An Improved Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs

by

Robert Andrew Rudd

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2010
[February 2010]

Author ....
Department of Electrical Engineering and Computer Science
January 30, 2010

Certified by ....
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

An Improved Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs

by

Robert Andrew Rudd

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I address the challenges of developing tools which use a mixed-level approach to dynamic binary analysis. The mixed-level approach combines advantages of both source-based and binary-based approaches to dynamic analysis, but comes with the added challenge of dealing with the implementation details of a specific implementation of the target language. This thesis describes the implementation of three existing tools which use the mixed-level approach: Fjalar, a C/C++ dynamic analysis framework, Kvasir, A C/C++ value profiling tool, and Dyncomp, a tool for inferring the abstract types of a C or C++ program.

Additionally, this thesis describes the steps I took in increasing the maintainability and portability of these tools. I investigated and documented platform specific dependencies; I documented the process of merging in upstream changes of Valgrind, the Dynamic Binary Instrumenter Fjalar is built on, to aid Fjalar in keeping in-sync with Valgrind bug-fixes; and I implemented a tool for debugging Dyncomp errors.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

First, I would like to thank my advisor, Michael Ernst, for his constant support and enthusiasm for my work. There were times when things didn't go as well as they should have, but Mike was always there to provide guidance and encouragement.

Thanks to Stephen McCamant for being a fantastic mentor. I am truly grateful for all he's done for me; Whether it was remote debugging sessions from 3000 miles away or simply offering advice on what to do next, Stephen was always there to help. I can't even begin to describe the amount I've learned by simply being able to converse with him on a regular basis.

Finally, I'd like to dedicate this thesis to my family: my parents, Brannock and Joanne, and my sister, Robyn. Without their unending love and support, I would not be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Dynamic analysis is a form of program analysis that relies on data collected during a program's execution. This is in contrast to static analysis, which relies on data collected without actually executing the program. Dynamic analyses are used for a wide variety of purposes, including profiling [4] and optimization [3][18], program understanding [12], and debugging [5].

The first step in building a dynamic analysis is, frequently, instrumenting the target program to output necessary information during execution. The two most common instrumentation techniques are source-based and binary-based.

Source-based instrumentation is typically used when language-level information is required, while binary-based instrumentation is used when machine-level information is required. Guo [11] proposed a new, mixed-level approach to binary-based analysis that increases the language-level information available to tools which use the binary-based technique.

This mixed-level approach is as follows: require that the target program be compiled with some form of debugging information (information used by debuggers to map machine-level constructs to language-level constructs) in the binary, and proceed with a standard binary-based instrumentation. This provides easy access to machine-level information. If language-level information is necessary, use the debugging information to translate machine-level information into a language-level representation. This approach allows a tool to have the benefits of a binary-based instrumentation (ease of dealing with low-level concepts like memory safety), while still being able to access language-level abstractions. Additionally, a tool will not need to deal with much of the language's language-level complexities; this can greatly simplify the implementation of the tool, especially when dealing with large, complex languages like C++. This thesis discusses my work in maintaining tools that use a mixed-level approach to instrumentation.

This mixed-level approach is not without its drawbacks: in addition to any difficulties that come with a binary-based analysis such as having to deal with complexities in the machine's architecture, there is an added burden of having to deal with complexities along the interface between the platform and language: details of the language's implementation for that platform, details of the object file formats and debugging information, how the debugging information actually maps to language-level constructs, calling conventions, etc. Overall, these drawbacks make tools that implement the mixed-level approach sensitive to changes in either the implementations of the language or platform.

In practice, the above drawbacks do not significantly complicate the development of a tool that implements the mixed-level approach as the above information is usually available in the platform's ABIs (Application Binary Interface). They can, however, make porting and maintaining the tool a challenge. Applications that are difficult to port and/or maintain run the risk of becoming outdated. Users will be much less likely to use a tool if it requires specific versions of an operating system, system libraries, or a compilation tool chain,

Using this mixed-level approach, a previous researcher in my group built Fjalar, a general-purpose framework for building C/C++ dynamic analysis tools, along with two tools built on top of it: Kvasir, a tool for tracing value and Dyncomp, a tool for determining the abstract types of programs.

This thesis details my work on improving the portability and maintainability of Fjalar, Kvasir, and Dyncomp This thesis is intended to be aid anyone who wishes to develop a Fjalar tool; modify Dyncomp, Kvasir, or Fjalar's source; gain a better understanding of how the mixed-level approach is implemented.

1.2 Using a mixed-level analysis to dynamically detect invariants in C/C++ programs.

This section introduces a complete system that implements the mixed-level approach to infer program invariants for C and C++ programs. A *program invariant* is a fact about a program at a particular point in that program's execution. Some example invariants for a program could be:

```
cur_day is one of {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"}
```

```
arr_index <= arr_length
```

```
node->next->prev == node
```

Where `cur_day`, `arr_index`, and `node` are program variables. The invariants of a program have been used for many applications including program understanding, generating test cases, refactoring, and automatic error detection/patching. Several dozen papers have described over a dozen distinct applications of dynamic invariant detection; A listing of these papers can be found at `http://groups.csail.mit.edu/pag/daikon/pubs/`.

1.2.1 System overview

Figure 1-1 presents the components of this C/C++ invariant detection system:

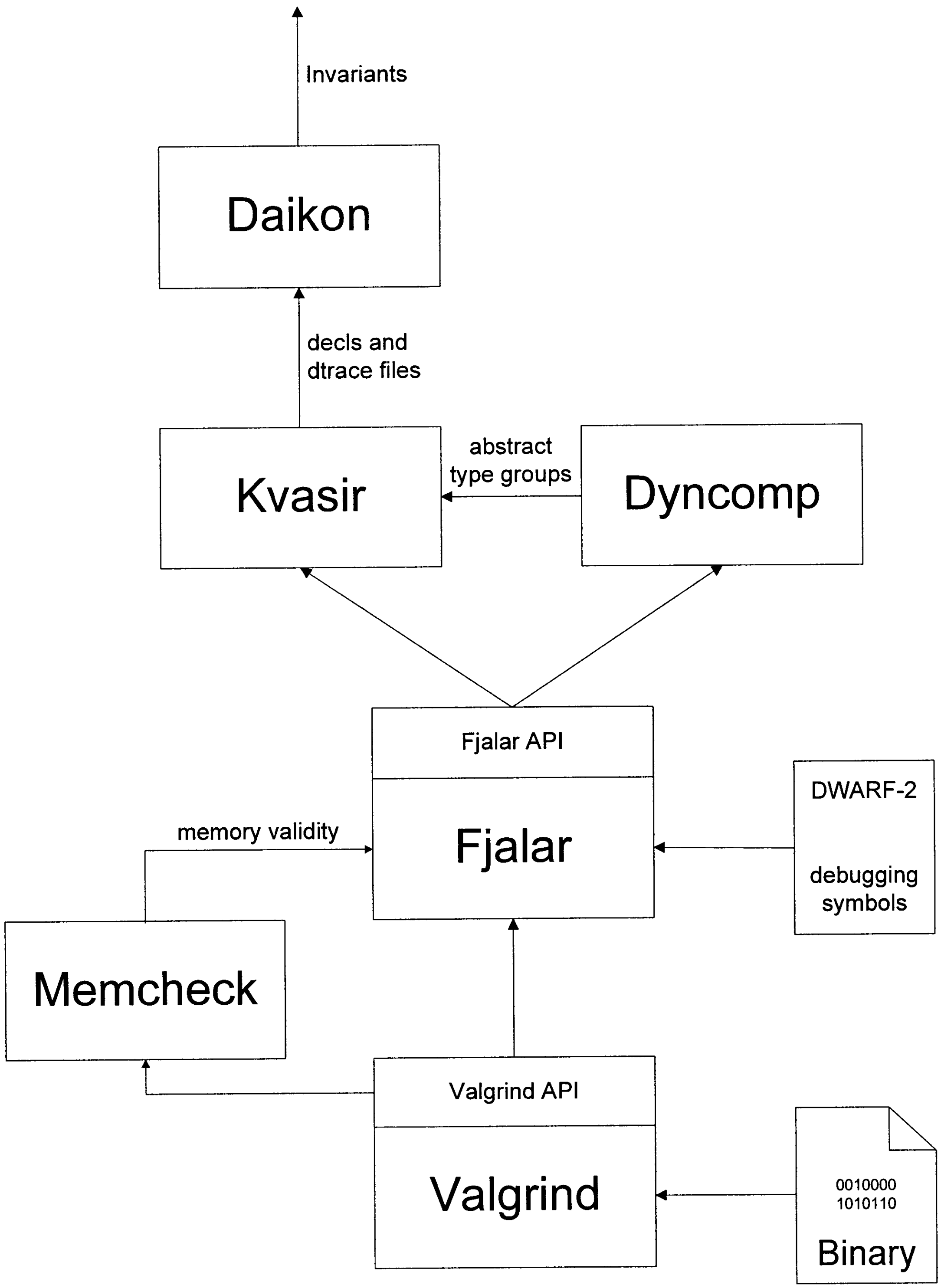


Figure 1-1: Architecture of the C/C++ invariant detector.

- Valgrind, a dynamic binary instrumentation framework, is used for instrumenting and running program binaries. Valgrind is the lowest-level component of this system. Section 1.2.2 gives a brief overview of Valgrind.

- Memcheck, a Valgrind tool for detecting memory errors, is used for determining the validity of memory addresses. Section 1.2.3 gives a brief overview of Memcheck.

- Fjalar, a C/C++ binary analysis framework built as a Valgrind tool, is used for accessing source-level information. Fjalar uses the instrumentation facilities provided by Valgrind and the memory information provided by Memcheck. Section 1.2.4 gives a brief overview of Fjalar.

- Dyncomp, a Fjalar tool for inferring the abstract types of C and C++ programs, is used to increase the quality of the resulting invariants by providing abstract types, which help Daikon better choose over which variables to search for invariants. Section 1.2.5 gives a brief overview of Dyncomp.

- Kvasir, a Fjalar tool for tracing the values of C and C++ programs, is used as the C/C++ front-end for Daikon. Section 1.2.6 gives a brief overview of Kvasir.

- Daikon is used for inferring the invariants of a program. Daikon requires language-neutral declaration and trace files as input, which are provided by Kvasir. Section 1.2.7 gives a brief overview of Daikon.

1.2.2 Valgrind: a framework for heavyweight dynamic binary instrumentation

Valgrind [15] is a framework for developing and running dynamic binary analysis tools. Valgrind uses a dynamic binary-based instrumentation technique: analysis code is added to the original code of the target program at run-time. One key advantage of Valgrind is that it can be used on normally compiled program binaries; there is no need for compiling or linking with a special tool chain.

To perform binary instrumentation, Valgrind uses an approach its developers call Disassemble and Resynthesize (D&R): the binary's machine code is loaded by Valgrind and translated into an Intermediate Representation (IR) known as VEX IR. VEX IR is a non-platform-specific assembly-like representation which follows Single Static Assignment (SSA) rules. VEX IR is RISC-like in nature: each instruction performs at most one operation, all operations operate only on temporaries and literals, and all loads and stores are explicit.

Once the code has been translated into VEX IR, a Valgrind tool can insert its own instrumentation by adding VEX IR statements. After this instrumentation, the IR is translated back into machine code and executed under the supervision of Valgrind.

1.2.3 Memcheck: A Valgrind tool for detecting memory errors

Memcheck is a Valgrind tool for detecting memory errors in C and C++ programs. Memcheck can detect errors related to the accessing of invalid memory, e.g. memory that that has never been allocated or has been freed; or using undefined values, i.e., values that have never been initialized [7].

To achieve this, Memcheck shadows every byte of memory in the target program and keeps track of their validity by observing all memory operations. Nethercote and Seward [14] detail how Memcheck's shadow memory is implemented.

1.2.4 Fjalar: a framework for dynamic analysis of C and C++ programs

Fjalar is a framework that allows developers of dynamic C/C++ analysis tools to combine the dynamic binary instrumentation features of Valgrind with high-level source information. This allows Fjalar tools to perform analyses at the instruction level, but still have access to language constructs, such as variables, types, and functions. Additionally, through the use of Memcheck, Fjalar provides tools with ways to query whether given locations of memory are initialized, allocated, or neither. Fjalar's implementation is detailed in chapter 2

1.2.5 Dyncomp: A Fjalar tool for inferring the abstract types of C/C++ programs.

Dyncomp is a Fjalar tool that infers the abstract types of C and C++ programs. Abstract types are classifications of variables intended to be more fine-grained than declared types. To infer these abstract types, Dyncomp uses the dynamic inference type algorithm proposed by Guo et al. [10]. These finer-grained types are used to control what variables Daikon will try to compare to reduce irrelevant invariants between unrelated types. The abstract type inference algorithm and Dyncomp's implementation of it are detailed in chapter 3.

1.2.6 Kvasir: A Fjalar tool for tracing values in C/C++ programs.

Kvasir is a Fjalar tool that traces values in C and C++ programs. Kvasir is the default Daikon C/C++ front-end. When executed on a target program, Kvasir produces the declarations and traces files required for Daikon to produce invariants. Kvasir is detailed in chapter 4.

1.2.7 Daikon: A tool for dynamically detecting program invariants

Daikon [9] is a tool for dynamically detecting program invariants by performing machine learning over traces of values produced during a program's execution. Daikon is language-independent and relies on external tools, referred to as Daikon front-ends, to provide declaration files and trace files, which are in a language-neutral format.

A Daikon declaration file, or *decls file*, is a text file containing declarations of all program points that Daikon will detect invariants for. Each program point declaration contains variable declarations for all variables relevant to the program point.

A Daikon trace file, or *dtrace file*, is a text file containing value traces for all executed program points in the target program. Each value trace contains the values of all variables that were declared in the program point's declaration.

In addition to Kvasir, the C/C++ front-end, Daikon front-ends have been implemented for a variety of languages including Eiffel, Java, and Perl. Daikon has been used for a variety of tasks including program understanding, automatic test generation [19], and automatic error detection [13][6] and repair [16].

The Daikon distribution, along with manuals for users and developers, can be obtained at `http://groups.csail.mit.edu/pag/daikon/`. Kvasir, Dyncomp, and the Perl and Java front-ends are included in this distribution.

1.3 Outline

This thesis is organized into the following chapters:

- Chapter 2 describes the implementation of the Fjalar C/C++ dynamic analysis framework, its platform-specific dependencies, and the steps I took to increase its maintainability.

- Chapter 3 describes an algorithm for inferring the abstract types of a program, Dyncomp's implementation of this algorithm, and areas where platform and language implementations affect Dyncomp's results.

- Chapter 4 details the implementation of the Kvasir value tracing tool.

- Chapter 5 concludes with a summary of my work and goals for future work.

Chapter 2

Fjalar, a dynamic analysis framework for C and C++ programs

Fjalar [11] allows developers of dynamic analysis tools to combine the dynamic instrumentation capabilities of Valgrind (section 1.2.2) with the rich language-level information within a program binary's debugging information. Fjalar allows tools to work with high-level language constructs, such as the variables, types, and functions during a program's execution. Fjalar currently serves as the basis of two C/C++ tools: Kvasir, a Daikon front end for C and C++ programs; and Dyncomp, a dynamic abstract type inference tool.

This chapter outlines the implementation of Fjalar and is helpful for anyone who wishes to develop Fjalar tools, modify Fjalar's source, or understand how to implement a framework for combining source constructs with dynamic binary analyses.

Fjalar is composed of four primary modules (see figure 2-1):

- The DWARF parser. The DWARF parser runs before a target program is executed and transforms the DWARF debugging information in the program's binary into a more suitable high-level representation. The DWARF parser is described in section 2.1.

- The function instrumenter. The function instrumenter runs along side the program as its executing and uses Valgrind's API to instrument function entries and exits with Fjalar callbacks. The instrumenter is described in section 2.2.

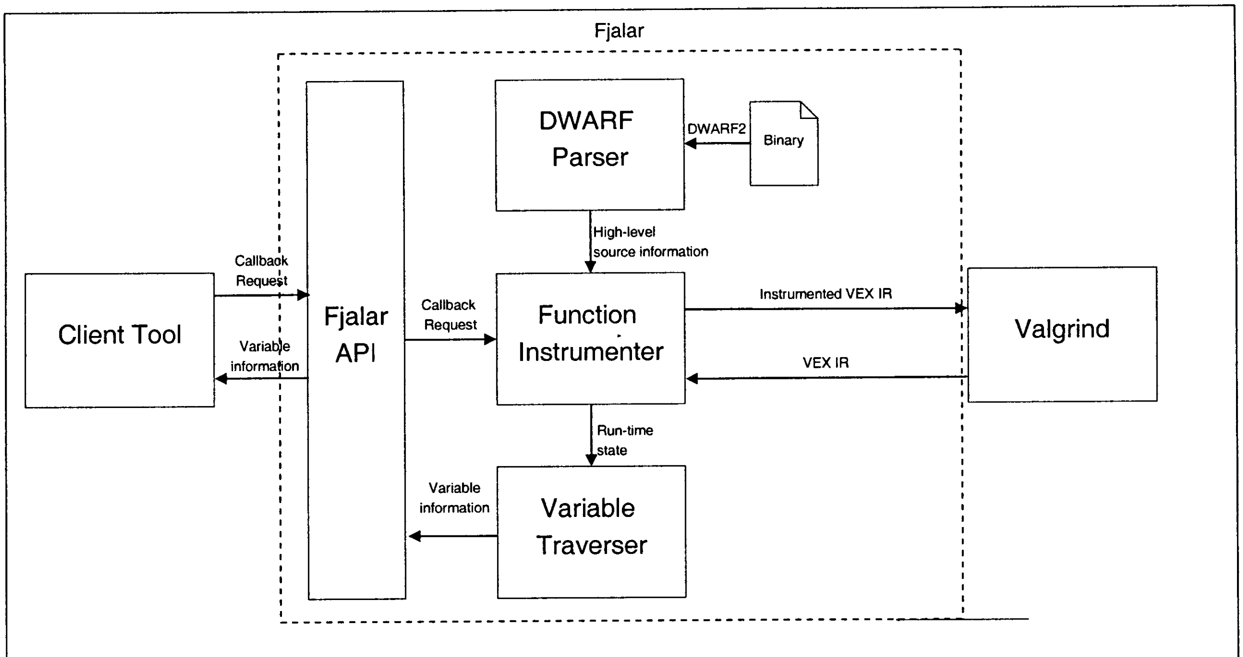


Figure 2-1: Fjalar architecture.A client tool uses Fjalar by issuing a callback request, then receiving callbacks that contain variable information.

- The variable traverser. The variable traverser may be invoked at any time before, after, or during the execution of a target program to provide information, such as the name, type, and size, of a variable. When used during the program's execution, the variable traverser can provide the value for the variable at the time of invocation. The variable traverser is described in section 2.3

- Fjalar's API. The API (application programming interface) specifies a simple interface through which client tools can view variable information and request callbacks be inserted into the target program. Fjalar's API is described in section 2.4

This chapter also describes the portability challenges we faced in creating Fjalar in section 2.5.

2.1 The DWARF Parser

2.1.1 The DWARF debugging format

Debugging information maps between the low-level information contained in a binary and the symbolic, high-level constructs in the source code. This mapping allows debuggers to present higher level information to a programmer than they could otherwise. DWARF is a standardized format for debugging information [17]. There are currently three versions of the DWARF format. DWARF version 1 was developed and released in the mid-1980s, but never achieved widespread use due to inefficiencies in its representation of language constructs causing it to require large amounts of storage [8]. DWARF version 2 was released in 1990 and is currently the standard debugging format for Linux binaries. Fjalar is only designed to work with DWARF version 2. DWARF version 3 was released in 2006 and has yet to gain wide use. Fjalar utilizes the information available in the DWARF format to provide client tools with language-level information for use in their dynamic binary analyses.

2.1.2 Readelf hooks

The ELF binary format is the standard object file format on Linux systems, Fjalar uses `readelf`, a component of GNU Binutils, for reading ELF files and extracting any available DWARF debugging information. Fjalar contains a modified version of `readelf` in which all the functions for displaying DWARF information are replaced by callbacks into Fjalar's `typedata.c`. When parsing has finished, `typedata.c` will contain an array of every DWARF entry in the target binary. This representation, while comprehensive, is flat and preserves little of the source's structure. This makes it difficult to perform the lookups Fjalar needs, such as retrieving all global variables or determining all methods associated with a class, so this representation is translated into a more convenient format as described in section 2.1.3.

2.1.3 Translating DWARF information into Fjalar data structures

The majority of Fjalar's source-level information is represented by three data structures: (1) FunctionEntry, (2) VariableEntry, and (3) TypeEntry. They are declared in fjalar_include.h.

FunctionEntry

The FunctionEntry structure contains information about a particular function. It contains the function's name, the name of the file in which the function is defined, the start and end address of the function's instructions, whether or not the function is declared as file-static, and pointers to the function's formal parameters and local variables. When dealing with C++ programs, a function's FunctionEntry also contains the mangled and demangled name of the function and, if it is a member function, a pointer to the TypeEntry of its enclosing class.

VariableEntry

The VariableEntry structure contains information about a particular variable. It contains the variable's name, location, declared type, and whether it is a global variable, local variable, formal parameter, or return variable. Additionally, if it is a field or member variable it contains a pointer to its enclosing structure's TypeEntry.

TypeEntry

The TypeEntry structure contains information about a particular type. There are predefined TypeEntrys for all the primitive types defined in C and C++, such as char, int, long. A TypeEntry contains the declared type of the variable, the name of the type, and the size of the type in bytes.

If the type corresponds to a struct, class, or union (this is referred to as an aggregate type throughout the Fjalar codebase), it contains a pointer to an AggregateType structure which contains additional information including a member variables and, in the case of C++ classes, constructors, destructors, and superclasses.

Translation algorithm

`generate_fjalar_entries.c` translates the low-level representation of the DWARF debugging information provided by `typedata.c` into the data structures described in section 2.1.3 by making two processing passes over `typedata.c`'s array of DWARF entries.

First it makes a pass for functions. During this pass, Fjalar creates a `FunctionEntry` for every function contained in the debugging information and creates two hash tables for accessing them. One hash table is indexed by the address of the function's first instruction (referred to as the `startPC`) and the other is indexed by the address of the instruction corresponding to the first line of code of the function's body (referred to as the `entryPC`).

Fjalar then makes a second pass over the array of DWARF entries, creating a `VariableEntry` and, if necessary, a `TypeEntry` for every variable. At this point, Fjalar associates formal parameters and local variables with their containing functions and creates a list of all the global variables in the program.

Finally, Fjalar iterates over one of the `FunctionEntry` tables to link any member functions and constructors/destructors with their enclosing class. This entire process takes linear time proportional to the size of the debugging information.

2.2 Instrumenting function entries and exits.

Fjalar instruments the entry and exit of every function in the target program with a callback into Fjalar. This callback collects the necessary information for variable traversal, such as the stage of the target program's stack. Variable traversal is detailed in section 2.3.

To perform this instrumentation, Fjalar is built on top of Valgrind, a framework for dynamic binary instrumentation [15] that works with unmodified program binaries. This avoids the disadvantages [11] of source-based instrumentation or customized compilation tool-chains.

Valgrind uses dynamic binary recompilation: Valgrind first loads a target program into its own process and disassembles the target program's machine code, one small code block at a time, into an intermediate representation (IR) known as the VEX IR. Valgrind then provides its client tool with the VEX IR and an API to instrument the IR with its own VEX instructions. Valgrind then reassembles the instrumented IR block back into machine code and executes it. This process is covered in much greater detail by Nethercote [15].

2.2.1 Determining entries and exits

The DWARF information provides the first and last instructions of a function, however, this is insufficient for accurately detecting entries and exits.

For entries, Fjalar's callback should be executed after the prologue of the function has run, as the variable location provided by the DWARF information tends to only be valid for functions with a valid stack frame. For exits, the last instruction may not even be an exit, as it could be a jump to an instruction earlier in the function. Even if it is an exit, there may still be other exits, as the compiler may emit multiple instructions which exit the function. To deal with the above issues, Fjalar makes use of both the available DWARF information and the VEX IR. To determine function entry, Fjalar needs the `startPC`, the `entryPC` (section 1.3.4), and the VEX basic block which contains the `startPC`. A VEX basic block is a sequence of VEX IR instructions with a single entry and a single exit. Fjalar examines every instruction starting from the beginning of a basic block, asserting it has found an appropriate entry point if:

1. the current instruction is at `entryPC`, or

2. the current instruction is at the end of the basic block.

The consideration of (2) as a valid entry point is motivated by the assumption that the prologue will always be completed by the end of the first basic block of the function. Pseudocode for this algorithm is shown figure 2-2.

```
handle_possible_entry(Instruction* basicBlock) {
    // Only examine the first basic block of a function
    if(!table_contains(functionEntryByStartPC,
                       basicBlock[0])) then
        return

    func := table_lookup(functionEntryByStartPc,
                         basicBlock[0])
    entryPoint := address_of(basicBlock[0])

    for i := length(basicBlock) to 0 do
        instr = basicBlock[i]
        if(address_of(instr) < func.entryPC) then
            insert_call(entryPoint, fjalar_entry)
            return
}
```

Figure 2-2: Pseudocode for handling function entry. This function examines a VEX IR basic block and, if it is the first basic block of a function, inserts a Fjalar callback after the function's prologue. This function is called for every basic block the program executes.

Determining function exits is much simpler due to Valgrind representing all function exits with the VEX IR statement Exit, thus Fjalar can determine function exits by simply tracking Exits.

Non-local exits

Non-local exits are particularly troublesome for Fjalar. A non-local exit in C and C++ is any exit from a function that is not done through a ret instruction. Non-local exits typically show up in C programs through the use of setjmp and longjmp and in C++ programs through throwing exceptions. Non-local exits do not show up as a normal function exit in the assembly and accordingly do not show up in the VEX IR as Exit statements. As Fjalar's only mechanism for detecting function exits is through the Exit statement, there's no instrumentation of non-local exits.

2.3 Variable Traversal

Fjalar provides information about all non-local variables in a program to client tools. This information includes type, name, location, value, and the number of levels of pointer indirection. Due to the memory-unsafe nature of C and C++, safely determining the values of variables in a target program can be difficult as any pointer variable may point to an invalid memory location which could cause a crash if accessed. The handling of variable traversal is the most complex aspect of Fjalar. Variables can be broken into 2 groups: non-pointers and pointers. For both cases the DWARF information provides enough information to determine the variable's location at run-time.

2.3.1 Translating DWARF location information

Locations in the DWARF information are provided as a sequence of DWARF expressions. Each DWARF expression is itself made up of a DWARF operation and one or more DWARF atoms which serve as operands. In addition to the DWARF atoms, the DWARF operations may also perform operations on an implicit stack. The DWARF location information can be thought of as an instruction set for a simple stack machine. Fjalar currently implements a subset of this instruction set. This subset contains the most common DWARF instructions and we have not encountered any DWARF expressions which contains instructions Fjalar does not support. Fjalar's DWARF expression interpreter can be found in `fjalar_traversal.c` and complete information about the DWARF location information is described by Silverstein [17].

2.3.2 Non-pointer variables

For non-pointer variables, Fjalar interprets the sequence of DWARF expressions in the DWARF information to determine the variable's location. Variables in registers are handled by a lookup into Valgrind's register state, which contains the values for the target program's registers.

```
// numargs.c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("Number of arguments: %d\n", argc);
  return 0;
}
```

Figure 2-3: Source for numargs.c.

```
<2><298>: Abbrev Number: 6 (DW_TAG_formal_parameter)
    DW_AT_name         : argc
    DW_AT_decl_file    : 1
    DW_AT_decl_line    : 4
    DW_AT_type         : <24a>
    DW_AT_location     : 2 byte block: 91 6c
                        (DW_OP_fbreg: -20)
```

Figure 2-4: DWARF information entry for argc.

2.3.3 Example non-pointer variable traversal

This section describes the traversal of a non-pointer variable using the DWARF information. Consider the program numargs.c shown in figure 2-3. Figure 2-4 contains the portion of the DWARF information entry that relates to the variable argc.

The DW_AT_location line of the DWARF information entry for argc contains the expressions to be evaluated to determine the location of argc. In this case there is only one, however there can be an arbitrary number. The DW_OP_fbreg:-20 expression entails retrieving the value that is in the frame base register (usually the %ebp/%rbp register for x86/x86-64) and adding -20 to it. When a client requests a traversal of argc, Fjalar executes the DWARF expression, retrieving register values from Valgrind as necessary, and returns the value and address through a callback into the client tool.

2.3.4 Pointers and Arrays

Pointer and array variables require more work on Fjalar's part. The C language standard does not require that pointers point to a valid location. Instead, it leaves the behavior of dereferencing pointers that point to invalid locations, such as NULL or a location that was recently `free`'d, undefined. This makes Fjalar's job of performing variable traversal difficult: Fjalar must avoid dereferencing pointers to invalid locations to prevent crashing both itself and the target program, but must be able to determine whether a pointer is valid or not to provide values to all well-defined variables. This is especially important because Fjalar client tools may attempt to dereference pointers even at times when the target program itself would not do so. Due to the relationship between pointers and arrays in C/C++, specifically implicit conversion of arrays to pointers and pointers to arrays, arrays can exhibit these issues.

Pointers

In addition to the variable information mentioned in section 2.3, Fjalar provides values that may only be reachable through multiple layers of pointer indirection, e.g., for the variable declared as `int **var`, Fjalar provides the values of `var`, `*var`, and `**var`. Special care must be taken when doing this to avoid accessing invalid memory locations. Consider the following C declaration that declares a pointer:

```
int *ptr;
```

Fjalar provides client tools with the addresses and values of the variables `ptr` and `*ptr`. Although `*ptr` is not an actual variable in the source code, I will refer to such pointer dereference constructs as variables; in the Fjalar source, these are referred to as derived variables. Providing the value of `*ptr` is a dangerous operation because Fjalar has no guarantee that the memory location pointed to by `ptr` is valid. To ensure safety when traversing pointers, Fjalar makes use of Memcheck, a Valgrind tool designed to detect memory errors [14]. Before accessing any memory location in the target program, Fjalar queries Memcheck for whether or not the location is allocated and initialized. If Memcheck reports the address as either unallocated or uninitialized Fjalar will not access the location and will inform the tool.

Example traversal of pointer variables

This section describes the traversal of two pointer variables. Consider the program ptrs.c in figure 2-5. Fjalar needs to provide a client tool with the value of goodPtr, badPtr, and *goodPtr, but must not attempt to dereference badPtr to obtain *badPtr as it is a null pointer and dereferencing it will most likely cause the program to crash. To achieve this, Fjalar determines the values of goodPtr and badPtr using the DWARF information (refer to section 2.3.3 for an example of how this is done). Figure 2-6 contains the DWARF information entries for goodPtr and badPtr.

For *goodPtr and *badPtr, Fjalar queries Memcheck with the values obtained from goodPtr and badPtr. Memcheck will return INITIALIZED for goodPtr's value and UNALLOCATED for badPtr's value.

Finally, Fjalar will invoke a client callback for each of the four variables with the variable's VariableEntry, value, address, and array size (Fjalar treats pointers as arrays consisting of one element, section 2.3.4 covers Fjalar's treatment of actual arrays). The callback function signature is roughly:

```
TraversalResult performAction(VariableEntry* var, void* addr, void*
valuePtr, int numElements, VariableState state)
```

The arguments have been simplified for illustrative purposes, the actual signature can be viewed in the example in section 2.4.3. A pointer to the value is provided as opposed to the actual value to allow the tool to control what type the value is interpreted as. For the four variables mentioned before, the values for addr, *(int *)valuePtr (the value pointed to by valuePtr interpreted as an integer), and numElements are in table 2.1.

As table 2.1 shows, Fjalar would be able to provide addresses for all four variables, and correct values for all well-defined variables. For *badPtr, 0 is passed for valuePtr to inform the tool that the pointed-to location is invalid.

```
1.  // ptrs.c
2.
3.  int number = 100;
4.  int *goodPtr = &number;
5.  int *badPtr = NULL;
6.
7.  int main(int argc, char** argv) {
8.    printf("goodPtr: %p\n badPtr: %p\n", goodPtr, badPtr);
9.    return 0;
10. }
```

Figure 2-5: Source for ptrs.c.

```
<1><26a>: Abbrev Number: 6 (DW_TAG_variable)
    DW_AT_name        : goodPtr
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 4
    DW_AT_type        : <284>
    DW_AT_external    : 1
    DW_AT_location    : 9 byte block: 3 60 8 50 0 0 0 0 0
                       (DW_OP_addr: 500860)
<1><28a>: Abbrev Number: 6 (DW_TAG_variable)
    DW_AT_name        : badPtr
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 5
    DW_AT_type        : <284>
    DW_AT_external    : 1
    DW_AT_location    : 9 byte block: 3 70 8 50 0 0 0 0 0
                       (DW_OP_addr: 500870)
```

Figure 2-6: DWARF information entries for goodPtr and badPtr.

Arrays

Arrays compound the difficulties of pointers (section 2.3.4) with the need to determine the number of valid elements. As arrays in C and C++ do not contain size information,

Variable	addr	*(int *)valuePtr	numElements	state
goodPtr	0x500860	0x500858	1	INITIALIZED
*goodPtr	0x500858	100	1	INITIALIZED
badPtr	0x500870	0	1	INITIALIZED
*badPtr	0	N/A	1	UNALLOCATED

Table 2.1: Values passed to client tool callback.

determining the number of valid elements can be difficult. The DWARF information provides the size of statically allocated arrays, however, there is no information available for arrays that are dynamically allocated at run-time, thus determining the size of static arrays is a simple lookup, but a much more involved process (outlined in case 3 of the following algorithm) for dynamic arrays. There is one additional complication: arrays in C and C++ are commonly converted to pointers. Also, all arrays passed as arguments to a function in are implicitly converted to a pointer. Thus, Fjalar must decide whether or not any pointer it observes is actually a pointer to a single element, or an array of multiple elements. Fjalar employs the following algorithm to determine the number of elements at the pointed to address.

1. Determine if the address points to the global address space. If so try to find a global variable in the DWARF information that could correspond to this address. This involves checking if there is a variable with the same address in the program's global address space, or if the address is in the bounds of a global array.

   If a corresponding non-array variable is found, this is a pointer. If this lies between the bounds of a global array in the DWARF information, this is an array of size `array.end - ptr_addr + 1`, where `array.end` is the location of the end of the array as specified in the DWARF information.

2. Determine if the address points to an array currently on the stack. To do this, Fjalar iterates over all functions currently on the stack to try to find one whose stack frame contains the address, Fjalar determines the address range of a function's stack frame by subtracting the stored stack pointer from the stored frame pointer. If it is in some

function's stack frame, it searches all of the function's local variables for a corresponding variable with a search identical to the one in step 1.

3. If the address is not in the global address space or on the stack, linearly scan the memory region after the address using Memcheck until an unallocated address is encountered and return a size of `first_unallocated_addr - ptr_addr`.

   This generally yields good results because Memcheck replaces the target program's dynamic memory allocator with its own; this memory allocator ensures each allocated block is surrounded by a *red zone*, a region of memory marked as both unallocated and uninitialized and the above scan will terminate upon reaching this red zone.

4. If none of the previous searches returned a result, a default size of 1 is returned.

C-like psuedocode for this algorithm is provided in figure 2-7. The algorithm provides correct results for statically allocated arrays. For heap allocated arrays, it will never underestimate the size of an array, but it may overestimate the size of arrays that are inside a larger allocated object as it simply returns the difference between the pointed-to address and the first unallocated address after the pointed-to address.

Example traversals of array variables

This section describes the traversal of three formal parameter variables from the program in figure 2-8: `staticGlobalArray`, `stackArray`, and `dynamicArray`. As the variables are formal parameters, there is no distinction between whether or not they are actually pointers or arrays. For the purpose of this example, we will assume that we are requesting a traversal after the function has been invoked from main and as such, all three parameters point to multiple elements. Additionally, the range `0x500000`–`0x500fff` is assumed to be part of the program's global address space and the range `0x7ffff000`–`0x7fffffff` is assumed to be part of the program's stack address space. To make this example concrete, I will assume the values of the function's parameters are:

```
staticGlobalArray = 0x500960
```

```
determineArraySize(void* ptr_addr, size_t elem_size) {
    // Case 1
    if(isGlobalAddress(ptr_addr)) {
        for array in globalArrayList {
            if((array.begin < ptr_addr) &&
               (array.end   > ptr_addr)) {
                return array.end - ptr_addr + 1;
            }
        }
    // Case 2
    } else if(isStackAddress(ptr_addr)) {
        for array in stackArrayList {
            if((array.begin < ptr_addr) &&
               (array.end   > ptr_addr)) {
                return array.end - ptr_addr + 1;
            }
        }
    // Case 3
    } else if(isHeapAddress(ptr_addr)) {
        int first_unallocated_addr = probeForward(ptr_addr);
        return first_unallocated_addr - ptr_addr;
    }
    // Case 4
    return 1;
}
```

Figure 2-7: Pseudocode for determining the size of an array.

```
stackArray   = 0x7fffffff
dynamicArray      = 0x501000
```

This section is broken down into three subsections:

- The traversal of staticGlobalArray, a statically sized array in the program's global address space.

- The traversal of stackArray, a statically sized array located somewhere in the program's stack.

- The traversal of dynamicArray, a dynamically allocated array located somewhere in

```
// arrays.c
char globalArray[10] = {0,};

void arrayFunction(char* staticGlobalArray,
                   char* stackArray,
                   char* dynamicArray) {
    printf(" staticGlobalArray: %p\n");
    printf(" stackArray: %p\n");
    printf(" dynamicArray: %p\n");
}

int main(int argc, char** argv) {
    int i;
    char stackArray[25] = {0,};
    char *dynamicArray = malloc(25);
    memset(dynamicArray, 0, 25);

    arrayFunction(globalArray, stackArray, dynamicArray);
    return 0;
}
```

Figure 2-8: Source for arrays.c.

the program heap.

Traversing a statically sized global array

To determine the size of staticGlobalArray, Fjalar first checks if the address, 0x500960, is within the program's global address space (all addresses that lie within the .data, .bss, or .rodata sections of the binary as a global address). Once it discovers that it is a global address, Fjalar searches through all global variables for a variable with a matching address and finds globalArray, as 0x500960 is present in globalArray's DWARF information entry. Fjalar then examines globalArray's VariableEntry for the number of elements, which was parsed from the DWARF information by the DWARF parser. Figure 2-9 shows the relevant DWARF information for globalArray. The size of the array is determined from the DW_AT_upper_bound:9 line, which contains the array's upper bound. Fjalar then executes the tool's callback with the address, a pointer to the first element, and an array size of 10.

```
<1><32a>: Abbrev Number: 12 (DW_TAG_variable)
    DW_AT_name        : globalArray
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 2
    DW_AT_type        : <31a>
    DW_AT_external    : 1
    DW_AT_location    : 9 byte block: 3 74 9 50 0 0 0 0 0
                       (DW_OP_addr: 500960)
<1><305>: Abbrev Number: 6 (DW_TAG_base_type)
     DW_AT_name        : long unsigned int
     DW_AT_byte_size   : 8
     DW_AT_encoding    : 7        (unsigned)
<1><31a>: Abbrev Number: 10 (DW_TAG_array_type)
    DW_AT_sibling     : <32a>
    DW_AT_type        : <26a>
<2><323>: Abbrev Number: 11 (DW_TAG_subrange_type)
    DW_AT_type        : <305>
    DW_AT_upper_bound : 9
```

Figure 2-9: Relevant DWARF information entries for globalArray.

Traversing a statically sized stack array

To determine the size of stackArray, Fjalar begins by checking if the address, 0x7fffffff, is in the program's global address space. Once Fjalar determines that this is not a global address, it checks if it is in the program's stack space (all addresses between the first executed function's frame pointer and the stack pointer) and discovers that it is a stack address. Fjalar then searches through all variables associated with any function earlier in the call stack than arrayFunction for a variable that has a matching address. In this case, it finds that stackArray in main has a matching address. From this point, Fjalar proceeds in a similar fashion to the previous section and returns a size of 25 using stackArray's VariableEntry. Figure 2-10 shows the DWARF information entries relevant to stackArray.

Traversing a dynamically allocated array

To determine the size of dynamicArray, Fjalar proceeds as in the previous two sections and checks if the address, 0x501000, is in the program's global or stack address spaces.

```
<2><2c4>: Abbrev Number: 8 (DW_TAG_variable)
    DW_AT_name        : stackArray
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 10
    DW_AT_type        : <2f5>
    DW_AT_location    : 2 byte block: 91 40
                       (DW_OP_fbreg: -64)
<1><2f5>: Abbrev Number: 10 (DW_TAG_array_type)
    DW_AT_sibling     : <305>
    DW_AT_type        : <26a>
<2><2fe>: Abbrev Number: 11 (DW_TAG_subrange_type)
    DW_AT_type        : <305>
    DW_AT_upper_bound : 24
<1><305>: Abbrev Number: 6 (DW_TAG_base_type)
    DW_AT_name        : long unsigned int
    DW_AT_byte_size   : 8
    DW_AT_encoding    : 7        (unsigned)
```

Figure 2-10: Relevant DWARF information entries for stackArray

Once Fjalar determines that it is neither a global address nor stack address, Fjalar performs the steps described in case 3 of the algorithm specified in section 2.3.4 . It begins a linear scan of memory starting at 0x501000, querying Memcheck for the validity of each address. Memcheck will return INITIALIZED for every address between 0x501000 and 0x501024, and UNALLOCATED for 0x501025. As 0x501025 is the first uninitialized block after 0x501000, this is assumed to be the first address that is not part of the array. Fjalar then calculates dynamicArray's size by subtracting the first unallocated address, 0x501025, from the dynamicArray's address, 0x0501000, resulting in a size of 25.

For comparison, the relevant DWARF information entries for dynamicArray are shown in figure 2-11. In constrast to the previous two variables, the entries do not provide information on the bounds of the array.

```
<2><2d9>: Abbrev Number: 9 (DW_TAG_variable)
    DW_AT_name        : dynamicArray
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 11
    DW_AT_type        : <264>
    DW_AT_location    : 2 byte block: 91 68
                       (DW_OP_fbreg: -24)
<1><264>: Abbrev Number: 5 (DW_TAG_pointer_type)
    DW_AT_byte_size   : 8
    DW_AT_type        : <26a>
<1><26a>: Abbrev Number: 6 (DW_TAG_base_type)
    DW_AT_name        : char
    DW_AT_byte_size   : 1
    DW_AT_encoding    : 6         (signed char)
```

Figure 2-11: Relevant DWARF information entries for dynamicArray

2.4 Fjalar API

This section is an overview of the API Fjalar provides to client tools. It illustrates the way a client tool is able make use of the language-level information Fjalar provides during program execution. Fjalar's user interface and client API are covered in much greater detail at the Fjalar website: http://groups.csail.mit.edu/pag/fjalar/

2.4.1 Function instrumentation

This section describes the function instrumentation portion of the Fjalar API. Every tool must implement the following two functions:

```
void fjalar_tool_handle_function_entrance(FunctionExecutionState*);
void fjalar_tool_handle_function_exit(FunctionExecutionState*);
```

Fjalar inserts a call to the first function after the prologue of every function in the target program, and to the second function before every return statement. The FunctionExecutionState will contain the collected dynamic state of the target program at the time of the call, including the address of the top of the stack, the beginning of the current stack frame, and a complete copy of all the values between these two addresses (this is done to allow analysis of formal parameters even in situations where the compiler chooses to reuse their locations on the stack). Additionally it contains a pointer to a FunctionEntry struct (section 2.1.3). Consider the code fragment below:

```
int main(int argc, char **argv) {
  int b = 5;
  int *p = &b, *q;
  return 5;
}
```

Fjalar will instrument the code and insert a call before the first instruction of the initialization `int b = 5`, and one more before the last instruction of the statement `return 5`. The behavior of the above function after instrumentation and translation would be similar to the behavior following fragment:

```
int main(int argc, char **argv) {
  fjalar_tool_handle_function_entry(state);
  int b = 5;
  int *p = &b, *q;
  fjalar_tool_handle_function_exit(state);
  return 5;
}
```

2.4.2 Variable Inspection

Fjalar allows a client tool to inspect all non-local variables in scope at a given function entry or exit. This is primarily done through the following function:

```
void visitVariableGroup(VariableOrigin origin,
                        FunctionEntry* funcPtr,
                        Bool isEnter,
                    Addr stackBase,
                    Addr stackBaseGuest,
                    TraversalAction* callback);
```

The tool provides a VariableOrigin, an enum describing what variable group to traverse (global variables, formal parameters, or the return value), a FunctionEntry corresponding to the current function, whether or not the program is currently paused at a function entry or exit, a pointer to the base of Fjalar's copy of the function's stack frame (section 2.4.1), and a pointer to the base of the function's stack frame. Fjalar uses these parameters to iterate through the list of variables associated with funcPtr and call callback for every variable that has a VariableOrigin of origin. Fjalar passes a variable's VariableEntry and auxiliary information, such as its location in memory, whether or not it is a sequence of elements, and the number of elements to callback.

In addition to providing a function for visiting entire groups of variables, Fjalar provides convenience functions to examine an individual variable given only a VariableEntry, however the information provided is much more limited as it is unable to determine information like memory location and number of elements without the run-time information provided by a FunctionExecutionState.

2.4.3 Example Tool

Figure 2-12 contains an example client tool that makes use of the Fjalar API to print the memory address of all variables at every function entry. Some trivial functions (such as allocators and command line handlers) are omitted for brevity.

2.5 Portability challenges

This section outlines the challenges we encountered while trying to keep Fjalar portable across different versions of Linux kernels and system libraries, while performing the low-level

```
// This simple callback function prints the memory address of a
// program variable described by var
TraversalResult basicAction(VariableEntry* var,
                              char* varName,
                              VariableOrigin varOrigin,
                              UInt numDereferences,
                              UInt layersBeforeBase,
                              Bool overrideIsInit,
                              DisambigOverride disambigOverride,
                              Bool isSequence,
                              // pValue only valid if
                              // isSequence is false
                              Addr pValue,
                              Addr pValueGuest,
                              // pValueArray and numElts
                              // only valid if
                              // isSequence is true
                              Addr* pValueArray,
                              Addr* pValueArrayGuest,
                              UInt numElts,
                              FunctionEntry* varFuncInfo,
                              Bool isEnter) {
    if (isSequence) {
        VG_(printf)("%s (%p) - %d elements\n" ,varName, pValueArray,  numElts);
    } else {
        VG_(printf)(%s (%p)\n", varName, pValueArray");
    }
    // We want to dereference more pointers so that we can find out array
    // size for derived variables:
    return DEREF_MORE_POINTERS;
}

// These functions are called during every instance of a function
// entrance and exit, respectively:
void fjalar_tool_handle_function_entrance(FunctionExecutionState* f_state) {
    VG_(printf)("[%s - ENTER]\n", f_state->func->fjalar_name);

    // visitVariableGroup calls the function pointed to by the final
    // argument for every variable in the chosen group. To correctly
    // determine information on formal parameters, it must be passed
    // the current stack pointer and a copy of the stack.     VG_(printf)("Global variables:\n");
    visitVariableGroup(GLOBAL_VAR, 0, 1, 0, 0 ,&basicAction);
    VG_(printf)("  Function formal parameters:\n");
    visitVariableGroup(FUNCTION_FORMAL_PARAM,
                       f\_state->func,
                       1,
                       (Addr)f_state->virtualStack
                       + f_state->virtualStackFPOffset,
                       f_state->FP,
                       &basicAction);
}

void fjalar_tool_handle_function_exit(FunctionExecutionState* f_state)
{} // Do nothing for exits
```

Figure 2-12: An example Fjalar client tool.

analysis described in the previous sections. This section will cover the situations where Fjalar is dependent on the system ABI and steps the we've taken in keeping Fjalar maintainable.

2.5.1 Use of ABIs

Fjalar depends on information directly present in the DWARF debugging information for most of its functionality. As the DWARF debugging format is standardized across platforms, depending on it doesn't tie Fjalar to a particular platform. Unfortunately, there are a few situations where the DWARF information is inadequate for the facilities we needed Fjalar to provide. There are currently two situations where Fjalar must make use of ABI information: determination of return values and extent of the stack frame.

Fjalar provides client tools with a `VariableEntry` representing the return value for every non-void function in the target program. Unfortunately, the debugging information doesn't contain entries for return values. So Fjalar needs to determine the type and location of return values itself.

To determine the type of a return value, Fjalar uses the function signature, which is available in the debugging information. To determine the location, Fjalar implements a portion of the host system's C ABI, specifically the portion which dictates where a given value will be stored on return. This requires some platform specific code as the return conventions are different for both of Fjalar's supported platforms, x86 and x86-64. This also creates an assumption that all function calls for the created binary will follow the system's standard convention for returning values.

Fjalar also depends on ABI information for determining the extent of the stack frame. The stack frame of a function is completely determined by the position of the stack pointer as well as the system's calling convention. Fjalar assumes the current stack frame consists of the region of memory from the bottom of the red zone, to the DWARF-specified stack frame base. Where the *red zone* (which is distinct from the Memcheck red zone mentioned in section 2.3.4) is an ABI-specified scratch area below the stack pointer, typically used by signal handlers.

2.5.2 Maintenance

One of the most important goals of my additional work on Fjalar was easing the task of maintenance. One problem that comes along with building a tool that works at the compiled binary-level is that it can be incredibly sensitive to changes in the operating system or system libraries. We mitigated this as much as possible by depending on Valgrind or the standardized DWARF information for most portions of Fjalar's implementation. However, due to the need for substantial modifications to Valgrind's and Memcheck's source code that are too specific to be useful upstream, we are unable to trivially substitute new versions of Valgrind within Fjalar. This is problematic as Fjalar's support for an operating system and system libraries is dependent on Valgrind.

One thing learned from this situation was the importance of doing frequent merges of updates from the Valgrind upstream repository into the copy of Valgrind in Fjalar's repository. After two years of tracking Valgrind changes, Fjalar had no dedicated maintainer, and as such, no one performed the needed Valgrind merge. This was detrimental for two reasons: we were missing out on improvements made to both Valgrind and Memcheck and we were unable to run Fjalar on newer systems. Due to Valgrind intercepting many standard library calls and system calls, Valgrind sometimes needs to contain explicit support for kernels and important system libraries such as the C standard library.

When I joined the lab, Fjalar did not work with the newest versions of the GNU C library. This made it difficult for interested users using very new versions of Linux to try out Fjalar on Linux systems. Their only option was to install an older OS to use Fjalar. We understood that for Fjalar to be of use to people, it must avoid requiring specific operating system or library versions. To solve this problem we undertook two steps: spend the time to get Fjalar properly merged with Valgrind and have a plan for future merging.

The first part was very time consuming and offered motivation for the coming up with a solid plan for tracking Valgrind changes more closely. The version of Valgrind in Fjalar at the time was two years old and Memcheck was almost three years old. The merging effort took a period of about three months due to extensive changes in both tools requiring a substantial rewrite of many of the portions of Fjalar which interacted directly with Valgrind and Memcheck, as well as reimplementing our modifications to Valgrind and Memcheck. It was clear that it was undesirable to have to pay such a high time cost again. We decided to draft a plan to more closely track changes to Valgrind. To do this we set a hard schedule of one complete merge per month, performing partial merges as deemed necessary. The end result was a dramatic reduction in the complexity of the merges as a much smaller portion of Fjalar and Valgrind's source had to be modified at a given time.

Furthermore, we decided that the process would have to be documented fully in such a way that the merge can be done by a person unfamiliar with the Fjalar source code. This was to cope with a situation where Fjalar has no primary maintainer. This documentation provides step-by-step instructions on how the merge should proceed, an in-depth and comprehensive explanation of each part of Fjalar that is directly dependent on Valgrind, and all of our modifications to Valgrind and Memcheck. We found that employing this strategy was a tremendous boon in keeping Fjalar up to date. Fjalar now currently supports all modern Linux operating systems, and is routinely merged by people unfamiliar with the Fjalar source code every month.

Chapter 3

Dyncomp, a dynamic abstract type inference tool for C and C++ programs

Programmers usually have a notion of the *abstract types* of variables within their programs. However, even in languages requiring type declarations, the declared types rarely capture all of the abstract types. For example, variables declared as `int` may be used to hold a variety of unrelated quantities, such as counts, indexes, or times. Dyncomp [11] dynamically infers a program's abstract type groups, groups of variables which have the same abstract type.

Consider the program `flightCost.c` shown in figure 3-1. This program computes the cost of a flight with a base cost of $300 that has an extra surcharge of $50 if the weight of the traveler's luggage is greater than 50 pounds. All of the variables are of type `int`, but there are really two abstract types: *weight* (`bag1Weight`, `bag2Weight`, `totalWeight`, and `maxWeight`) and *money* (`weightSurcharge`, `airfareCost`, and `totalCost`). Dyncomp is able to determine these abstract types by observing the program's operations, though it will obviously not be able to name the types. The output of Dyncomp after an execution of `flightCost.c` is shown in figure 3-2; each line of variables represents an abstract type group.

```
//flightCost.c
#include <string.h>
#include <stdio.h>
#include <time.h>

int bag1Weight = 25;
int bag2Weight = 35;
int maxWeight = 55;
int totalWeight;

int weightSurcharge = 50;
int airfareCost = 300;
int totalCost;


int main() {
  totalCost = airfareCost;
  totalWeight = bag1Weight + bag2Weight;
  if(totalWeight > maxWeight) {
    totalCost += weightSurcharge;
  }

  printf(\"The total cost is: %d\n\", totalCost);
  return 0;
}
```

Figure 3-1: Source for flightCost.c.

```
..main()::EXIT0
::bag1Weight ::bag2Weight ::maxWeight ::totalWeight
::airfareCost ::weightSurcharge ::totalCost
```

Figure 3-2: Dyncomp results for the execution of flightCost.c. Each line containing variables represents a distinct abstract type group, i.e., All variables on the same line have the same abstract type.

Once these abstract type groups are discovered they can serve as a tool for program understanding by explicitly displaying the inferred abstract types and possibly lead a programmer to discover errors such as abstraction violations. Additionally these abstract type groups can be used as inputs to other program analyses, such as program slicing, error detection, or detection of program invariants (see chapter 4).

- Section 4.1 describes the abstract type inference algorithm.
- Section 4.2 describes Dyncomp's implementation of the algorithm.
- Section 4.3 describes the limitations of Dyncomp's implementation with respect to machine architecture and compilers.
- Section 4.4 describes the challenges of maintaining and debugging Dyncomp. It also includes the implementation of a tool to make debugging and portability work easier.

3.1 Algorithm

Dyncomp uses a unification-based algorithm for partitioning values and variables by their abstract type into disjoint sets called abstract type groups [10].

Dyncomp works on the assumption that the operations within a program encode the programmer's knowledge of abstract types: certain operations that operate on multiple values imply that the values are of the same abstract type. These operations on values will be referred to as *interactions* between the values. In the previous example (figure 3-1) the statement `bag1Weight + bag2Weight` indicates that the values of `bag1Weight` and `bag2Weight` have the same abstract type. Dyncomp attempts to decode the knowledge implicit within these interactions by tracking all of the interactions that occur during execution. Abstract types are transitive, thus for the variables `a`, `b`, and `c`, if `a` interacts with `b` and `b` interacts with `c`, `a` and `c` have the same abstract type. These interactions are tracked through the use of a union-find data structure: whenever an interaction is observed, the value's abstract type groups are unioned.

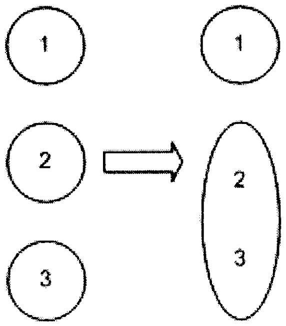
```
1.   int a = 1;
2.   int b = 2;
3.   int c = 3;
4.   b = b + c;
5.   b = a;
```

Figure 3-3: Pseudocode for handling function entry. This function examines a VEX IR basic block and, if it is the first basic block of a function, inserts a Fjalar callback after the function's prologue. This function is called for every basic block the program executes.

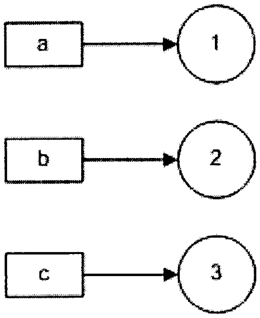
Dyncomp can also infer the abstract type groups for variables. In the case of variables, two variables are members of the same abstract type group if any of the values that the first variable has contained is a member of the abstract type group of any of the values that the second variable has contained. Initially, all variables are assumed to have distinct abstract types.

The remainder of this section is used to demonstrate this algorithm on the C snippet in figure 3-3. In the following images, circles and ellipses represent value abstract type groups, while rectangles represent variable abstract type groups. A solid arrow between a rectangle and a circle implies that one of the members of the rectangle's abstract type group has contained one of the members of the circle's abstract type group. A hollow arrow represents an abstract type group merging.

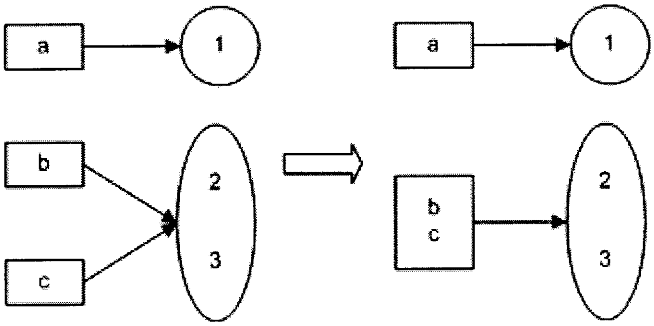
This C snippet contains three distinct values and three distinct variables. By the end of execution 1 is in its own abstract type group while 2 and 3 are together in a second group. This is due to the interaction between 2 and 3 on line 4. The merging that occurs at line 4 is illustrated below:



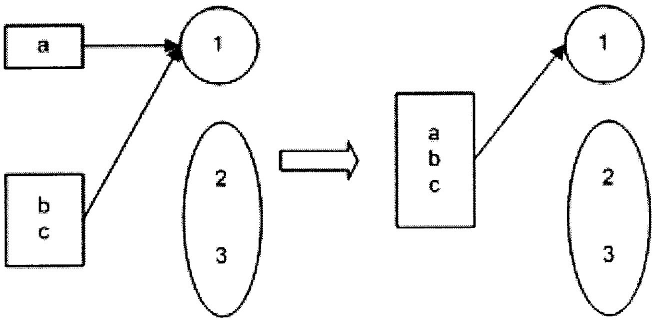
The variable abstract type groups go through a more involved process. After the execution of line 3 all three variables are members of disjoint abstract type groups:



After the execution of line 4, 2 and 3 are now members of the same abstract type group as mentioned above. Because both b and c contain values that are members of the same abstract type group, their abstract type groups are merged:



Finally, after the execution of line 5, a now contains a value that is a member of the abstract type groups of b and c and a's group is merged in:



Thus, at the end of execution of the snippet in figure 3-3, all three variables are members of the same abstract type group and are considered to have the same abstract type.

3.2 Implementation

Dyncomp implements the algorithm in section 3.1 as a dynamic binary analysis. Dyncomp considers every byte of allocated memory as a possible storage location for a value. For every allocated byte of memory, Dyncomp maintains a 32-bit identifier tag for the stored value. This tag represents the abstract type group of the value. To store these tags, Dyncomp uses a sparse array similar to a multi-level page table; this is modeled after Memcheck, which uses a similar data structure for tracking validity information for memory addresses. Additionally, an identifier tag is maintained for every architectural register.

Defining interactions

To provider finer-grained control over the abstract type groups produced by Dyncomp, Dyncomp is parametrized as to what operations are considered interactions (section 3.1). Users can choose between the following interaction definitions:

Dataflow No operations are interactions. Thus, every value belongs to a singleton set representing its abstract type. Variable abstract type groups will only be merged if a single value flowed through multiple variables.

Dataflow and Comparisons Comparison operations (e.g., < and ==) are considered interactions. Thus whenever two values are compared, their abstract type groups are merged.

Units Addition, subtraction, and comparison are considered interactions. This mode is modeled after scientific rules which dictate that values may only be added, subtracted, or compared if they have the same units.

Arithmetic All arithmetic operations are considered interactions. This mode is the most inclusive and yields fewer abstract type groups. This is Dyncomp's default mode as we believe it is easier for users to split up sets that are too large, than to combine sets that Dyncomp did not infer.

3.2.1 Tracking dataflow

On start up, Dyncomp creates a new tag for every allocated byte of memory and every register. After this point, new tags will only be created when a literal is encountered in the VEX IR or a block of memory is allocated. In the case of a literal, a single tag will be created; in the case of a memory allocation, a new tag will be created for every byte allocated.

To track the flow of tags between various memory and register locations, Dyncomp makes direct use of Valgrind's instrumentation functionality. Every VEX IR instruction which causes a movement of values is intercepted by Dyncomp, and a shadow version of the instruction is inserted after. This *shadow* version is an instruction that causes a propagation of tags that mimics the dataflow of the original instruction.

3.2.2 Tracking interactions

To track interactions, Dyncomp uses a global union find structure, which is used for merging sets of tags and determining if two tags are from the same set. Specifically, this union find structure maps a given tag to some canonical tag, referred to as a tag leader. All tags that have been merged will return the same tag leader when looked up in the union find structure, thus the tag leader is an identifier for a set of merged tags. Dyncomp instruments all VEX IR instructions which are considered interactions, as specified by input parameters (section 3.2). After any IR corresponding to an interaction that determines the current tags for the operands to the instruction, Dyncomp inserts a call and merges them in the global union find structure.

3.2.3 Determining Variable Abstract Type Groups

Tracking interactions is sufficient for determining the value abstract type groups, but more work needs to be performed to determine the variable abstract groups. Dyncomp determines the variable abstract type groups from the inferred abstract type groups on a per-function basis.

To determine abstract type groups for variables, Dyncomp maintains a per-function union find structure which is responsible for unioning abstract type groups for all the values that a given variable has every contained. In addition to the value interactions tracked above, Dyncomp uses Fjalar to instrument every function entrance and exit with a call to a variable observation procedure. This procedure is responsible for updating the per-function union find structures to reflect the most recently observed values in the program. Pseudocode for this variable observation procedure is provided in figure 3-4.

The variable observation procedure performs two primary actions:

Propagation of interactions This corresponds to lines 1–4 of figure 3-4. This action ensures any changes in the value abstract type groups of previously held values are reflected in the variable's abstract type groups.

Observation of new values This corresponds to lines 5–8. This action ensures a variable's abstract type group is, at minimum, the union of the abstract type groups of all values the variable has held at a function entry or exit.

At the end of execution, Dyncomp invokes a procedure that performs a final propagation of interactions and uses the per-function union-find structure to determine the program's variable abstract type groups; all variables that have the same final tag leader are members of the same abstract type group. For simplicity, Dyncomp converts these final tag leaders, which are typically large integers, into a comparability number, which is a natural number between 0 and n, where n is the number of distinct abstract type groups. Pseudocode for this procedure is shown in figure 3-5.

3.2.4 An example execution of Dyncomp

To further illustrate Dyncomp's implementation of the abstract type inference algorithm (section 3.1), this section contains an example execution of Dyncomp on the program contained in figure 3-6. I will assume this program is being run on a generic x86 machine and

```
process_variables(FunctionEntry* curFunc) {
 for each variable v in curFunc {
  // value_uf is the global union find structure
  // which is used to track interactions.
  // var_tags contains the leaders observed during
  // the previous invocation of this loop.
  Tag leader = value_uf.find(curFunc.var_tags[v])

  // if var_tags[v] is no longer the leader of its
  // set, then its set has been unioned since the last invocation
  if (leader != var_tags[v]) {
    // Propagate unioning from the global union find structure
    // to the per-function union-find, curFunc.var_uf, and update
    // var_tags[v] with the new leader.
    curFunc.var_uf.union(leader, curFunc.var_tags[v])
  }

  // Retrieve the tag corresponding to the current value of
  // v.
  Tag new_tag = get_tag(addressOf(v));

  // If needed, create new set in the per-function union-find.
  if(!curFunc.var_uf.contains(new_tag)) {
    curFunc.var_uf.make_singleton(new_tag);
  }

  // Union the recently retrieved tag with the existing tags in
  // the per-function union-find.
  curFunc.var_uf.union(new_tag, curFunc.var_tags[v]);

  // Update var_tags[v] with the current leader
  curFunc.var_tags[v] = curFunc.var_uf.find(var_tags[v]);
  }
 }
```

Figure 3-4: Pseudocode for the variable observation procedure which runs at every function entry and exit. This procedure propagates changes in value abstract type groups to variable abstract type groups.

```
get_comp_number(FunctionEntry* curFunc) {
  for each variable v curFunc:
    Tag leader = value_uf.find(var_tags[v])
      if (leader != var_tags[v]) {
        curFunc.var_uf.union(leader, var_tags[v])
     }

    Tag final_tag = var_uf.find(leader);
      // Retrieve the comparability number for
      // this tag, creating one if necessary.
     if (!comp_num_map.contains(final_tag)) {
        comp_num_map.put(final_tag, g_cur_comp_num++);6.
      }
    return comp_num_map.get(final_tag);
  }
```

Figure 3-5: Pseudocode for the procedure that performs the final propagation and outputs a priority number. This procedure is run after the target program has finished executing.

that the generated code follows System V ABI calling conventions. To make this example concrete, I will also assume that the state of the machine's stack pointer after the executions of the main's prologue is 0xFFF0.

First, Dyncomp loads the program's binary and initializes its data structures. Valgrind takes over and begins executing the program's instructions, inserting instrumentation as necessary (see section 1.2.2 for an overview of Valgrind and section 2.2 for details on Fjalar's function instrumenter).

Next, main's body is entered and its prologue is executed. After the execution of main's prologue, Dyncomp's function entry callback is executed. Both Dyncomp's function entry callback and function exit callback (section 2.4.1) perform the algorithm shown in figure 3-4. At this point in execution, Dyncomp performs no actions as main has no formal parameters nor are there global variables. After Dyncomp's entry callback returns, term1 and term2 are initialized with with the values 5 and 7. The state of the program's stack at line 10 is shown in figure 3-7: term1's value is stored at address 0xFFEC and term2's location is

```
1.  //sum.c
2.
3.  void printNums(int a, int b) {
4.    printf(\"a: %d, b: %d\n'', a, b);
5.  }
6.
7.  int main() {
8.    int term1 = 5;
9.    int term2 = 7;
10.   printNums(term1, term2);
11.   term1 += term2;
12.   return term1;
13. }
```

Figure 3-6: Source for sum.c

stored at address 0xFFF0. As Dyncomp tracks the movement of values, shadowed versions (section 3.2.1) of these movements are executed, causing the tag mappings for 0xFFEC and 0xFFF0 to be updated. As the values that were stored into these addresses were literals in the program's source, two new tags, 1 and 2, are created for both addresses.

Figure 3-8 shows the contents of Dyncomp's primary data structures at line 10: g_varUf is the global union-find structure used for tracking interactions (see section 3.2.2), tagMap is a map between addresses and tags for tracking dataflow, and printNumsEntry is the FunctionEntry associated with printNums. printNumsEntry contains varUf, the per-function union-find for propagating value interactions to value abstract type groups and varTags, an array of the last observed tags for in-scope variables.

Next, the program calls printNums with term1 and term2 as parameters. The values for term1 and term2 are copied to the top of the stack and a jump is made to printNum's body. Figure 3-9 shows the contents of the same portion of the stack from figure 3-7. printNum's prologue is executed, followed by Dyncomp's entry callback. This time, there is work for Dyncomp to do: Dyncomp propagates the tags corresponding to the values of the function's arguments. This causes an insertion of the tags 1 and 2 into printNumEntry.varUf, and 1 and 2 to be associated with a and b in printNumEntry.varTags. Dyncomp then propagates any new unions in g_valUf to printNumEntry.varUf (lines 18-29 of figure 3-4) but there

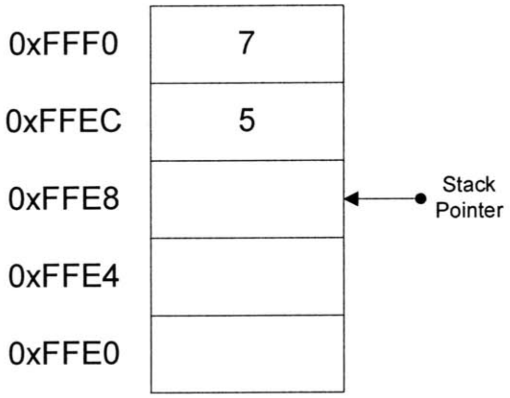
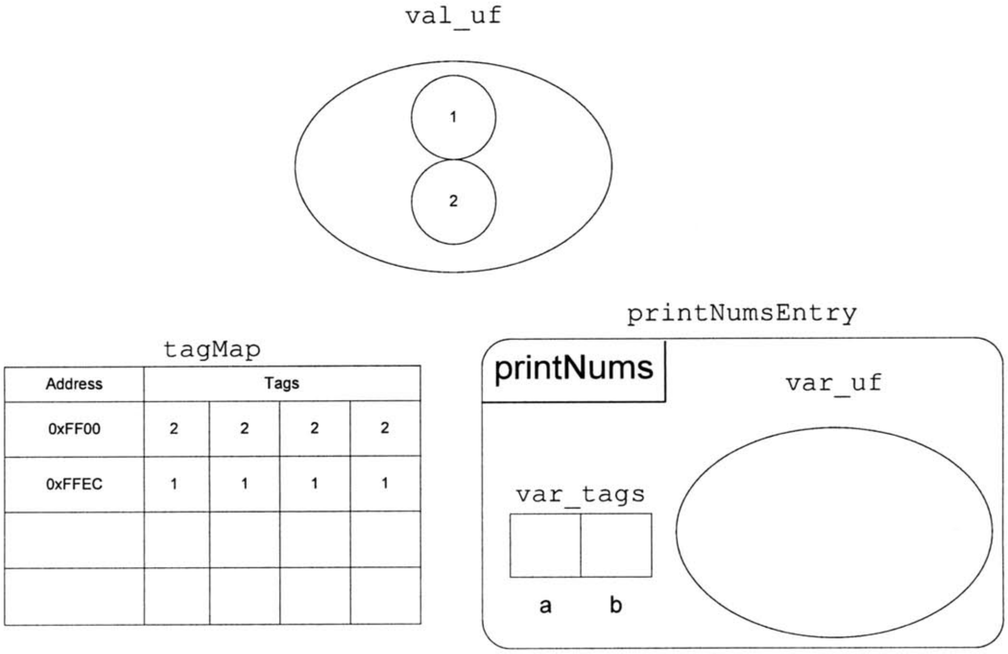


Figure 3-7: The contents of the top of `sum.c`'s at line 10. 0xFFF0 contains the value of term2, and 0xFFEC contains the value of term1.



tagMap

Address	Tags			
0xFF00	2	2	2	2
0xFFEC	1	1	1	1

Figure 3-8: The contents of Dyncomp's primary data structures at line 10, tagMap currently contains tags for the recently initialized 0xFFEC and 0xFFE8; these tags are newly created due to 0xFFEC and 0xFFE8 being loaded with values that occur as literals in the source (see lines 8 and 9 of sum.c.)

are none in this case. Figure 3-10 shows the contents of Dyncomp's data structures after the callback for `printNum` has returned. After that callback, the `printf` is called, but for simplicity, I will assume it has no effect. Finally, `printNum` returns and Dyncomp's exit callback is executed. Nothing will be updated as no interactions have occurred since the entry callback.

After `printNum` returns, the program adds the value of term1 and the value of term2, storing the result in term2. The state of the program's stack is shown in figure 3-11. Dyncomp will track this interaction by unioning the tags corresponding to term1's address and term2's address in the global value union-find. The storage of the result in `term2` will be tracked by Dyncomp, causing an update to `g_tagMap`. `0xFFF0` will now be associated with tag `1`, the tag leader of the newly unioned tag set. The state of Dyncomp's data structures after this operation is shown in figure 3-12.

Finally, `main` returns and the program's execution ends. Control is then passed back to Dyncomp, and Dyncomp performs the final propagation algorithm shown in figure 3-5. During this process, the interaction on line 11 that occurred after `printNum` is reflected in `printNumEntry.varUf` and a comparability number is generated for `a` and `b`. As `a` and `b` have the same final tag, they receive the same comparability number and thus, they belong to the same abstract type. The final state of Dyncomp's data structures is shown in figure 3-13.

3.3 Architectural Limitations

A choice was made early in Dyncomp's development to track only one tag per register; the rationale was that only one value should be present in a register at a time. This assumption held for the most part for x86, but was incorrect for x86-64 (AMD64, EMT64/Intel 64). The System V x86 ABI [2] requires that arguments to a function be passed via the call stack, while the System V AMD64 ABI [1] mandates that the first few arguments (the actual number depends on their size) be passed via register and that all structs less than 8 bytes be passed via a single register. This new calling convention creates a problem when a single struct

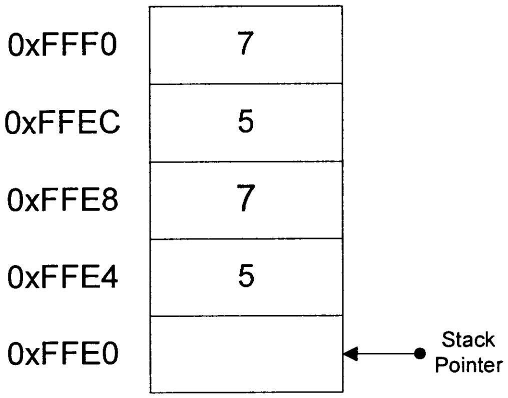
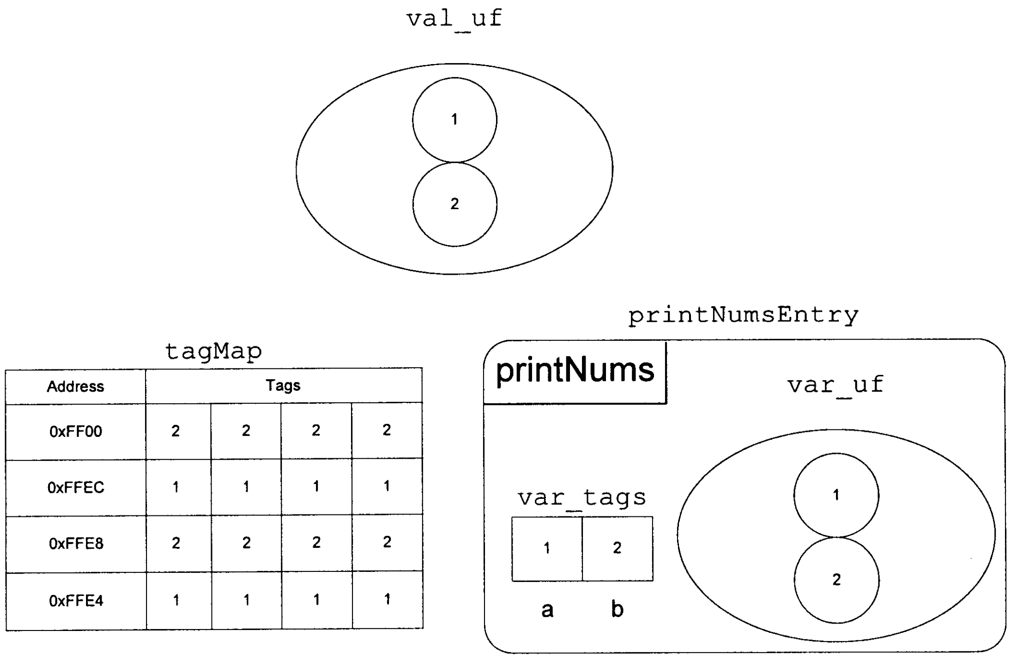


Figure 3-9: The contents of the same portion of the stack shown in figure 3-7 at line 4. The values for printNum's arguments have since been pushed on the stack. 0xFFE8 contains the value of b, and 0xFFE4 contains the value of a.



tagMap

Address	Tags			
0xFF00	2	2	2	2
0xFFEC	1	1	1	1
0xFFE8	2	2	2	2
0xFFE4	1	1	1	1

Figure 3-10: The contents of Dyncomp's data structures at line 4. The movement of values on the program's stack has been reflected in g_tagMap.

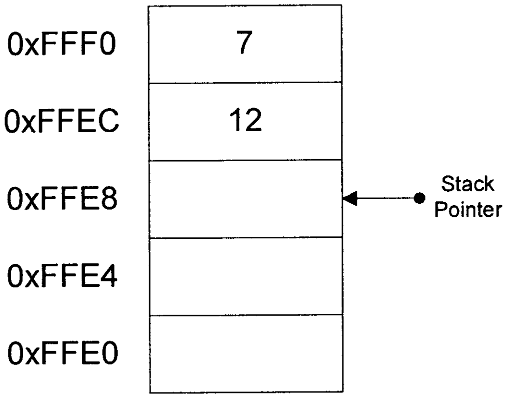
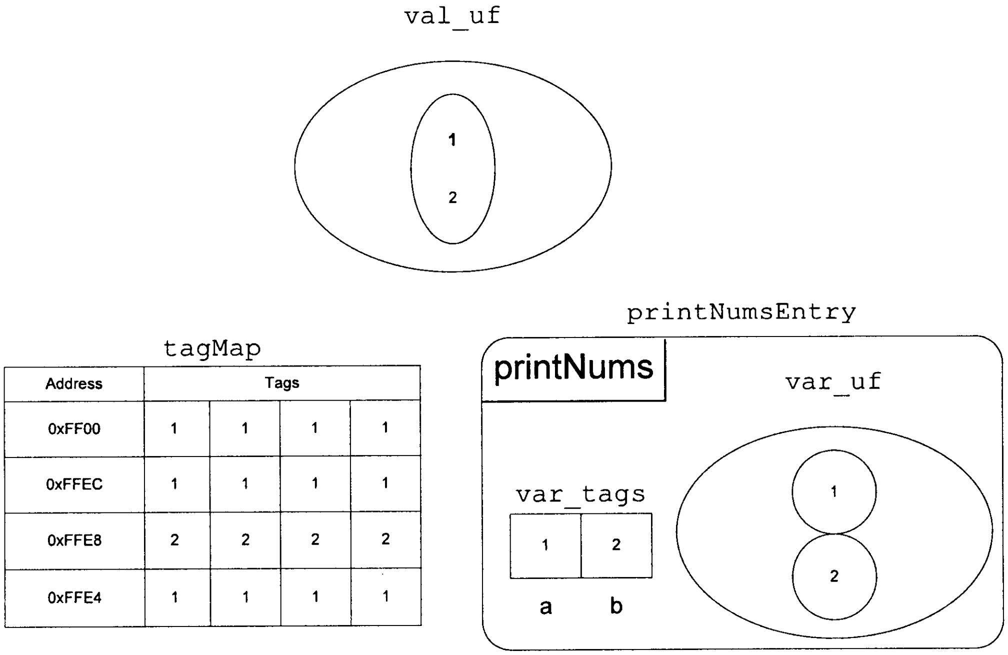


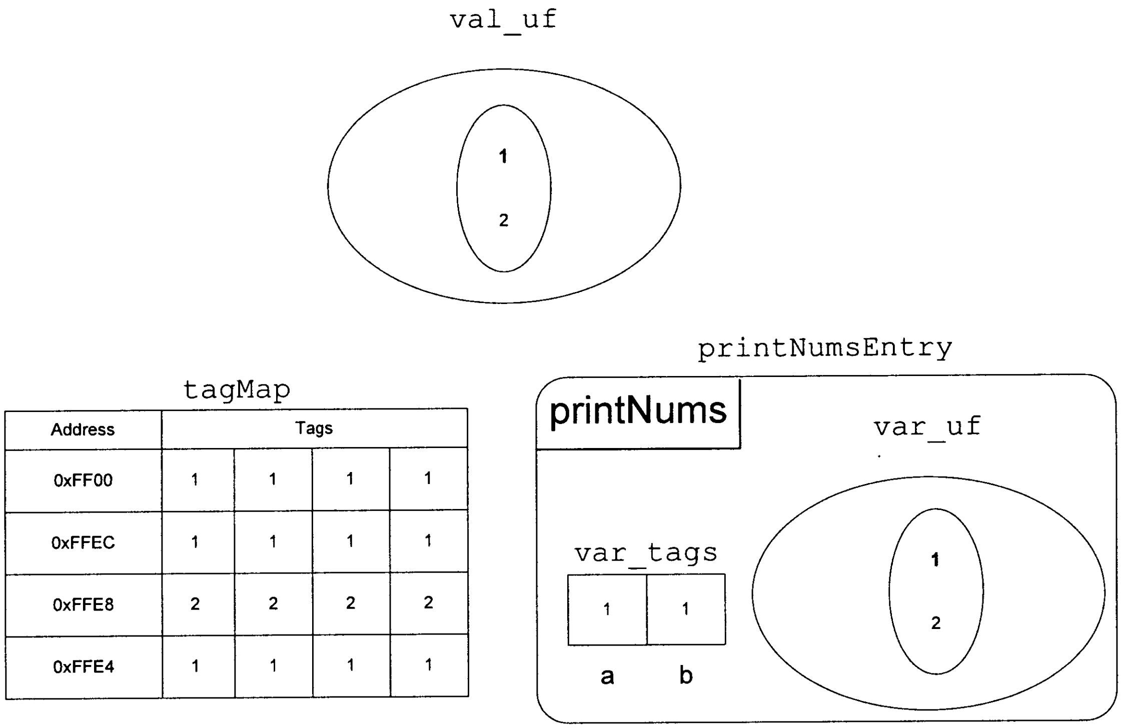
Figure 3-11: The contents of the same portion of the top of sum.c's stack at line 12. printNum's stack frame has since been removed from the stack, and term1's value has been updated due to the operation on line 11.



tagMap

Address	Tags			
0xFF00	1	1	1	1
0xFFEC	1	1	1	1
0xFFE8	2	2	2	2
0xFFE4	1	1	1	1

Figure 3-12: The contents of Dyncomp's primary data structures at line 12. The interaction on line 11 has been reflected in g_valUf by unioning the sets of tags 1 and 2.



Address	Tags			
0xFF00	1	1	1	1
0xFFEC	1	1	1	1
0xFFE8	2	2	2	2
0xFFE4	1	1	1	1

Figure 3-13: The final contents of Dyncomp's structures. Dyncomp's final propagation has caused the union in g_valUf to be reflected in printNumsEntry.varUf.

with several different fields is less than 8 bytes and passed as an argument to a function.

Figure 3-14 shows an example where this problem may occur. Depending on how the compiler chooses to align this struct in memory, its final size may be less than 8 bytes. If it does end up below this size, whenever getC is called with a struct of this type, the struct will be placed in a single register. When Dyncomp's variable processing function gets called at the entry to getC, it will update the tag of all the fields. Since a register can only have one tag, in.a, in.b, and in.c will all be observed with the same tag. This causes Dyncomp to place them all in the same comparability group. We hope to fix this issue in future versions of Dyncomp by having byte-granularity tag tracking for registers.

3.3.1 The effects of low-level library functions

Additionally, Dyncomp's analysis can be hindered by implementation details of low-level libraries. One such example is the C standard library function memset, which sets a block

```
struct small {
  int a;
  char b;
  char c;
}

int getC(struct small in);
```

Figure 3-14: A struct that is potentially under 8 bytes in size. If an instance of this struct is placed in a single register, all of its fields may be inferred as of the same abstract type.

```
movq        $0, myStruct(%rip)
movq        $0, myStruct+8(%rip)
movq        $0, myStruct+16(%rip)
```

Figure 3-15: memset for structs that are 24 bytes. Each $0 literal in the assembly causes the creation of a new tag, so each 8 byte chunk will have a separate tag.

```
movl        $0, %edx
movl        $myStruct, %edi
cld
mov   %eax, %ecx
movq  %rdx, %rax
rep stosq
```

Figure 3-16: A more general memset. Makes use of a rep stosq instruction. There is only one $0 literal, thus all the bytes of the struct will have one tag.

of memory to some input value. Some implementations of this function cause Dyncomp to view it as an interaction between every byte of memory that is set. Consider the following C statement: `memset(&myStruct, 0, sizeof(b));`, where `myStruct` is some C struct of arbitrary size. There are many possible ways to implement this. Figure 3-15 contains a memset implementation for x86-64 that is used for small structs. it simply contains a `movq` for every 8 byte chunk of the target struct. Figure 3-16 on the other hand makes use of the `rep stosq` instruction, which is essentially a loop using `%eax` as a source, `%ecx` as a counter, and `%edi` as the destination.

In figure 3-15, a new tag is being created for every 8 byte chunk of memory being replaced, due to the 0 literal in the assembly. In figure 3-16 only a single tag is created due to the fact there is only one 0 literal which is moved into `%eax` and then used as the source for all copies to the target destination. This has the effect of giving every field in the structure the same tag.

Conceptually, the problem stems from the fact that `memset`, contrary to the premise Dyncomp's inference algorithm, does not encode the programmer's information on the abstract types of the program. Rather, `memset` is an operation performed on a raw memory block, not high-level types. Ideally, Dyncomp would ignore `memsets` as they should not be considered when determining the abstract type groups of a program. One way to achieve this would be by tracking every call to `memset` and noting that any tags resulting from it are not to be tracked. Unfortunately, this is insufficient as our experience has shown that modern versions of GCC will inline `memset` as it sees fit, regardless of optimization settings. This makes it difficult to determine when a `memset` is being used by simply examining the resulting VEX IR.

3.4 Debugging Dyncomp errors

A program's abstract types are a property of its source code, thus Dyncomp's resulting abstract type groups should not vary if a program is run under different environments. One of the more difficult portions of Dyncomp's development has been getting it to produce the same results (modulo the limitations mentioned in section 3.3) for different platforms. This is difficult due to subtle differences in how operations are performed at a low level on different platforms. This section goes into the difficulties in determining why Dyncomp inferred two variables were of the same abstract type and a tool developed for providing developers and interested users a high level overview of why variables were reported to be of the same abstract type.

3.4.1 Determining the cause of abstract type group merges

Most Dyncomp errors result in Dyncomp incorrectly inferring that two variables that have distinct abstract types are of the same abstract type; these are frequent because it only takes a single interaction between values to cause the merging of abstract type groups. The opposite error, incorrectly inferring that an abstract type group is actually two separate abstract type groups, is much more rare as this requires Dyncomp to miss all of the (usually numerous) interactions between the groups. The first step a Dyncomp developer needs to take in fixing such an error is to determine why these two variables were determined to be comparable. That is, given an abstract type group with variables $v_0, v_1, ..., v_i$. a developer needs to determine why two variables v_a and v_b are members of the same group.

Determining why v_a and v_b are members of the same abstract group involves determining all relevant interactions that had an affect on the abstract type groups of these two variables. This is difficult for two primary reasons:

1. Value interactions are a critical part in inferring variable abstract type groups, but it is difficult to correlate a given value interaction with the variables it may affect. To properly track what value interactions affect what variables, Dyncomp would need a data flow tracking mechanism to find out what variables (if any) every live value in the program flowed from.

2. All interactions that involve a value which has been contained by any of the variables in the abstract type group are possible candidates for relevant interactions. v_a and v_b may not be in the same abstract type group due to an interaction between each other, but they both may have interacted with other members of the abstract type group.

3.4.2 Modeling abstract type inference as a graph reachability problem

An existing debugging output of Dyncomp contains all variable observations (section 3.2.3). There is also an option for printing a complete trace of all value interactions. In practice, however, we found the complete trace of all value interactions to be unsuitable for human understanding. These traces typically ranged from 500 megabytes for simple programs, to tens of gigabytes for an execution of the Perl interpreter on a test case that sorts a small array.

To reduce the complexity in understanding these trace files, we decided to model the problem of determining the cause of abstract type group merges as a graph reachability problem. In this model every value is represented as a node in the graph, and every interaction between two values is an edge between the values. The distinct value abstract type groups are represented by the connected components of the graph, one for each distinct abstract type group.

Figure 3-17 contains `temperature.c`, a simple program that calculates average temperatures. The graph representation of `temperature.c`'s value abstract groups is shown in figure 3-18; these abstract groups are based on the unit definition of interaction (section 3.2), thus the divisions on lines 17–19 do not affect the abstract types. There are multiple instances of the value 2, due to each 2 literal in the source being a unique value.

Variable abstract type groups can also be expressed using this model. The unioning of the abstract type groups of values that a variable has contained is can be represented by placing an edge between the values. The abstract type graph after adding these edges is shown in figure 3-19. The dashed edges represent a common variable between the two values. At this point it is clear that all of the temperature variables are of the same abstract type.

Implementation of a debugging tool

We decided it would be worthwhile to create a tool that would use the above model with the trace files produced by Dyncomp to allow Dyncomp developers to more easily find statements which cause interactions between abstract type groups.

Dyncomp's trace files contain 3 primary entry types:

- New tag entries - These entries contain the ID of the tag created and the source line corresponding to the current instruction. There is one new tag entry for every tag allocation during execution.

- Tag merge entries - These entries contain the two tags that just interacted, the new

```
1.   //temperature.c
2.
3.   int miamiTemp        = 90
4.   int orlandoTemp      = 75;
5.   int seattleTemp      = 45
6.   int sacramentoTemp   = 50;
7.   int bostonTemp       = 10
8.   int jerseyTemp       = 30;
9.
10.  int curYear = 2010
11
12.  int sumTemps(int temp1, int temp2)
13.    return temp1 + temp2;
14.  }
15.
16.  int main() {
17.    int southernAverage = sumTemps(miamiTemp, orlandoTemp)/2;
18.    int northernAverage = sumTemps(bostonTemp, jerseyTemp)/2;
19.    int westernAverage  = sumTemps(sacramentoTemp, seattleTemp)/2;
19.    return 0;
20.  }
```

Figure 3-17: Source code for `temperature.c`, a program that computes three temperature averages.

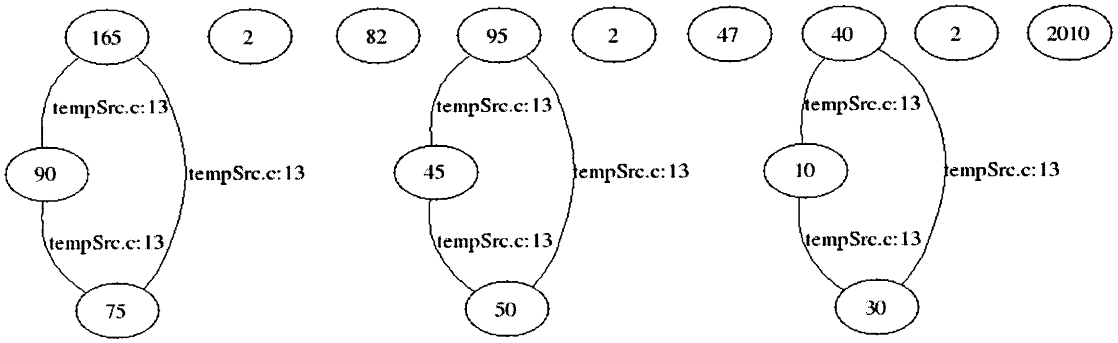


Figure 3-18: A graph representing the value comparability sets. `temperature.c`. Nodes are values in the programs and edges represent an interaction between the nodes. All connected components represent a distinct comparability set.

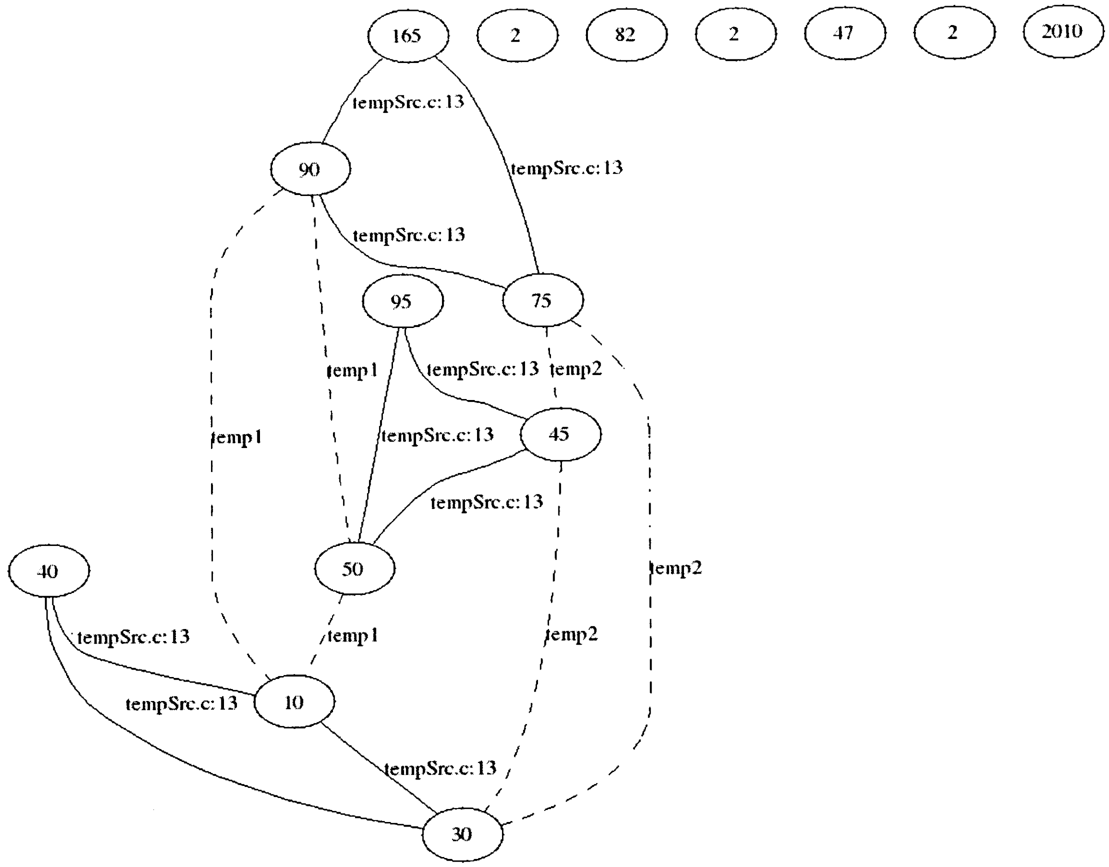


Figure 3-19: A graph reflecting the variable comparability sets. The dashed edges represent two values have been observed for the same variable.

leader tag, and the source line of the code that generated it. There is one tag merge entry for every observed interaction during execution.

- Variable observation entries - These entries contain the variable that has just been observed, its current and previous tag, and the name of the current function. There is one variable per variable per function entry and exit.

I created a tool to parse the trace files and create a graph representation similar to the one described in section 3.4.2. As before, unique tags are used to track the abstract types of the variables. A new node is created for every new tag entry and a new edge is created for each tag merge entry. A map, generated from the variable observation entries , between every variable in the program and the set of all tags observed for that variable. This map is created from the trace file's variable observation entries.

Once a trace file is fully parsed, a developer may query for an *interaction path*, a sequence of statements from the program's source responsible for the interactions that caused the merging of two abstract type groups. The tool determines this by attempting to find a path between any tag from the first variable's tag set and any tag from the second variable's tag set. There may be multiple interactions paths between two variables; however, in practice, this hasn't been much of an issue, because in most debugging situations, any interaction path between two variables is undesirable and worth investigation.

This tool aided me in debugging differences that arose when running Dyncomp on x86-64 hosts versus x86 hosts. In one situation, it allowed us to discover a bug that was causing spurious merges between global variables due to the new instruction pointer-relative addressing mode in the x86-64 instruction set. Occasionally, an index variable would be used in the instruction pointer-relative addressing mode, causing a merge between the index value and the value in the instruction pointer register. Due to new values very rarely being moved into the `%rip` register, every index variable was eventually merged together due to constant merges with the ip register.

In addition to serving as a debugging aid for Fjalar developers, this tool could also assist users of Dyncomp in understanding the resulting abstract type groups from Dyncomp, e.g., if the produced abstract type groups do not match the programmer's assumptions, it could be useful to see what lines are responsible for these unexpected groups.

Chapter 4

Kvasir, a Daikon front-end for C and C++ programs

Kvasir is a value profiling tool that records the runtime values of data structures. It serves as the default C/C++ front-end for the Daikon dynamic invariant detector.

This chapter outlines the implementation of Kvasir as a client tool of the Fjalar framework. It is intended for anyone who wishes to modify Kvasir, implement a Fjalar client tool, or is curious about the implementation of a scalable value profile tool for C/C++ binaries.

- Section 1 describes the requirement for a Daikon C/C++ front end.
- Section 2 describes how Kvasir produces Daikon declarations.
- Section 3 describes how Kvasir produces Daikon value traces.
- Section 4 describes how Kvasir makes use of Dyncomp to improve the detected invariants.

4.1 Requirements for a Daikon C/C++ front end

In order to be an effective Daikon C/C++ front end, Kvasir needs to:

1. Provide values for all variables at a given program point. It is not enough for Kvasir to only output trace information for variables that are actively accessed during the execution of a program point, but it should be able to provide values for all variables. Additionally, it should provide values for the fields of variables that are aggregate structures, such as C structs or C++ objects, and values that are pointed to by a pointer variable.

2. Be robust in the face of unsafe memory accesses. Kvasir should not crash the target program due to accessing an invalid memory address.

These requirements can be conflicting: to provide the value for pointed-to values (requirement 1), Kvasir will need to dereference pointer variables in the program's source; however, to be robust in the face of C's memory model (requirement 2), Kvasir should never dereference a pointer to an invalid memory location. To simultaneously meet both requirement, Kvasir examines every non-local pointer in the target program, but queries Fjalar for the validity of the location before attempting to access it (section 2.3 details how Fjalar is able to determine the validity of memory locations).

4.2 Creation of declaration files

A Daikon declaration file is a series of program point declarations, each of which contains a list of variable entries. A variable entry contains information on the variable's name, type, comparability, and enclosing object. These entries represent the program points and variables Daikon will attempt to determine invariants over.

Most of the work in determining the names and structure of variables is performed by Fjalar. Kvasir simply iterates through all the entries in Fjalar's table of functions, requesting a traversal of the function's variables with the function `printDeclsEntryAction` as a callback. `printDeclsEntryAction` transforms the `VariableEntry` and `FunctionEntry` structures provided by Fjalar into variable and program point declarations for the declaration file. This is a straightforward process as these structures already contain high level

```
ppt LinkedListNode:::OBJECT
  ppt-type object
   variable this
    var-kind variable
    dec-type LinkedListNode
    rep-type hashcode
  variable this->element
    var-kind field element
    dec-type int
    rep-type int
  variable this->next
    var-kind field next
    dec-type LinkedListNode
    rep-type
```

Figure 4-1: An object program point declaration for a linked list node.

string representations of the information required for the declaration file.

4.2.1 Handling object invariants

In addition to detecting invariants that hold true at a given program point, Daikon can, if desired, detect when an invariant holds true across multiple programs points and group them into a single invariant. More formally, given some set of (presumably related) program points, Daikon can find all variables that are common to all program points in the set, any invariants that hold over those variables at those program points, and group the invariants into a single invariant that holds over the entire set.

This facility is commonly used for classes: in the case of a class, the set of related program points are the class's public methods and the common variables are the class's fields. The invariants which hold over all methods are *object invariants*. It can also be used for `structs` in C.

For Daikon to detect object invariants for a class, the front-end must produce an object program point declaration for the class. An object program point declaration is a structural description of a class and contains the class's name and fields.

Figure 4-1 contains an example of an object program point record for a class represent-

```
struct LinkedListNode{
  struct LinkedListNode* next;
  struct LinkedListNode* prev;};
```

Figure 4-2: Declaration of a C struct

```
this       == this->next->prev
this->next == this->next->next->prev
```

Figure 4-3: Two object invariants

ing a node of a linked list. This record declares to Daikon that there is a class named LinkedListNode with three fields: this, element, and next. Detailed information on the format of program point declarations can be found in the Daikon developer manual at http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html.

To produce object program point declarations, Kvasir proceeds in two phases: first Kvasir traverses all the FunctionEntry structures provided by Fjalar, producing procedure program declarations and recording the name, fields, and superclasses of any classes encountered. Second, Kvasir iterates over the recorded class information, producing object program point declarations.

In addition to the object program point declarations, records for fields must contain the name of the object program point record corresponding to its enclosing class. This is easily produced as Fjalar provides a TypeEntry for the enclosing class of field variables (section 2.1.3).

While kvasir can emit object program point declarations for both C and C++ programs, by default object program points declarations are not emitted for C programs. An example of object invariants obtained for a C struct that represents a circular linked list Node is shown in Figure 4-3.

Figure 4-2 and figure 4-3 demonstrate Daikon can produce object invariants even if the underlying language has no built-in support for objects. Because Kvasir emitted an object program point record for the LinkedListNode struct and a reference to this object record for all instances of the struct. Daikon was able to produce invariants that held for all instances of the struct.

4.3 Creation of trace files

A Daikon trace file contains a trace of all executed program points for a given run of the target program. The trace file is organized into sections for each detected invocation of a program point. These program point sections contain the name and value of all relevant variables, where relevant variables are those declared under the program point in the declarations file. If the variable is either unallocated or uninitialized, the string ``uninit'' is printed as the variable's value.

To produce a trace entry, Kvasir requests Fjalar's variable traverser to visit all variables with Kvasir's variable printer as a callback. This printer will print the name of the variable, and, if safe, provide a value for the variable. Kvasir determines whether or not it is safe to dereference variable's address by asking Memcheck (through Fjalar) if all addresses associated with the variable are both allocated and initialized, see Nethercote [14] for details on how Memcheck determines whether or not program addresses are initialized or allocated.

4.4 Utilizing Dyncomp for more useful invariants

Variables entries can specify a comparability identifier, and Daikon will never attempt to compare variables with different identifiers. This allows the front-end to have a more fine-grained control over Daikon's production of invariants. For example take the following C snippet which declares 2 integer variables.

```
int curIdx, curYear;
```

curYear represents a calendar year,while curIdx represents an index into the array buffer. If provided input with no comparability information, Daikon may produce an invariants of the form curIdx <= curYear which, while true, may not be useful. One potential problem comes from the fact that curYear and curIdx represents different abstract types and comparison between them has limited use. In an effort to reduce less relevant invariants, Kvasir can use Dyncomp to determine the abstract types of the program.

Kvasir performs its comparability analysis by allowing Dyncomp to perform its analyses before Kvasir prints out any variable information. At the end of program execution, Kvasir will have access to Dyncomp's mapping between variables and their comparability sets. Kvasir will then print out the declarations, looking up a variable's comparability set in Dyncomp, and mapping that comparability set to an integer. All variables within a given comparability set will have the same comparability ID. Utilizing Dyncomp in this way allows Kvasir and Daikon to produce much fewer spurious invariants than a standard run. Using Dyncomp, we've seen the number of invariants reduced by upwards of a factor of 50. Guo [11] presents a detailed evaluation of Dyncomp's use in dynamic invariant detection.

Chapter 5

Conclusion and Future work

This thesis presented the in-depth implementation of the Fjalar (chapter 2) C/C++ dynamic binary analysis framework and two Fjalar tools: Kvasir(chapter 4) and Dyncomp(chapter 3), all of which were originally developed by Philip Guo and the MIT Program Analysis Group.

Additionally, it presents my contributions to increasing the maintainability of the Fjalar framework by documenting platform dependencies (section 2.5.1), requirements for keeping Fjalar up to date, and the procedure for merging upstream changes in Valgrind and Memcheck. (section 2.5.2). It presents my work on simplifying the debugging of Dyncomp comparability set errors (section 3.4): the implementation of a tool to query for sequences of source statements that caused the merging of abstract type groups.Finally, it described an extension I've made to Kvasir that allows the output of object program points for use by Daikon (section 4.2.1).

5.1 Future work

There are many features and improvements we'd like to make to the framework and tools detailed above:

5.1.1 Fjalar

- Design and implement a more general method for controlling how Fjalar traverses the inside of data structures at run-time.

- Add support for traversing local variables. This would almost be trivial to add, as their `VariableEntry`s are created during DWARF parsing.

- Improve support for optimized C/C++ binaries. Newer versions of the `gcc` compiler suite provide debugging information that is comprehensive enough to obtain information on optimized binaries, thus this wouldn't be too difficult.

- Add support for new platforms. While Valgrind provides most of the support Fjalar needs to support a platform, there are some situations where Fjalar must implement a portion of the system's ABI (section 2.5.1).

- Extend Fjalar to support the other languages the `gcc` compiler suite supports.

- Use Fjalar to build new dynamic analysis tools. One idea is a tool that can infer unit relationships (e.g., physics units), much like Dyncomp can infer abstract types.

- Improve support for non-local exits, such as `setjmp/longjmp` in C and exceptions in C++. Currently a Fjalar tool's exit instrumentation will not be executed for functions that exit non-locally.

5.1.2 Kvasir

- Implement more general value profiling in Kvasir so that it might be useful for other tools besides Daikon. Another possibility would be to extend the current value profiling too be able to create summaries of data structures with forms and shapes so the results could be used by a human to improve program understanding.

5.1.3 Dyncomp

- Implement byte-granularity tags for registers. This was covered in section 3.3: currently, Dyncomp only maintains one tag per architectural register which results in errors in the resulting abstract type groups. This problem is more pronounced for AMD64 hosts, due to calling conventions which allow entire structs to be placed into a single register.

- Investigate speedups. Currently Kvasir runs about an order of magnitude slower with Dyncomp than without. We believe that this could be lowered to about to a factor about 60-80x, which is the slowdown imposed by Memcheck alone. This should be possible as Dyncomp's implementation parallels Memcheck in many ways.

Bibliography

[1] System V Application Binary Interface – AMD64 Architecture Processor Supplement.

[2] System V Application Binary Interface – Intel386 Architecture Processor Supplement.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[4] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 276–285. IEEE Computer Society Washington, DC, USA, 2003.

[5] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, page 422. ACM, 2007.

[6] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.

[7] Valgrind developers. Valgrind user manual. `http://valgrind.org/docs/manual/manual.html`.

[8] M.J. Eager and E. Consulting. Introduction to the DWARF debugging format, 2007.

[9] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[10] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 18–20, 2006.

[11] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 5, 2006.

[12] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *5th International Workshop on Program Comprehension (WPCo97)*, pages 80–85, 1997.

[13] Sri Hari Krishna Narayanan, Seung Woo Son, Mahmut Kandemir, and Feihui Li. Using loop invariants to fight soft errors in data caches. In *Asia and South Pacific Design Automation Conference*, pages 1317–1320, Shanghai, China, January 18–21, 2005.

[14] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. page 74, 2007.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. 2007.

[16] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[17] J. Silverstein. DWARF Debugging Information Format. *Proposed Standard, UNIX International Programming Languages Special Interest Group*, 1993.

[18] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):195, 2001.

[19] Tao Xie and David Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.