps is discussed in more detail in the next section.

We use an additional type $Mem(T)$ to indicate type T for an object in memory. This is a placeholder for type information read from the memory type map L. No restrictions are imposed on $Mem(T)$ except on a type check, in which case the type $Mem(T)$ is the same as T.

We now present the transitions allowed in the system; every other transition leads to an error. Each rule is of the form $S, L, M, H, I \rightarrow S', L', M', H', I'$. It indicates a transition from the execution environment on the left, with the instruction stream I to the execution environment on the right and the changed instruction stream. Each rule shows the transition for the statement at the head of the instruction queue. We show the type checks and the changes in the execution environment. The type equations shown in bold are the dynamic type checks that are needed to ensure that a particular execution is type safe. Details of how the type check is performed are given in the next section.

We use the notation $convert(n, T_1, T_2, op)$ to indicate the convertion of the value n of type T_1 to a value of type T_2 using the operation op. op can be $trunc$ for truncation, $zext$ for zero extension, $sext$ for sign extension and so on. It retuns the converted value. The semantics for these functions are well known and not detailed here. We use the operator $\circ$ as a place holder for the return value of a function call.

Binary Operations

$$\frac{\mathbf{S[v_1] == T} \quad \mathbf{S[v_2] == T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := add_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 + n_2], H)} \text{ [add]}$$

$$\frac{\mathbf{S[v_1] == T} \quad \mathbf{S[v_2] == T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := sub_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 - n_2], H)} \text{ [sub]}$$

$$\frac{\mathbf{S[v_1] == T} \quad \mathbf{S[v_2] == T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := mul_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 * n_2], H)} \text{ [mul]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := sdiv_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1/n_2], H)}\ \text{[sdiv]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := udiv_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1/n_2], H)}\ \text{[udiv]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := srem_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 \% n_2], H)}\ \text{[srem]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := urem_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 \% n_2], H)}\ \text{[urem]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := fadd_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 + n_2], H)}\ \text{[fadd]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := fsub_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 - n_2], H)}\ \text{[fsub]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := fmul_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 * n_2], H)}\ \text{[fmul]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := fdiv_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1/n_2], H)}\ \text{[fdiv]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := frem_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 \% n_2], H)}\ \text{[frem]}$$

Binary Bitwise Operations

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := shl_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 << n_2], H)}\ \text{[shl]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := lshr_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 >>> n_2], H)}\ \text{[lshr]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := ashr_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 >> n_2], H)}\ \text{[ashr]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := and_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 \& n_2], H)}\ \text{[and]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := or_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 | n_2], H)}\ \text{[or]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := xor_T\ v_1,\ v_2) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1 \oplus n_2], H)}\ \text{[xor]}$$

Conversion Operations

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := trunc\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, trunc)], H)} \text{ [trunc]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := fptrunc\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, fptrunc], H)} \text{ [fptrunc]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := zext\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, zext)], H)} \text{ [zext]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := sext\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto (n_1, T_1, T_2, sext)], H)} \text{ [sext]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := fpext\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, fpext)], H)} \text{ [fpext]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := fptoui\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, fptoui)], H)} \text{ [fptoui]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := fptosi\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, fptosi)], H)} \text{ [fptosi]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := uitofp\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, uitofp)], H)} \text{ [uitofp]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := sitofp\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto convert(n_1, T_1, T_2, sitofp)], H)} \text{ [sitofp]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{ptr} \quad M[v_1] = n_1}{(S, L, M, H, v := ptrtoint\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto n_1], H)} \text{ [ptrtoint]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := inttoptr\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto ptr], L, M[v \mapsto n_1], H)} \text{ [inttoptr]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T_1} \quad M[v_1] = n_1}{(S, L, M, H, v := bitcast\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto n_1], H)} \text{ [bitcast\_ssa]}$$

$$\frac{S[v_1] = Mem(T_3) \quad M[v_1] = n_1}{(S, L, M, H, v := bitcast\ T_1\ v_1\ to\ T_2) \rightarrow (S[v \mapsto Mem(T_3)], L, M[v \mapsto n_1], H)} \text{ [bitcast\_mem]}$$

Control Flow Instructions

$$\frac{\mathbf{S[cond]} == \mathbf{i1} \quad M[cond] = true}{(S, L, M, H, br\ cond\ l_t,\ l_f) \rightarrow (S, L, M, H, l_t)} \text{ [br\_true]}$$

$$\frac{\mathbf{S[cond]} == \mathbf{i1} \quad M[cond] = false}{(S, L, M, H, br\ cond\ l_t,\ l_f) \rightarrow (S, L, M, H, l_f)} \text{ [br\_false]}$$

$$\frac{\mathbf{S[v]} == \mathbf{T} \quad M[v] = n \quad n = c_k \quad 1 \leq k \leq n}{(S, L, M, H, switch_T\ v,\ c_1\ l_1,\ c_2\ l_2,\ ...,\ c_n\ l_n) \rightarrow (S, L, M, H, l_k)} \text{ [switch]}$$

Memory Access and Indexing Operations

$$\frac{}{(S, L, M, H, v := alloca\ T) \rightarrow (S[v \mapsto ptr], L[v \mapsto \bot], M[v \mapsto p], H)} \text{ [alloca]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{ptr} \quad M[v_1] = p \quad H[p] = n}{(S, L, M, H, v := load\ T^*\ v_1) \rightarrow (S[v \mapsto Mem(T_1)], L, M[v \mapsto n], H)} \text{ [load]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{ptr} \quad S[v] = T_1 \quad M[v_1] = p \quad M[v] = n}{(S, L, M, H, store\ T\ v,\ v_1) \rightarrow (S, L[p \mapsto T_1], M, H[p \mapsto n])} \text{ [store]}$$

Other Operations

$$\frac{S[v_1] = T_1 \quad M[v_1] = n_1 \quad M[cond] = true}{(S, L, M, H, v := select_T\ cond\ v_1,\ v_2) \rightarrow (S[v \mapsto T_1], L, M[v \mapsto n_1], H)} \text{ [select\_true]}$$

$$\frac{S[v_1] = T_2 \quad M[v_2] = n_2 \quad M[cond] = false}{(S, L, M, H, v := select_T\ cond\ v_1,\ v_2) \rightarrow (S[v \mapsto T_2], L, M[v \mapsto n_2], H)} \text{ [select\_false]}$$

$$\frac{S[v_1] = T \quad M[v_1] = n_1}{(S, L, M, H, v := v_1) \rightarrow (S[v \mapsto T], L, M[v \mapsto n_1], H)} \text{ [phi\_copy]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := icmp_T\ cond_op\ v_1,\ v_2) \rightarrow (S[v \mapsto i1], L, M[v \mapsto n_1 cond_o p n_2], H)} \text{ [icmp]}$$

$$\frac{\mathbf{S[v_1]} == \mathbf{T} \quad \mathbf{S[v_2]} == \mathbf{T} \quad M[v_1] = n_1 \quad M[v_2] = n_2}{(S, L, M, H, v := fcmp_T\ cond_o p\ v_1,\ v_2) \rightarrow (S[v \mapsto i1], L, M[v \mapsto n_1 cond_o p n_2], H)} \text{ [fcmp]}$$

$$\frac{\mathbf{S[v_0]} == \mathbf{ptr} \quad \mathbf{S[v_1]} == \mathbf{i64} \quad M[v_0] = p \quad M[v_k] = n_k \quad 1 \leq k \leq n}{(S, L, M, H, v := gep\ T^* v_0,\ T_1\ v_1,\ T_2\ v_2,\ ...,\ T_n\ v_n) \rightarrow (S[v \mapsto ptr], L, M[v \mapsto p + \Sigma(sizeof(T_k) * n_k)], H)} \text{ [gep]}$$

$$\frac{S[v_k] = T_k \quad M[v_k] = n_k \quad 1 \leq k \leq n}{(S, L, M, H, v := call\ f(T_1\ v_1,\ ...,\ T_n\ v_n); I) \rightarrow (S[f\_arg_k \mapsto T_k)], L, M[f\_arg_k \mapsto n_k], H, v := \circ; I)} \text{ [call]}$$

$$\frac{S[v] = T_1 \quad M[v_1] = n_1}{(S, L, M, H, v := \circ; ret_t\ v_1) \rightarrow (S[v \mapsto T_1], L, M[v \mapsto n_1], H, l_t)} \text{ [ret]}$$

$$\frac{(S, L, M, H, I) \rightarrow (S', L', M', H') \quad (S', L', M', H', I_2) \rightarrow (S'', L'', M'', H'')}{(S, L, M, H, I_1; I_2) \rightarrow (S'', L'', M'', H'')} \text{ [seq]}$$

The typechecks are inserted before all the type sensitive instructions by the instrumentation phase of our tool. The type checks ensure that the operands to the operation are of the correct type. If not, an error is flagged and the program cannot proceed further. In the semantics we show checks for all SSA values. However, for values that have a known type, e.g. the result of an add of two integers is known to be an integer, we do not need checks. We remove checks on such values as these can be proven statically to be true.

4.5 Memory Mapping

The memory map L tracks types for every byte in memory. For every byte, we track the type stored and a flag indicating the start of a value. Thus, for each pointer value we have a `(type, bool)` mapping. A similiar mapping is used for SSA values.

The map is initialized to $\perp$ indicating uninitialized memory. The mapping is updated on stores to memory. If a value of type T of size n is written to the location p, the following operations are performed.

$$L[p] = (T, true)$$

$$L[p+i] = (T, false), 1 \leq i < n$$

Metadata for all bytes being written to, is updated.

When a type check is performed, the types are matched for all the bytes being read. We also check that we are reading starting from the byte with the start sentinel. As all types are treated to be compatible with $\top$, if all the bytes being read are set to $\top$ the check passes. In such a case we subsequently set the type for the bytes read to the type read. This ensures that each location is only being used as a single type. The table 4.1 shows the details of the map L.

In Table 4.1, the left hand side indicates the query sent to the runtime from the instrumented program, while the right hand side illustrates the corresponding changes to the runtime type information. The $sizeof$ operator for a type gives the number of bytes that are modified, when a value of that type is written.

$L[p \mapsto T]$ $sizeof(T) = i$	$L[p] = (T, true)$ $L[p+k] = (T, false),\ 1 \leq k < i$
$L[p] == T$ $sizeof(T) = i$	$L[p] == (T, true)$ $L[p+k] == (T, false),\ 1 \leq k < i$
$L[p] == T$ $sizeof(T) = i$	$L[p+k] == \top,\ 0 \leq k < i$ $L[p] = (T, true)$ $L[p+k] = (T, false),\ 1 \leq k < i$

Table 4.1: Updating and Reading Shadow Memory

CHAPTER 5

IMPLEMENTATION

Our tool is built using LLVM 2.7, with LLVM-GCC as the front end. It handles all C/C++ language constructs, including indirect function calls and variadic functions. The tool works on the whole program, though it can be restructured to be run on individual files.

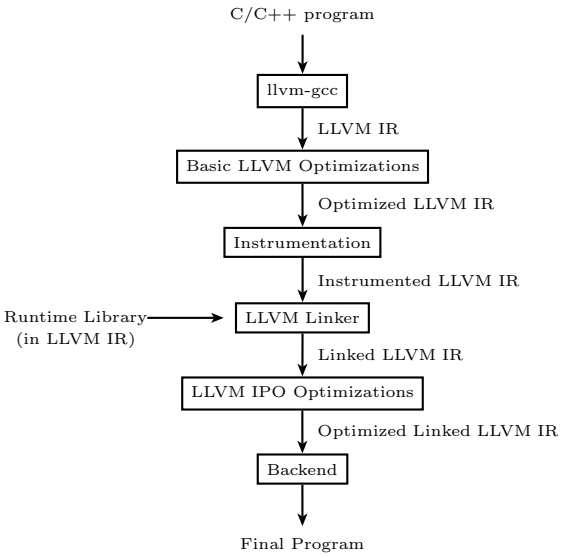

Figure 5.1: Tool Architecture

Figure 5.1 shows the various parts of the tool. The instrumentation phase inserts calls to the runtime type checks and other tracking functions. The instrumented program is then linked with the tool's runtime, which provides an implementation for the runtime checks. The instrumentation maintains all the semantics of the original function, and does not break compatibility with external code.

The instrumentation steps are described next, followed by a description of the runtime.

5.1 Instrumenting with Runtime Checks

The instrumentation phase of our tool recognizes the instructions that need type checks and adds calls to the runtime functions. In this section, we describe the instrumentation inserted for different LLVM constructs.

5.1.1 Initialization

We insert a call to the runtime initialization function which initializes the shadow memory. The initialization function is added to the list of constructor functions using the llvm `global_ctors` [17] global variable. The function is added as the highest priority constructor, making it the first function invoked at runtime.

5.1.2 Globals

All globals at the LLVM IR level are pointer values and may have initializers [17]. Our tool instruments the program to insert calls to initialize the metadata for all globals. For each initialized global, it looks at the type of the initializer and creates calls to the runtime. Globals without initializers are set to $\perp$.

For globals of primitive types like,

```
@nbody = internal global i32 0
```

the instrumentation is simple, and looks as follows,

```
%1 = bitcast i32* @nbody to i8* ;
call void @trackGlobal(%1, i8 3, i64 4)
```

where the arguments in order are the pointer, the type tag, the size of the type.

For globals of structure type, we initialize each field of the structure with its correct type tag. For the structure object shown below,

```
%1 = type { i32, i32 }
@obj = common global %1 zeroinitializer
```

we get the following instrumentation.

```
%10 = getelementptr inbounds %1* @obj, i64 0, i32 0 ;
%11 = bitcast i32* %10 to i8* ;
call void @trackGlobal(i8* %11, i8 2, i64 4)
%12 = getelementptr inbounds %1* @obj, i64 0, i32 1 ;
%13 = bitcast i32* %12 to i8* ;
call void @trackGlobal(i8* %13, i8 2, i64 4)
```

For globals that contain an array type element, we initialize the first element of the array to its type, and then copy the metadata value, to the rest of the elements. For the structure below that contains an array,

```
%0 = type { float, [10 x %1], double }
%1 = type { i32, i32 }
@obj = common global %0 zeroinitializer, align 32 ;
```

we initialize the first element of the array using calls to `trackGlobal` using `%3` and `%5`. The metadata is then copied to the rest of the elements, using a call to `trackArray`. A similiar approach is applied whenever an array type is encountered. The `trackArray` function takes as arguments the starting pointer, size of each element, and the total number of elements in the array.

```
%0 = getelementptr inbounds %0* @obj, i64 0, i32 0 ;
%1 = bitcast float* %0 to i8* ;
call void @trackGlobal(i8* %1, i8 1, i64 4)
%2 = getelementptr inbounds %0* @obj, i64 0, i32 1, i64 0, i32 0 ;
%3 = bitcast i32* %2 to i8* ;
call void @trackGlobal(i8* %3, i8 2, i64 4)
%4 = getelementptr inbounds %0* @obj, i64 0, i32 1, i64 0, i32 1 ;
%5 = bitcast i32* %4 to i8* ;
call void @trackGlobal(i8* %5, i8 2, i64 4)
%6 = getelementptr inbounds %0* @obj, i64 0, i32 1 ;
%7 = bitcast [10 x %1]* %6 to i8* ;
call void @trackArray(i8* %7, i64 8, i64 10)
%8 = getelementptr inbounds %0* @obj, i64 0, i32 2 ;
%9 = bitcast double* %8 to i8* ;
```

Calls to initialize the global variables are appended to the initialization function, so that other global constructors that use the globals access the correct type information.

5.1.3 `main` Function

If the `main` function takes two or more arguments, we insert calls to initialize the metadata for the arguments at the entry to the `main` function. The inserted call looks as follows.

```
  define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
  call void @trackArgvType(i32 %argc, i8** %argv)
  ...
```

For programs that also expect an enviroment argument to `main`, the transformed code looks as follows.

```
    define i32 @main(i32 %argc, i8** %argv, i8** %envp) nounwind {
entry:
    call void @trackArgvType(i32 %argc, i8** %argv)
    call void @trackEnvpType(i8** %envp)
    ...
```

5.1.4 Locals

Local variables in LLVM are allocated using `alloca` instructions [17]. It allocates memory on the stack frame of the function, depending on the size of the type, and returns a pointer. The tool instruments each `alloca` call, and adds calls to set the metadata to $\perp$. This ensures local variables are allowed to be used only after these have been initialized.

```
  %argc_addr = alloca i32 ;
  %0 = bitcast i32* %argc_addr to i8* ;
  %1 = sext i32 1 to i64 ; number of elements allocated
  %2 = mul i64 4, %1 ; to get size of memory allocated
  call void @trackUnInitInst(i8* %0, i64 %2)
```

5.1.5 Store Instructions

Each store instruction in the program is instrumented to update the metadata for the pointer. We add a call to the runtime function `trackStoreInst` that takes the pointer, the metadata value, and the size of the type.

```
%1 = bitcast i32* @NumNodes to i8* ;
call void @trackStoreInst(%1, i8 3, i64 4)
store i32 %tmp3, i32* @NumNodes, align 4
```

5.1.6 Load Instructions

Each load instruction in the program is instrumented to read the metadata associated with the pointer being read. Every time the value that has been loaded is used in an instruction we insert a *type check*. Adding the check at the use of the loaded value ensures that no false positives occur due to dynamically dead loads.

```
%ptr_src = bitcast i8** %tmp1 to i8* ;
;read type info for %211 into %MD
call void @getTypeTag(i8* %211, i64 8, i8* %MD)
%tmp2 = load i8** %tmp1, align 8
;compare %MD against the type tag 1 for 8 bytes.
call void @checkType(i8 1, i64 8, i8* %MD, i8* %ptr_src)
%tmp3 = call i32 (...)* @atoi(i8* %tmp2) nounwind ;
```

As per the design in the previous section, instrumentation at the uses varies depending on the instruction the value is used in. Specifically, for store instructions we simply copy the type metadata to the metadata for the pointer used in the store. Similiarly, for casts, we push the checks to the uses of the casts instead of the casts themselves as these do not modify the value in any way.

```
%elt55 = getelementptr inbounds %1* %tmp257, i32 0, i32 0 ;
%295 = bitcast i64* %elt55 to i8* ;
;read type info for %295 into %126
call void @getTypeTag(i8* %295, i64 8, i8* %126)
%val56 = load i64* %elt55 ;
```

```
%tmp1 = getelementptr inbounds %1* %tmp, i32 0, i32 0 ;
%297 = bitcast i64* %tmp1 to i8* ;
;set type info for %297, using the type info in %126
call void @setTypeInfo(i8* %297, i8* %126, i64 8, i8 6)
store i64 %val56, i64* %tmp1
```

5.1.7 Function Calls

Type Tags for Arguments

We need to track types only for arguments that come from `load` instructions since all other SSA values are statically typed in the IR. Where type information is needed we clone functions to pass in the value in memory instead of as an SSA value. This change is done recursively and hence the information is propogated to all callees. This implies that we do not need to explicitly add type tags at call sites. This cloning can be done as a pre-processing pass and the checks are then inserted by the instrumentation phase. Types of all actual arguments are checked at call sites.

Byval Functions

LLVM allows function arguments to have the `byval` [17] attribute. This implies that the argument is passed *by value* i.e. a copy of the actual argument is passed as argument. The `byval` attribute is only allowed on pointer arguments. The tool must modify these functions to associate the type metadata of the actual argument with the copy being passed.

This is achieved by making an internal clone of the function that makes the copying of the parameter explicit. We remove the `byval` attribute, and instrument the entry of the function to make a copy of the argument. We copy the metadata to the new object, and modify all callers to instead call the modified function without the `byval` attribute.

If the function in question is externally visible, we create an internal clone which we modify as described above. For the externally callable function, we set the metadata for the byval argument to $\top$.

Variadic Functions

To be able to typecheck the arguments passed into a vararg function, it is necessary to know the types of all the arguments passed in the `va_list` structure. This ensures that the `va_arg` calls made to the list are accessing the correct types. We instrument all calls to internal varargs functions to add arguments for the total number of arguments and an array with the type tag for each argument. As the types of the arguments at the call site determine how the arguments are passed, and the type of `va_arg` call determines how the arguments are accessed, the size of the argument types is not strictly necessary, and hence we pass only type information.

```
%1 = call i32 (i32, ...)* @get(i32 0, i32* %i, float* %f) ;
```

is transformed to

```
%5 = alloca i8, i32 3 ;
%15 = getelementptr inbounds i8* %5, i32 0 ;
store i8 2, i8* %15
%16 = getelementptr inbounds i8* %5, i32 1 ;
store i8 1, i8* %16
%17 = getelementptr inbounds i8* %5, i32 2 ;
store i8 1, i8* %17
%18 = call i32 (i64, i8*, i32, ...)*
      @get.vrg(i64 3, i8* %5, i32 0, i32* %i, float* %f) ;
```

At the entry to each vararg function, we subtract from the total arguments passed, the number of fixed arguments and get the number of arguments in the `va_list`. Similiarly, we increment the pointer to the start of the metadata array by the number of the fixed arguments to get the start of the metadata for the elements in the `va_list`. For the original function `get` shown,

```
define internal i32 @get(i32 %unused, ...)
```

the transformed function looks as follows

```
define internal i32 @get.vrg(i64 %TotalArgCount, i8* %MD,
                                        i32 %unused, ...) {
entry:
  %0 = alloca i8, i32 4 ;
```

```
%1 = alloca i8, i32 4 ;
%2 = alloca i8, i32 4 ;
%3 = alloca i8, i32 8 ;
%4 = alloca i8, i32 4 ;
%5 = alloca i8, i32 8 ;
%varargs.count = sub i64 %TotalArgCount, 1 ;
%varargs.MD = getelementptr inbounds i8* %MD, i64 1 ;
...
```

For every call to `va_start` we insert a call to initialize the metadata for the `va_list` used in the call. For a call to `va_copy` we add a call to copy the metadata to the destination `va_list`.

```
call void @setVAInfo(i8* %ap, i64 %varargs.count, i8* %varargs.MD)
call void @llvm.va_start(i8* %ap)

call void @copyVAInfo(i8* %ap_copy, i8* %ap)
call void @llvm.va_copy(i8* %ap_copy, i8* %ap)
```

On every call to `va_arg`, we insert calls to the type of the value being read against the metadata.

```
%24 = bitcast %struct.__va_list_tag* %ap_copy15 to i8* ;
call void @checkVAArgType(i8* %24, i8 1) // 1 = ptr type
%25 = va_arg %struct.__va_list_tag* %ap_copy15, i32* ;
```

Indirect Function Calls

We transform all indirect calls to add arguments as in the case of variadic functions. This is to allow the instrumented version of the variadic function to be called from an indirect call site. We also clone all the address taken functions to add arguments. All internal uses of the original function are replaced with the clone. A call graph analysis could be used to selectively clone call sites that call variadic functions.

Library Functions

Linking instrumented code with uninstrumented library functions can lead to false positives since the uninstrumented code does not update the type metadata. Memory intialized by library functions will be marked as uninitialized and result in warnings. Some standard C library functions like `memcpy, memset, strcpy, strcat` are frequently used and lead to a lot of false positives if not handled. We identify calls to functions such as these and initialize the type information as it should be. Similiarly, to handle C++ code, we added wrappers for some C++ library functions, especially `std::string` functions.

We also identify calls to functions like `time, getrusage,__ctype_b_loc` that return data in static buffers which are subsequently accessed by user code. We initialize the metadata for the memory returned by such functions to $\top$.

5.2 Type Tracking Runtime

The instrumented code is linked with a library that provides an implementation of the runtime functions to track and check for types. Different implementations of the runtime can be used as needed, e.g. with varying degrees of debug information, with more efficient data structures to store the type tags, and different behavior on error.

We implemented a runtime that uses byte granularity shadow memory to track types. It is built to be efficient and only one operation is required to get the location of the metadata in the common case. We flag warnings when a type mismatch is encountered. The user can inspect the warnings and modify the program to make it type safe. This is useful for debugging purposes, giving multiple warnings in a single run.

The `x86_64` architecture user space addresses range from `0x000000000000` to `0x7fffffffffff`. We allocate 2^{46} bytes of shadow memory. It is located in the address range `0x2aaaac01e000` to `0x6aaaac01e000`. The function to map a pointer to its metadata is given below.

```
BASE = 0x2aaaac01e000 ;
END  = 0x6aaaac01e000 ;
```

```
SIZE = 1L << 46 ;
meta_offset = (ptr >= END) ? ptr-SIZE : ptr ;
meta_ptr = BASE + meta_offset ;
```

Table 5.1 gives a summary of all the runtime functions. We describe some of them in more detail below.

5.2.1 Initialization

The initialization function allocates the shadow memory for the storing the type metadata. The shadow memory is located at a fixed address, and occupies half the address space. It is initialized to $\perp$ (0).

5.2.2 Setting Metadata

The `trackStoreInst` and `trackGlobal` methods set metadata for a given pointer. For a particular pointer, we calculate the location of its metadata using the mapping function discussed above. The first byte of the metadata is then set at the type value given and the size is used to set the remaining bytes to a special *middle of the object* marker (`0xFE`).

5.2.3 Initialize Arguments to `main`

The tool assumes that the arguments passed to `main` are initialized and well formed null terminated strings. We walk through the `argv` and `envp` arrays, and initialize the metadata for the pointers stored in them.

5.2.4 Reading Metadata

The `getTypeTag` function reads the metadata for a given pointer and stores it in the memory provided. It reads `size` bytes.

5.2.5 Checking Metadata

When we have to compare the metadata to a given type, we check that the type information stored on the first byte matches the given type tag, and

that the rest of the bytes being read contain *middle of the object* markers. This ensures that we do not read across object boundaries.

5.2.6 Metadata for `va_list`

For each `va_list`, we keep track of the total number of arguments it contains, the metadata for each of them, and a counter to track the number of arguments read. The counter is initialized to 0, when we initialize the metadata. Every call to check the type of the next argument in a given `va_list` increments the counter and ensures that it is within the total number of arguments, and then checks the metadata for that argument. When we copy the metadata to a new list, the counter is copied as well.

5.3 Instrumentation Options

To allow for different levels of type safety, the user can allow/disallow certain instrumentations.

1. **Check Pointer Types:** The instrumentation by default treats all pointer types as the same type. If the user desires stricter checking, he may enable tracking of multiple pointer types, by using the `-enable-ptr-type-checks` flag to the tool.

2. **Track All Loads:** By default, checks happen at the uses of the loads, so as to reduce the number of false positives from dynamically dead loads. But if a system were concerned about checking every load exectued, this can be enabled by using the `-track-all-loads` flag.

3. **Checking Pointer Comparisons:** A comparison operation depends only on the value stored in a pointer, not its type. Pointers of different types can be compared against each other. Checks on pointer comparisons can be turned off using the `-no-ptr-cmp-checks`. Turning off these checks helps reduce overhead, and might reduce false positives in some programs.

Runtime Function	Description
`void shadowInit()`	Allocates and initializes the shadow memory
`void trackArgvType(int argc, char **argv)` `void trackEnvpType(char **envp)`	Initializes the metadata for the strings passed as arguments to `main`
`void trackGlobal(void *ptr, uint8_t typeNumber, uint64_t size)`	Initialize `size` bytes of metadata for the `ptr` with the value `typeNumber`
`void trackArray(void *ptr, uint64_t size, uint64_t count)`	Reads `size` bytes of metadata starting at the `ptr` and stores it at `(ptr + i*size)` for `1 < i < size`
`void trackStoreInst(void *ptr, uint8_t typeNumber, uint64_t size)`	Sets the type for `ptr` to `typeNumber`. It updates metadata for `size` bytes
`void trackInitInst(void *ptr, uint64_t size)` `void trackUnInitInst(void *ptr, uint64_t size)`	Sets the metadata for `size` bytes, starting at `ptr` to $\top$ and $\bot$ respectively
`void setVAInfo(void *va_list, uint64_t totalCount, uint8_t *metadata_arr)`	Initialize the metadata for the `va_list` with given count and metadata array. Initializes variables read to 0.
`void copyVAInfo(void *va_list_dst, void *va_list_src)`	Copy the metadata for the source `va_list` to the destination `va_list`
`void checkVAArgType(void *va_list, uint8_t TypeAccessed)`	For the given `va_list`, increments count of variables read, compares it against the total number of variables. Then checks the type being accessed against the value in the metadata array.
`void copyTypeInfo(void *dstptr, void *srcptr, uint64_t size)`	Copies `size` bytes of metadata for the given `srcptr` to the metadata for `dstptr`
`void trackStringInput(void *ptr)`	Initializes `strlen(ptr) + 1` bytes of metadata starting at `ptr` Used for library functions that return strings.
`void getTypeTag(void *ptr, uint64_t size, uint8_t *dest)`	Reads `size` bytes of metadata for the pointer `ptr` and stores it in `dest`
`void checkType(uint8_t typeNumber, uint64_t size, uint8_t *metadata, void *ptr)`	`metadata` contains the metadata for `ptr` read from the shadow memory. We now compare it against `typeNumber`. We flag a warning if a mismatch is detected
`void setTypeInfo(void *dstptr, uint8_t *metadata, uint64_t size, uint8_t type)`	Sets type metadata for `dstptr`. If `metadata` not `NULL`, i.e. the value came from a load, we set the copy stored in `metadata`. However, if it is `NULL`, the value came from an SSA value, and we use the type tag `type` and size `size` from the type of the SSA value.

Table 5.1: Runtime Functions

CHAPTER 6

OPTIMIZATION

To improve the performance of our dynamic type checker, we use static type inference to reduce the number of dynamic checks required. The static type inference algorithm finds objects used in a type consistent fashion allowing us to eliminate checks on them. We can also reduce the amount of propogation of type metadata needed for these objects.

6.1 Static Analysis

DSA [11] is a pointer analysis that also infers types for all pointer objects. The original DSA algorithm inferred a single type for a DSNode, the representation of a memory object, when possible. If such a type could not be inferred, the node was *Collapsed*, merging all its outgoing pointers. In this implementation, type safe nodes could be identified as the non-collapsed nodes. We have enhanced the algorithm to improve the type inference by basing it on primitive typing.

DSA now tracks types at each offset inside the object. Instead of inferring single structure or array type for a given object, we can now infer a *set of types* at a given offset in the object. If there are no conflicting type accesses at all offsets inside an object we consider that object typesafe. We also restrict the types we track to the primitive types. We believe this reflects the right degree of flexibility in the use of structures and arrays in C/C++ by enforcing types only on the primitive values which are being used in computation. The inference is based only on the use of an object in a type sensitive operation, like an `add`. Operations like `bitcast` which do not alter the representation of a value and are not strictly type sensitive do not influence the type inference.

6.1.1 Structure Typing

The original type inference algorithm in DSA did not account for the physical subtyping cases, such as the implementation of object oriented behaviour using `structs` in C. In the following example [12], the function `foo` could take an argument of any subclass of type `Point`.

```
typedef struct {
  int x, y;
} Point;
typedef struct {
  int x, y, color;
} ColorPoint;

void foo(Point *p_arg) {
  p->x = ...
  p->y = ...
}

main() {
  Point *p;
  ColorPoint *pcp;

  ...

  foo(p);
  foo(pcp);
}
```

As DSA only inferred a single type for a given object, we would have failed to infer a type for `p_arg`, once its DSNode is merged with `p` and `cpc` in the context insensitive analysis.

The DSA algorithm has been extended to recognize types at various byte offsets within a given DSNode. The new algorithm infers the following DS-Graph for foo. All the fields of the object being accessed are recorded along with the type used to read/write to them.

This allows us to remove all checks on accesses to `p_arg` because all its fields are only used as a single type.

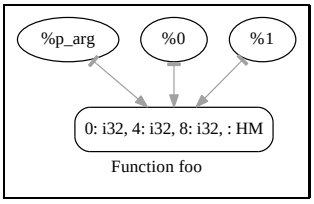

Figure 6.1: DSGraph for correct structure subtyping

However, if a user were to make an error while subtyping as in the following example,

```
typedef struct {
  float x,y;
} FloatPoint;
```

calling `foo` with this object would result in the following DSGraph. Such a DSNode would still need checks on all accesses to it, reporting errors at runtime.

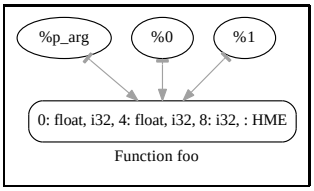

Figure 6.2: DSGraph for incorrect structure subtyping

6.1.2 Array Type Inference

The original DSA implementation did not identify arrays inside structure objects in most cases, causing a large number of collapsed nodes. We have modified DSA to identify the types of arrays inside structures in cases where we use proper structure indexing to index the array. We do not distinguish the outgoing pointers from different elements in the array, merging all the nodes they point to. This is however still much better than collapsing the node altogether. We get DSGraph shown for `struct bpnode` defined as shown.

```
struct bpnode {
  short i1;
  short i2;
  int j;
  double k;
```

```
  double arr[3];
};
```

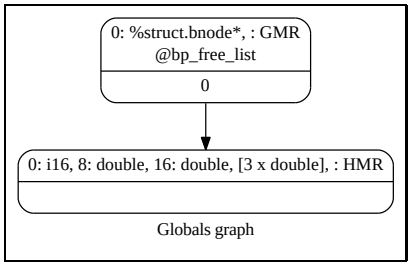

Figure 6.3: DSGraph showing array typing

6.1.3 Optimizing the Dynamic Tracking

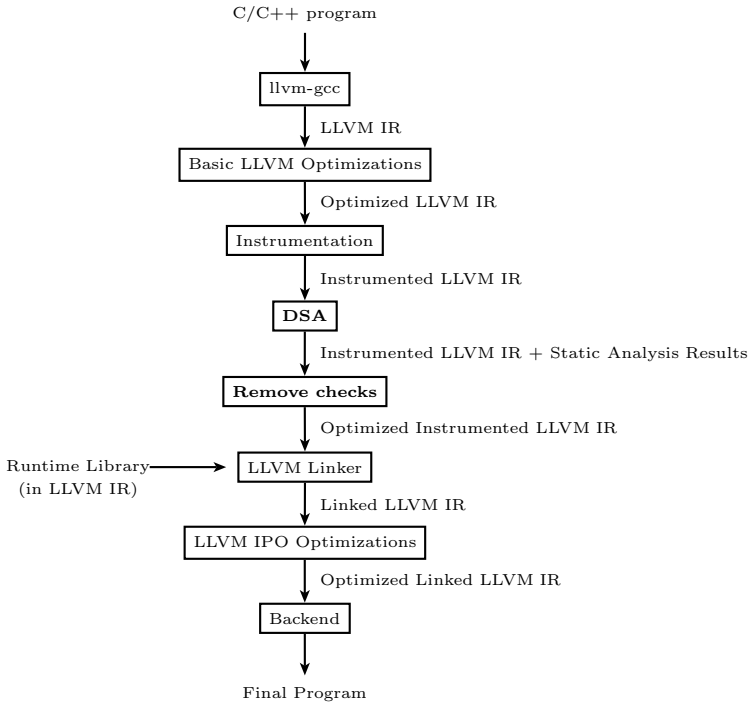

Figure 6.4: Tool Architecture with Static Analysis

Figure 6.4 shows the tool architecture when the static analysis is used to optimize the result. Two new stages are added as compared to Figure 5.1. DSA is used to analyze the instrumented code to identify the type safe nodes. Using the results of the static analysis, the instrumented code is modified as follows,

1. For typesafe objects we remove all type checks.

2. We initialize all typesafe objects to be null because the static analysis results are not sound in the presence of uninitialized memory.

3. For all typesafe memory objects, the metadata is initialized to $\top$. This ensures that we are still able to catch uninitialized memory access errors. This is also needed in cases where the non fully context insensitive nature of DSA does not alias certain pointers, leaving them with different completeness flags in different contexts. Unfortunately, this means we incur the overhead of tracking type information, even if the program is largely type safe. We hope to address this issue in the future.

The static analysis results are only applicable in the absence of memory safety errors like dangling pointer errors and out-of-bounds accesses. These can be detected by simultaneously checking with tools like SAFECode [18], which ensures sound type inference in the presence of memory safety errors. Overheads of this technique can be reduced by adopting faster runtimes as suggested by Baggy Bounds Checking [16], WIT [19] or SoftBound [15]. Protection against dangling pointer attacks can also be achieved by using garbage collection [5], as used by CCured [13].

6.2 Optimizing the Runtime

We have optimized our runtime type checking functions in the following way.

1. We allocate our shadow memory at a fixed address. This eliminates extra loads needed to access the metadata.

2. We have also optimized the type check for the common case of a type match by making it the fast path. This has been done by making this the first condition checked. The checks for a mismatch or match with $\top$ have been placed in a slow path check.

3. We have reduced the number of typechecks inserted on the uses of a particular load by removing checks that are dominated by other checks for the same SSA value.

CHAPTER 7

ANALYSIS OF TYPE ERRORS

We now look at examples from the programs that reported type errors and examine their sources.

7.1 Sqlite

7.1.1 Mismatched Format String

Listing 7.1: Format String Error

```
sqlite3MPrintf(db, "CREATE%s INDEX %.*s",
               onError==OE_None ? "" : " UNIQUE",
               pEnd->z - pName->z + 1,
               pName->z);
```

Sqlite defines its own version of the various printing functions. This allows the tool to detect format string errors. In the Listing 7.1, the format string expects an `i32` as its 2nd parameter. However, the value passed is a function of `pEnd->z` and `pName->z`, both pointers, and equivalent to `i64` on our system. As the function is variadic, no cast is inserted. We reported this error and it has subsequently been fixed[1].

7.1.2 Uninitialized Memory

Our tool also helps detect an error when uninitialized or unallocated memory is being read. In the Listing 7.2, the contents of `pFrom` are copied to `pTo`. The program then reads `pTo->z`. This is a utility function called from multiple places in the code. When it is called from `sqlite3VdbeExec`, the value of `pTo`

[1] `http://www.sqlite.org/src/ci/90cfeaf7b6`

is initialized in `sqlite3VdbeSerialGet` which initializes the `pTo` structure; however, in some cases it does *not* initialize the `z` field. This leads to an error being flagged at this location when we try and read the value. However, the variable `zData` is never used. This error is only flagged when we use the tool with the `-track-all-loads` option, that checks every load.

Listing 7.2: Uninitialized variable read

```
SQLITE_PRIVATE int sqlite3VdbeMemCopy(Mem *pTo, const Mem *pFrom){
  int rc = SQLITE_OK;
  char *zBuf = 0;

  /* If cell pTo currently has a reusable buffer, save a pointer to it
   ** in local variable zBuf. This function attempts to avoid freeing
   ** this buffer.
   */
  if( pTo->flags&MEM_Dyn ){
    if( pTo->xDel ){
      sqlite3VdbeMemRelease(pTo);
    }else{
      zBuf = pTo->z;
    }
  }

  /* Copy the contents of *pFrom to *pTo */
  memcpy(pTo, pFrom, sizeof(*pFrom));

  if( pTo->flags&(MEM_Str|MEM_Blob) && pTo->flags&MEM_Static ){
    /* pFrom contained a pointer to a static string. In this case,
     ** free any dynamically allocated buffer associated with pTo.
     */
    sqlite3_free(zBuf);
  }else{
    char *zData = pTo->z;

    pTo->z = zBuf;
    pTo->flags &= ~(MEM_Static|MEM_Ephem);
    pTo->flags |= MEM_Dyn;
    pTo->xDel = 0;

    if( pTo->flags&(MEM_Str|MEM_Blob) ){
      if( sqlite3VdbeMemGrow(pTo, pTo->n+2, 0) ){
        pTo->n = 0;
        rc = SQLITE_NOMEM;
      }else{
        memcpy(pTo->z, zData, pTo->n);
        pTo->z[pTo->n] = '\0';
        pTo->z[pTo->n+1] = '\0';
        pTo->flags |= MEM_Term;
      }
    }
  }
  return rc;
}
```

7.2 186.crafty

7.2.1 Switching Union Fields

The 186.crafty benchmark in two functions interchangeably uses the two fields of a union. Listing 7.3 shows one of them. The other function is

`LastOne` and does something similiar. Here, the same memory location is written to as an `i64`, and read as multiple `i16`s.

The C standard allows programmers to read bytes from the middle of an object, as long as they do not cross object boundaries. We still flag this as an error to bring to the programmers attention such an access of an object. Accessing bytes by doing pointer arithmetic, may be unsafe if the endianness is not taken into consideration in certain applications. A better practice would be to read the value and perform bitshifts to get the correct bytes. The warning generated hints to the programmer to use the more typesafe option.

Listing 7.3: Using Union Fields

```
int FirstOne(BITBOARD arg1)
{
  union doub {
    unsigned short i[4];
    BITBOARD d;
  };
  union doub x;
  x.d=arg1;
#  if defined(LITTLE_ENDIAN_ARCH)
  if (x.i[3])
    return (first_ones[x.i[3]]);
  if (x.i[2])
    return (first_ones[x.i[2]]+16);
  if (x.i[1])
    return (first_ones[x.i[1]]+32);
  if (x.i[0])
    return (first_ones[x.i[0]]+48);
#  endif
#  if !defined(LITTLE_ENDIAN_ARCH)
  if (x.i[0])
    return (first_ones[x.i[0]]);
  if (x.i[1])
    return (first_ones[x.i[1]]+16);
  if (x.i[2])
    return (first_ones[x.i[2]]+32);
  if (x.i[3])
    return (first_ones[x.i[3]]+48);
#  endif
  return(64);
}
```

7.3 464.h264ref

7.3.1 Testing Endianness

As Listing 7.4 shows, there is a type error when the values is written and read as different types. However, this is most likely not security critical. We currently do not allow reading subparts of an object. We hope to add that in the future reducing false positives as these.

Listing 7.4: Testing Endianness

```
/*!
 *****************************************************************************
 * \brief
 *      checks if the System is big- or little-endian
 * \return
 *      0, little-endian (e.g. Intel architectures)
 *      1, big-endian (e.g. SPARC, MIPS, PowerPC)
 *****************************************************************************
 */
int testEndian()
{
  short s;
  byte *p;

  p=(byte*)&s;

  s=1;

  return (*p==0);
}
```

7.4 471.omnetpp

7.4.1 Passing `NULL` to a varargs function

Listing 7.5: Passing `NULL`

```
static sEnumBuilder _EtherMessageKind( "EtherMessageKind",
    JAM_SIGNAL, "JAM_SIGNAL",
    ETH_FRAME, "ETH_FRAME",
    ETH_PAUSE, "ETH_PAUSE",
    ETHCTRL_DATA, "ETHCTRL_DATA",
    ETHCTRL_REGISTER_DSAP, "ETHCTRL_REGISTER_DSAP",
    ETHCTRL_DEREGISTER_DSAP, "ETHCTRL_DEREGISTER_DSAP",
    ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
    0, NULL
);


sEnumBuilder::sEnumBuilder(const char *name, ...)
{
    cEnum *e = findEnum(name);
    if (!e)
    {
        e = new cEnum(name);
        e->setOwner(&enums);
    }

    va_list va;
    va_start(va,name);
    for(;;)
    {
        int key = va_arg(va,int);
        const char *str = va_arg(va,const char *);
        if (!str)
            break;
        e->insert(key,str);
    }
}
```

As Listing 7.5 shows, `NULL` is passed to a varargs function without an explicit cast. The compiler passes it as an `i64` while the function being called expects a string leading to a potential problem. For calls to varargs

functions if `NULL` is passed as a variable argument, it should be explicitly cast to the correct type [20].

7.5 gs

7.5.1 Float Comparison Macros

Listing 7.6: Float comparison macros

```
#  define _f_as_l(f) *(long *)(&(f))
#  define is_fzero(f) ((_f_as_l(f) << 1) == 0)  /* +0 or -0 */
#  define is_fzero2(f1,f2) (((_f_as_l(f1) | _f_as_l(f2)) << 1) == 0)
#  define is_fneg(f) ((_f_as_l(f)) < 0) /* -0 is negative, oh well */
```

Listing 7.6 shows code that attempts to determine whether a floating point value is zero or not. The macros explicitly read a floating point value as a `long`. Such tricks employed by programmers for efficiency reasons are hard to differentiate from real type errors.

7.6 units-1.88

7.6.1 Uninitalized Variable Read

Listing 7.7 shows code that attempts to count the number of newline characters read in the variable `count`. However, the function `fgetslong` uses uninitialized variable `dummy` if the `count` is `NULL`. This variable is subsequently accessed in both `fgetslong` and `fgetscont` leading to warnings.

The value of `dummy` is not used for any result computation subsequently in the code. If a programmer were to use these functions, without knowing that `dummy` is uninitialized, it could lead to problems. Thus, the tool can be useful in identifying benign errors before these cause problems.

Listing 7.7: Uninitialized variable `dummy`

```
/*
   Fetch a line of data with backslash for continuation.  The
   parameter count is incremented to report the number of newlines
   that are read so that line numbers can be accurately reported.
*/

char *
fgetscont(char *buf, int size, FILE *file, int *count)
{
  if (!fgets(buf,size,file))
    return 0;
  (*count)++;
  while(strlen(buf)>=2 && 0==strcmp(buf+strlen(buf)-2,"\\\n")){
    (*count)++;
    buf[strlen(buf)-2] = 0; /* delete trailing \n and \ char */
    if (strlen(buf)>=size-1) /* return if the buffer is full */
      return buf;
    if (!fgets(buf+strlen(buf), size - strlen(buf), file))
      return buf;  /* already read some data so return success */
  }
  if (buf[strlen(buf)-1] == '\\') {   /* If last char of buffer is \ then   */
    ungetc('\\', file);               /* we don't know if it is followed by */
    buf[strlen(buf)-1] = 0;           /* a \n, so put it back and try again */
  }
  return buf;
}


/*
   Gets arbitrarily long input data into a buffer using growbuffer().
   Returns 0 if no data is read.  Increments count by the number of
   newlines read unless it points to NULL.
*/

char *
fgetslong(char **buf, int *bufsize, FILE *file, int *count)
{
  int dummy;
  if (!count)
    count = &dummy;
  if (!*bufsize) growbuffer(buf,bufsize);
  if (!fgetscont(*buf, *bufsize, file, count))
    return 0;
  while ((*buf)[strlen(*buf)-1] != '\n' && !feof(file)){
    growbuffer(buf, bufsize);
    fgetscont(*buf+strlen(*buf), *bufsize-strlen(*buf), file, count);
    (*count)--;
  }
  return *buf;
}
```

7.7 ks

7.7.1 Mismatched argument type

Listing 7.8 shows a macro used as an error handler. If the program cannot open the input file, it tries to output an error message. The argument `inFile` passed to `fprintf` is of `FILE *` type instead of the expected `i8*`. This error is also reported by CCured [13]. We only detect this error if tracking is on for tracking different pointer types.

Listing 7.8: Wrong argument type to `printf`

```
/* simple exception handler */
#define TRY(exp, accpt_tst, fn, fail_fmt, arg1, arg2, arg3, fail_action) { \
              (exp); \
              if (!(accpt_tst)) { \
                  fprintf(stderr, "(%s:%s():%d): ", __FILE__, fn, __LINE__); \
                  fprintf(stderr, fail_fmt, arg1, arg2, arg3); \
                  fprintf(stderr, "\n"); \
                  fail_action; \
                  } \
                  }
void
ReadNetList(char *fname)
{
    FILE *inFile;
    char line[BUF_LEN];
    char *tok;
    unsigned long net, dest;
    ModulePtr node, prev, head;

    TRY(inFile = fopen(fname, "r"),
        inFile != NULL, "ReadData",
        "unable to open input file [%s]", inFile, 0, 0,
        exit(1));

    TRY(fgets(line, BUF_LEN, inFile),
        sscanf(line, "%lu %lu", &numNets, &numModules) == 2, "ReadData",
        "unable to parse header in file [%s]", inFile, 0, 0,
        exit(1));
     ...
}
```

7.8 099.go

7.8.1 Out-of-bounds array access

Listing 7.9 shows a loop that access the array `diffs4` beyond its declared size. Even though we do not explictly track array bounds, since that memory is uninitialized this access causes a type error and is detected. This is also reported by CCured [13].

Listing 7.9: Array Bounds error

```
int diffs4[5][3];
int diffs4i[7][3] =
{
{  1,18, 1 },  /* square */
{  1, 1,18 },  /* pyramid */
{ 18, 1,19 },  /* pyramid */
{ 18, 1, 1 },  /* pyramid */
{ 19, 1,18 },  /* pyramid */
{  1, 1, 1 },  /* straight line */
{ 19,19,19 },  /* straight line */
};

void deadshape(int g,int rn){
   int pointlist,size,count;
   int ptr,i,j,diffs[5],ldtmp;
   eyeval[rn] = eyepot[rn] = eyemin[rn] = 8;
   size = grsize[g];
   pointlist = EOL;
   for(ptr = grpieces[g]; ptr != -1; ptr = mvnext[ptr])
      addlist(mvs[ptr],&pointlist);

   i = 0;
   for(ptr = pointlist; links[ptr] != EOL; ptr = links[ptr]){
      diffs[i] = list[links[ptr]] - list[ptr];
      ++i;
      }
   killist(&pointlist);
   if(size == 4){
      for(j = 0; j < 3; ++j)
         if(diffs4[0][j] != diffs[j])break;
         else if(j == 2)return; /* 4 in square */
      for(i = 1; i < 5; ++i)
         for(j = 0; j < 3; ++j)
            if(diffs4[i][j] != diffs[j])break;
            else if(j == 2)return;  /* 4 in pyramid */
      eyeval[rn] = eyemin[rn] = eyepot[rn] = 16; /* 4 in line */
      for(i = 5; i < 7; ++i)
         for(j = 0; j < 3; ++j)
            if(diffs4[i][j] != diffs[j])break;
            else if(j == 2)return;  /* 4 in straight line */
      for(ptr = grlbp[g]; ptr != EOL; ptr = links[ptr]){
               i = fdir[list[ptr]];
               count = 0;
               for(ldtmp = ldir[i]; i < ldtmp; ++i)
                       if(board[list[ptr]+nbr[i]] == g)++count;
...
}
```

CHAPTER 8

PERFORMANCE

In this section we provide a performance evaluation of our tool. To measure the execution time overhead introduced by our tool, we instrumented examples from the Olden [21],PtrDist, SPECINT 2000 and SPECINT 2006 benchmark suites. The benchmarks were compiled using `llvm-gcc` 2.7, and executed with the same inputs with and without instrumentation(with the `-no-ptr-cmp-checks` flag) on an x86_64 machine running Linux with 8GB of RAM. The times reported are the median of 3 executions. Error logging was turned off as it caused large overheads. The baseline was calculated using GCC with -O3. Table 8.1 shows the overhead of applying our tool, to these benchmarks. The last column shows the reduced overhead after optimizing using the static type inference discussed earlier. Some benchmarks were too large for DSA and were not optimized.

The overheads are quite high as the type information is continually updated throughout the execution. However, it is much lower than the reported overheads for all the other dynamic type checking tools [10, 8, 9, 22]. This is because all SSA variables are statically typed, and do not require runtime type information. The overhead is significantly reduced when type checks are removed based on the results of the static type inference.

The overheads with the static analysis in place are much lower, around 4x, though still higher than the reported overhead of CCured(3-87%). Their type inference is more sophisticated than ours, specially in the case of arrays.

The reduced overhead of 4x on incorporating the static analysis allows the use of this tool during the development cycle of a software, for the detection of type errors. The fact that it is automatic and has a low false positive rate also makes it feasible for real world C programs. We have used our tool on system softwares like squid, sqlite, thhtpd.

Benchmark	LOC	Base Time	Instrumented Time	Instrumented Overhead (Ratio)	Optimized Time	Optimized Overhead (Ratio)
Olden						
bh	2073	1.91	9.73	5.09	4.90	2.56
bisort	350	0.80	2.55	3.19	2.55	2.88
em3d	688	2.71	34.98	12.91	34.12	12.59
health	502	0.41	1.14	2.78	1.08	2.63
mst	428	0.12	0.50	4.17	0.42	3.50
perimeter	484	0.26	1.00	3.85	0.89	3.42
power	622	2.44	9.81	4.02	2.48	1.02
treeadd	245	4.38	13.24	3.02	6.94	1.58
tsp	582	1.99	5.90	2.96	5.93	2.98
voronoi	1129	0.33	1.73	5.24	1.11	3.36
PtrDist						
anagram	650	1.04	16.81	16.16	2.34	2.25
ft	1767	1.09	2.46	2.26	2.45	2.25
bc	7297	0.58	4.08	7.03	2.61	4.50
yacr2	3986	.76	14.72	19.37	0.77	1.01
ks	783	1.86	22.37	12.03	28.70	15.43
SPEC95						
099.go	29246	0.29	2.24	7.72	0.84	2.90
124.m88ksim	19233	0.02	0.36	18.00	0.28	14.00
130.li	7598	0.03	0.39	13.00	0.30	10.00
132.ijpeg	28178	0.22	1.97	8.95	0.54	2.45
SPEC2000						
164.gzip	8616	10.06	84.84	8.43	21.87	2.17
175.vpr	17739	4.31	61.51	14.27	13.89	3.22
181.mcf	2412	6.35	47.76	7.52	40.23	6.34
186.crafty	29650	4.27	84.72	19.84	38.33	8.98
197.parser	11396	2.79	22.67	8.12	13.96	5.00
254.gap	71363	1.90	31.18	16.41	-	-
255.vortex	67220	2.63	58.67	22.30	42.86	16.30
256.bzip2	4647	9.19	161.04	17.52	15.09	1.64
300.twolf	20508	3.58	38.93	10.87	7.67	2.14
SPEC2006						
401.bzip2	8293	2.73	29.60	10.84	27.81	10.19
429.mcf	2685	3.19	20.46	6.41	18.10	5.67
445.gobmk	190119	0.24	1.20	5.00	-	-
456.hmmer	35992	4.42	89.61	20.27	33.26	7.52
458.sjeng	13847	4.46	55.58	12.46	14.20	3.18
464.h264ref	51578	15.71	483.70	30.79	86.40	5.50
462.libquantum	4358	2.24	39.38	17.58	44.27	19.76
471.omnetpp	32500	0.43	6.14	14.28	5.43	12.63
473.astar	5842	9.16	80.07	8.71	33.57	3.66
Application						
sqlite	135761	3.72	71.23	19.15	-	-
median				**9.90**		**3.42**
average				**11.12**		**5.86**

Table 8.1: Performance overhead(Time in sec.)

CHAPTER 9

CONCLUSION

We presented the design of a dynamic type checking tool for C/C++ programs along with a static type inference algorithm to reduce the dynamic tracking overhead. We also presented the implementation details of the tool using the LLVM compiler infrastructure. The tool was tested on numerous benchmarks and we provide an evaluation of the overhead of the tool.

With an overhead of about 4x, our tool is suitable for use as a debugging tool by programmers. It is significantly faster than the tools that exist currently, with the exception of CCured. We plan to analyze the differences more closely in the future and use it to improve our static inference. We also plan to test the tool on larger programs and obtain performance characteristics for those.

We also present a type system for C, which though it is more restrictive than the C standard, ensures type safety of program on execution. The type system is designed to give few false positives on real world C programs to make debugging easier for the programmer. On the other hand, we believe it is strict enough to detect type errors when they occur.

We also present a static analysis that helps reduce the overhead of the tool. Presently, we use it to infer type safety for an object. In the future, we hope to be able to interpret its results to prove type safety for fields inside an object. This, we believe, will help us remove more checks than we currently do and reduce the overhead further. Currently the static analysis does not scale to programs with more thank 50K LOC. We shall work to improve the scalability of the analysis.

We plan to improve the optimization of the instrumentation on the basis of the static analysis results, and remove even more checks. We presently track type information even though the node is type safe. Future work incorporates removing tracking on type safe objects to lower overheads.

We also plan to incorporate it with our previous work that detects bounds

errors and dangling pointer errors [18], to provide comprehensive memory and type safety. We believe this won't present significant challenges as these are all built using the LLVM infrastruture and use the DSA algorithm for optimization.

REFERENCES

[1] J. Seward, "Valgrind, an open-source memory debugger for x86-gnu/linux." [Online]. Available: http://developer.kde.org/~sewardj/

[2] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Winter USENIX*, 1992.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[4] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Int'l Conf. on Softw. Eng.*, Shanghai, China, May 2006, pp. 162–171.

[5] H.-J. Boehm, "Space efficient conservative garbage collection," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/155090.155109 pp. 197–206.

[6] "LLVM," `http://llvm.org`, 2006.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, pp. 13(4):451–490, October 1991.

[8] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps, "Debugging via run-time type checking," *Lecture Notes in Computer Science*, 2001. [Online]. Available: citeseer.ist.psu.edu/loginov01debugging.html

[9] M. Burrows, S. N. Freund, and J. L. Wiener, "Run-time type checking for binary programs," in *In International Conference on Compiler Construction*. Springer, 2003, pp. 90–105.

[10] H. Shen, J. Wang, L. Ping, and K. Sun, "Securing c programs by dynamic type checking," in *Information Security Practice and Experience*, 2006, pp. 343–354.

[11] C. Lattner, A. Lenharth, and V. Adve, "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[12] S. Chandra and T. Reps, "Physical type checking for c," *SIGSOFT Softw. Eng. Notes*, vol. 24, pp. 66–75, September 1999. [Online]. Available: http://doi.acm.org/10.1145/381788.316183

[13] G. C. Necula, J. Condit, M. Harren, S. Mcpeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," in *In ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004.

[14] C. Lattner and V. 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.

[15] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, no. 6, pp. 245–258, 2009.

[16] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[17] C. Lattner et al., "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html.

[18] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006, pp. 144–157.

[19] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 263–277.

[20] M. P. Cline, G. A. Lomow, and M. Girou, *C++ FAQs: Frequently Asked Questions; 2nd ed.* Reading, MA: Addison-Wesley, 1999, section 5.2.

[21] M. C. Carlisle, "Olden: parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, 1996.

[22] W. Ji-min, P. Ling-di, P. Xue-zeng, S. Hai-bin, and Y. Xiao-lang, "Tools to make c programs safe: a deeper study," *Journal of Zhejiang University - Science A*, vol. 6, pp. 63–70, 2005, 10.1007/BF02842479. [Online]. Available: http://dx.doi.org/10.1007/BF02842479