Bridging the Gap Between Binary and Source Analysis

Philip J. Guo Stephen McCamant Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab
32 Vassar St., Cambridge, MA 02139 USA
{ pgbovine,smcc,mernst } @csail.mit.edu

ABSTRACT

Dynamic analyses for software engineering typically operate either at the source code level or at the binary level (possibly postprocessing results to source code terms for output). We propose a *mixed-level* approach that combines the source-level and binary-level approaches throughout the duration of the analysis. Compared to a one-level approach, the mixed-level approach simplifies implementation, improves robustness, and enables analyses that are impossible or impractical to perform purely at the source or binary level.

We have implemented a dynamic instrumentation toolkit, named Fjalar, that embodies the mixed-level approach, and we present two distinct analyses that are built upon the toolkit. The first tool performs value profiling — outputting a rich set of run-time values for further analysis. The other tool performs value partitioning — determining abstract types for concrete values. Compared to similar tools that use a source-based approach, the mixed-level tools built upon Fjalar were both easier to implement and more scalable, handling C and C++ programs of hundreds of thousands of lines.

1. INTRODUCTION

Program analysis is typically performed either on source code or on a compiled (binary) representation of a program. These representations have complementary advantages: for instance, problems are often posed in terms of source-level constructs, but binary analyses can be more robust and scalable.

Our goal is to make it easier for a dynamic analysis to obtain the key benefits of both source and binary analysis. We propose a *mixed-level* approach that bridges the gap between source-level and binary-level analysis, performing binary-level instrumentation while integrating source code information throughout the analysis. This approach allows analyses that are posed in terms of source-level constructs to be performed with the ease-of-development, ease-of-use, and scalability benefits of binary analyses.

As an example, consider an alias analysis, which reports whether two pointer variables might simultaneously refer to the same object. A dynamic alias analysis can detect whether two pointers were ever aliased during a set of executions. A profile-directed optimization can use the alias results, transforming the code to check whether the pointers were different, and if so to use a code path that allocates the pointed-to values in registers [6]. Such an analysis could be performed naturally in the mixed-level framework we introduce: a tool could observe at the machine level each instruction operating on an address, and then record its effect in terms of the corresponding language-level pointers. By contrast, other commonly used approaches would be much more cumbersome.

- A source-to-source translation tool could track each pointer modification by inserting recording routines for each operator at the source level, but such a strategy would be unwieldy, especially in dealing with the syntax and semantics of realistic languages.
- A technique that recorded information in a purely binary form, and then post-processed it to print using source terminology, would not be workable because the mapping between machine locations and language-level pointer expressions is needed to interpret each operation; such an approach would essentially have to store a complete trace.

This pattern of trade-offs applies to many dynamic analyses: we want output in terms of source constructs, but a binary analysis would be more natural to implement. Rather than a purely source-based analysis, a purely binary-based one, or a binary analysis with source information added as a postprocessing step, we suggest a mixed-level approach that performs a mainly binary analysis, supplemented with a mapping to the source level used throughout the analysis to interpret machine-level data and operations in terms of the source constructs.

We have built a toolkit, Fjalar, for machine-language dynamic instrumentation that supports the mixed-level approach. The toolkit allows for whole- or partial-program instrumentation of machine language binaries written in arbitrary languages, but takes advantage of limited source-level debugging information and can report results at the language level. The toolkit, built upon the Valgrind [19] binary translation tool for rewriting x86 executables, currently supports any C or C++ dialect that is compilable with `gcc`. It is easy to use, general, robust, and flexible. It has been used as the basis for a value profiling tool and a value partitioning (dynamic type inference) tool, and extending it to support C++ required just 4 days.

To assess whether the toolkit allows robust tools to be easily developed, we present two case studies addressing the profiling and type inference tools, respectively. Each case study compares a tool built using the Fjalar mixed-level toolkit with a tool built using a different methodology, discussing both the difficulty of implementation and the quality of the resulting tool. The Fjalar-based systems were easier to implement and the resulting tools were more scalable. These results suggest that the mixed-level approach is applicable to a range of dynamic analyses for software engineering.

The rest of the paper is organized as follows. Section 2 contrasts source- and binary-based approaches to dynamic analysis. Section 3 presents several approaches that combine source and binary information, including the mixed-level approach as implemented by the Fjalar toolkit. Section 4 introduces the problem of value profiling for software engineering and compares two implementations, one source-based and the other mixed-level. Section 5 describes another application of a mixed-level analysis, to the problem of value partitioning. Section 6 presents related work, and Section 7 concludes.

2. SOURCE-BASED AND BINARY-BASED DYNAMIC ANALYSIS

A dynamic program analysis observes a program's behavior at runtime. Dynamic analysis can be used for optimization (profiling, tracing, optimizations), error detection (testing, assertion checking, type checking, memory safety, leak detection), and program understanding (coverage, call graph construction); these categories are not mutually exclusive.

The two most common instrumentation techniques to obtain information from a running program are to modify a program's source code, or to modify a compiled binary representation of a program. (Other techniques exist but are less general. Linking a program with a modified version of a standard library is only applicable to certain analysis problems. Modifying a language interpreter is only applicable to certain target languages. Thus, we will not consider them further.) The following subsections describe the different advantages of source-based and binary-based analyses.

2.1 Source-based instrumentation

A source-based approach modifies the code of the target program (the program being analyzed) by adding extra statements that collect data or perform analysis. Code instrumentation by source-code rewriting is the most direct route to constructing a tool that produces language-level output, and it also makes some aspects of tool implementation relatively easy.

An analysis that operates by rewriting a target program's source code can use a high level of abstraction (that of the programming language) and can report results using language-level terms such as functions and variables. A source-based analysis can also inherit the portability of the underlying program.

A source-based analysis can also be relatively easy to implement. The developer only needs to consider one level of abstraction, that of the instrumented language. Standard programming tools suffice to examine and debug the output of a source-to-source rewriting tool. Compiler optimizations automatically reduce the overhead of instrumentation.

2.2 Binary-based instrumentation

A binary-based approach modifies a compiled executable to add instrumentation code. Some analyses can be most directly expressed at a binary level, and binary-based tools are usually easier to use once written.

The most important advantage of a binary-level analysis is that many analysis problems can be expressed more simply at a lower level of abstraction. At the syntactic level, a binary is a flat list of instructions rather than a nested expression that requires parsing. At the semantic level, there are fewer conceptually distinct machine operations than language-level abstractions, and the machine operations are much simpler. For instance, the language-level description of data has a complex structure in terms of pointers, arrays, and recursive structures. By contrast, the machine-level representation of data is as a flat memory with load and store operations. If the property to be analyzed can be expressed in terms of the simpler machine representation, the language-level complexities can be ignored.

There are also three ways in which binary-based analysis tools can be easier to use. First, a binary tool need not be limited to programs written in a particular language: Language-level differences between programs are irrelevant as long as the programs are compiled to a common machine representation. Second, a binary tool need not make any distinction between a main program and the libraries it uses: execution in a library is analyzed in just the same way as other program execution. There is no need to recompile libraries or to create hand-written simulations or summaries of their behavior as is often required for source-based analyses. Third, a binary-based tool requires fewer extra steps to be taken by a user (none, if the instrumentation occurs at runtime). A source-based analysis at least requires that a program be processed and then recompiled before running; this can be cumbersome, because compiling a large system is often a complicated process.

3. TWO-LEVEL APPROACHES TO DYNAMIC ANALYSIS

A two-level approach can combine features of source- and binary-based analyses by collecting information via binary instrumentation and performing other operations, including output, in language-level terms. Such an approach has the potential to produce language-aware output as from a source-based analysis, with the robustness and scalability of machine-level instrumentation. The mixed-level approach, presented in Section 3.2, improves upon previous two-level approaches by tightly integrating both types of information throughout the duration of the analysis.

3.1 Previous two-level approaches

Many binary-based dynamic analysis tools incorporate at least a small amount of source-level information as a post-processing step in order to produce human-readable output. For instance, a binary-based tool may discover a bug (say, an illegal memory operation) at a particular instruction. After the execution terminates, the tool translates the address, which would not in itself be helpful to a user who is trying to fix the error, to a line number in the program source code. Most uses of source information by binary analyses are limited to this sort of incidental postprocessing. However, for many analyses (such as the alias analysis of Section 1, the profiling of Section 4, and the type inference of Section 5), only using source-level information as a postprocessing step causes a loss of accuracy, because the results depend upon updating information about source constructs (such as functions and variables) while the analysis is running.

More benefit can be achieved by linking the source and binary levels more pervasively, but a tight integration of the two is uncommon, especially in automated analysis tools. A good example of such a tight integration is an interactive symbolic debugger. At each step in observing a program's execution, the user can access both source and binary-level information. For instance, a user can step either by source line or by instruction, and read data both as raw bytes and as variables and data structures. However, symbolic debuggers are not designed to be used in an automated fashion and thus do not provide certain services (e.g., maintenance of dynamic array sizes, memory validity checking) that are useful for building dynamic analyses. Our goal is to take this flexible approach, previously implemented in a special-purpose tool for interactive use, and integrate it with a larger class of automatic binary-level program analyses.

3.2 Mixed-level approach

We propose a variety of the two-level approach that utilizes binary and source-level information simultaneously during the analysis. This *mixed-level* approach can provide more accurate results with an easier and more robust implementation than binary-based, source-based, or two-level approaches that utilize the two types of information at different times. In the mixed-level approach, most of the tracking of a program's run-time behavior is performed at the instruction level, obtaining the benefits of binary analysis. When it is necessary to use source-level abstractions as the analysis is running, either as one step in the analysis or for output, the low-level binary information can be translated into a language-level representation. This translation requires a limited form of source information (obtained, for instance, from debugging information inserted by the compiler), but need not consider all of a language's source-level complexities.

Not every source-level analysis can be conveniently translated to operate via binary instrumentation. When such a translation is possible, though, a mixed-level approach can achieve the same or better results than source-level instrumentation with a simpler and more flexible implementation.

The primary advantage of the mixed-level approach over other two-level approaches is the fact that it can simultaneously utilize both binary-level and source-level information throughout the duration of the analysis. For example, in Figure 1, no binary-level information indicates array sizes. Source-level information is required during the course of the analysis in order to direct it to only read selected regions of memory. It is possible to perform the same analysis by collecting coarse-grained snapshots of memory and only using source-level information about array sizes and variable locations as a postprocessing step, but that would be much more difficult to implement and infeasibly inefficient.

3.3 The Fjalar toolkit

We have designed and implemented a toolkit for building dynamic analysis tools that embodies the mixed-level approach. The Fjalar toolkit (named after a dwarf in Norse mythology) incorporates variants of the Valgrind, Memcheck, and Readelf tools and provides functionality for integrating them in order to simplify common dynamic analysis tasks. Its API provides services that are useful for many types of dynamic analyses: binary rewriting, memory allocation and initialization tracking, mapping between memory addresses and language-level terms such as variables and functions, and recursive traversals of source-level data structures during runtime.

We have used the Fjalar toolkit to develop two different dynamic analyses (described in Sections 4 and 5) based on the mixed-level approach. Our experience shows that Fjalar can be used to build tools that are quite scalable and have acceptable performance given the unavoidable overhead involved in performing a fine-grained analysis. This section describes the components of Fjalar.

3.3.1 Binary instrumentation

A dynamic analysis requires a method to instrument the target program so that it can provide and process the desired information during runtime. Fjalar uses Valgrind [19], a program supervision framework based on dynamic binary rewriting, to insert instrumentation operations in the target program execution. Because it operates at the machine level, Valgrind is naturally language-independent and makes no distinction between user and library code. However, it does depend on the calling conventions of the particular binary format and on the operating system, so the current version only works for ELF binaries on x86/Linux systems (although it is under active development to support additional platforms [24]). Fjalar's users can insert code in the target program that accesses machine-level information about registers, memory, and instructions. For instance, a tool built upon Fjalar can insert instrumentation code to collect and analyze data gathered at certain points in the target program's execution (e.g., function entries and exits).

3.3.2 Memory safety

Memory safety is crucial for a dynamic analysis, especially of languages such as C and C++ which allow programmers to directly manipulate memory, to be robust to crashes and to produce correct and precise results. For instance, variables and pointers may be uninitialized; the analysis must suppress or flag junk values, so as not to corrupt the results. Memory may be deallocated, making a pointer invalid; dereferences of such a pointer yield either junk values or a segmentation fault. Memory may be uninitialized, just like pointers, so an initialization analysis must track all of the heap, stack, and registers. Values can be freely cast to different types, necessitating fine-grained tracking — at least at the byte level. Pointer types are ambiguous; a pointer of type `int*` may point to a single integer, or to an array of integers. Array lengths are implicit; even if an `int*` pointer is known to be an array, the run-time system provides no indication of its size. Related to an earlier point, even if the array's size is known, it is not known which of its elements have been initialized. Furthermore, if the target program has memory-related bugs, it is important that the analysis tool not crash, even if the target program attempts a truly illegal operation. Programs may have latent memory-related bugs which do not disrupt normal execution, but do appear when running under an analysis tool.

Fjalar utilizes a modified version of Memcheck [23], another Valgrind tool, to provide the requisite memory safety guarantees. Memcheck is typically used to detect memory errors such as memory leaks, reading/writing unallocated areas of memory, and the use of uninitialized values. Similar to Purify [14], Memcheck tracks whether each byte of memory has been allocated (by assigning an *A-bit* to every byte which is set only when that byte is allocated) and whether each bit of memory has been initialized (by analogously assigning a *V-bit* to every bit). Memcheck rewrites the target program's binary to copy and maintain A- and V-bits throughout memory and registers. Fjalar's API allows tool builders to query the A- and V-bits in order to ensure that memory accesses do not result in a segmentation fault or invalid data, respectively.

3.3.3 Integrating source information during analysis

Many dynamic analyses need access to source-level constructs such as variables and functions during execution. Fjalar maps between source-level terms and binary-level data using symbolic debugging information generated by compilers. It uses the DWARF2 [12] standard for representing debugging information in a compiled binary, so it can work with any language for which a compiler can produce such information (including any language supported by `gcc`). This may require the target program to be recompiled, but because compiling with debugging information is a common development practice, usually such recompilation is not difficult. Debugging information is only required for the parts of the program for which results are requested; in particular, unmodified versions of the standard system libraries can be used. We wrote a parser for debugging information based on the Readelf tool from GNU Binutils [11], then customized it slightly for differences between languages, which was much less work than writing separate source code parsers for each language.

Given an expression that represents a storage location (i.e., an lvalue), Fjalar's API allows a tool to find the address and size of that object at a particular moment in the target program's execution. This source-level information, coupled with the A- and V-bits, allows tools to not only be able to safely access valid regions of memory, but more importantly, to associate semantic meanings with observed memory contents. For example, without source-level information during the analysis, a tool can only tell that a certain block of bytes contains some binary value, but with such information, it can interpret that binary value as an integer, floating-point number, string, etc.

```
struct record {
  int *a, *b;
  int c[10];
}

void foo(struct record rec) {}

int main() {
  int localArray[100];
  ... initialize the 100 ints within localArray ...
  struct record r;
  r.a = localArray;
  r.b = (int*)malloc(sizeof(int));
  foo(r);
}
```

Figure 1: A C program that mixes structs, arrays, and pointers. For example, `rec.a` is a pointer to `localArray`, and it is useful for a tool to recognize that `rec.a` refers to an array of 100 integers and be able to observe information about these values. Fjalar's API provides a mechanism for safe recursive traversal of structs and arrays.

3.3.4 Recursive data structure traversal

Fjalar gives users the ability to iterate through the contents of source-level variables, arrays, and data structures at runtime while maintaining the robustness, coverage, and language-independence of a binary-level analysis. Most non-trivial programs involve the use of data structures such as arrays, linked lists, trees, and aggregate types (such as structs and classes). Many kinds of dynamic analyses can benefit from run-time observation of the contents of these data structures, even at times when the target program did not directly manipulate these structures. A tool that only observes data at times when the program manipulates it is easier to build, but produces limited information. Fjalar enables the construction of more powerful tools that can traverse any data structure in scope. For example, in Figure 1, `foo`'s argument `rec` is a structure that contains two pointers and a static array. To observe all of its contents, a tool must be able to follow valid pointers and recursively traverse inside of structures and arrays, observing the values of the pointers `rec.a`, `rec.b`, `rec.c`, and the arrays referred to by those pointers: `rec.a[]`, `rec.b[]`, `rec.c[]`.

A purely binary-based analysis cannot accomplish such detailed observation absent some indication of how to interpret raw binary values as data structures. It is possible but extremely complicated to accomplish such observation with a source-based analysis, because it must parse and generate complex source syntax which deal with data structures and, more significantly, maintain metadata such as pointer validity, memory initialization, and array sizes.

Fjalar's API provides recursive data structure traversal functionality, allowing tools to observe the contents of arrays, follow pointers to observe structures such as linked lists and trees, and recursively traverse struct fields, all while ensuring memory safety so that the analysis does not crash the target program. For an address in the global area or stack, the debugging information indicates the names, types, number of elements, and locations of statically-sized arrays. For an address in the heap, Fjalar determines the size by probing in each direction for "redzones" that Memcheck inserts around dynamically-allocated regions of memory to detect array bounds overflow errors. This technique is not perfect, but it has worked well in practice.

4. VALUE PROFILING CASE STUDY

This section presents a case study of two separate C/C++ value profiling implementations. The first, Dfec, works at the source level, and the second, Kvasir, uses the mixed-level approach as implemented in the Fjalar toolkit.

Value profiling [6] is a technique that observes the run-time values of variables, expressions, registers, memory locations, etc. Value profiling is a general technique that is incorporated in any dynamic analysis that is concerned with what the program computes. (Purely control-flow-based dynamic analyses, such as coverage tools, need not incorporate value profiling.) A form of value profiling is even implemented in hardware, in order to support speculation, branch prediction, and other optimizations.

Because of its importance and generality, many different approaches to value profiling exist. Section 4.1 describes the requirements for a general value profiling tool for software engineering applications that can provide information about variables and expressions in a C or C++ program. Then, Sections 4.2 and 4.3 contrast two implementations that aim to meet these requirements.

4.1 Requirements

A value profiling tool for software engineering applications should provide accurate and rich information about the run-time values of arbitrary data structure accesses. The desire for accurate information requires fine-grained tracking of pointer and memory use, to determine when a particular expression's value is meaningless (e.g., uninitialized). The desire for rich information means that it is not enough to merely observe values that the program is directly manipulating at a moment in time; other values may be of interest, even if their uses appear before and/or after that moment. Value profiling for software engineering can be viewed as having the characteristics of three other analyses: traditional value profiling, data structure traversal, and memory error detection.

A profiler observes program behavior (such as performance, control flow, or values) but strives not to change that behavior. When it is acceptable to produce only limited information, one way to avoid changes in program behavior is to observe only values that the program directly manipulates. For instance, a value profiler could record the value of a given expression only at instructions that read or write that expression. This ensures that the value is valid, and that the access to it does not cause a segmentation fault or other behavioral change. Software engineers may be helped in some tasks by such limited information, but additional information can be very valuable. A value profiling tool for software engineering should be able to, without causing the program to crash, examine the values of arbitrary data structure elements that the target program did not.

4.1.1 Value profiling application

Value profiling is a broadly applicable technique that can supply information to a human or a subsequent tool. For concreteness, we consider a specific application, dynamic invariant detection [9, 21], for which the two tools we compare (Dfec and Kvasir) were constructed. The Daikon dynamic invariant detection tool (`http://pag.csail.mit.edu/daikon/`) reports likely program invariants by performing machine learning over the values of variables (and other expressions) during a program execution. The result of the analysis is a set of properties, similar to those found in formal specifications or assert statements, that held during the observed executions. Daikon itself is language-independent; its input is a trace of variable names and values. Daikon must be coupled with a language-specific front end (a value profiler) that instruments a target program to produce the trace during execution. Daikon front ends exist for C/C++, Java, Perl, and other languages and data formats. The two programs of this case study, Dfec and Kvasir, are

Original:

```
bool g;  // global variable
int foo(int x) {
  ...
  return g ? x++ : -x;
}
```

Instrumented by Dfec:

```
int foo(int x) {
  trace_output("foo():::ENTER");
  trace_output_int("x", x);
  trace_output_bool("g", g);
  ...
  int return_val = g ? x++ : -x;
  trace_output("foo():::EXIT");
  trace_output_int("x", x);
  trace_output_int("return", return_val);
  trace_output_bool("g", g);
  return return_val;
}
```

Figure 2: Source code before/after instrumentation by Dfec.

both Daikon front ends for C/C++.

At each *program point* for which likely invariants are desired (by convention, procedure entries and exits), the trace indicates the value of each variable that is in scope. At function entries, these are global variables and formal parameters, and at function exits, they are global variables, formal parameters, and return values. We call these variables, as well as those that result from traversing inside of data structures held by these variables, the *relevant variables*, and they permit Daikon to infer procedure preconditions and postconditions. C++ member functions are treated like normal functions with an extra `this` parameter, and generalization over preconditions and postconditions yields object (class) invariants.

4.2 Source-based approach: Dfec

Dfec (Daikon Front End for C) [17] is a source-based front end for the Daikon invariant detector. Dfec works by rewriting the source code of the target program to insert code that outputs the values of relevant variables (see Figure 2).

A user runs Dfec to instrument the target program's source code, compiles the instrumented source, then runs the resulting executable. As the program executes, it outputs the names and values of relevant variables at each execution of a program point. Although designed to support both C and C++, Dfec is usable in practice only for small C programs.

Dfec uses the EDG C/C++ front end [8] to parse, instrument, and unparse source code. This source code rewriting works well for outputting values, but information about initialized and valid variables and memory must be maintained dynamically. Dfec includes a sophisticated and complex run-time system that associates metadata with each pointer and chunk of memory. It inserts code that checks and updates the metadata at allocations, assignments, uses, and deallocations of memory. Syntactic changes to the source program are reduced by defining 'smart pointer' C++ classes, instances of which check and update the metadata when the program performs pointer operations.

4.2.1 Advantages of Dfec

Dfec exemplifies some benefits of a source-based approach. It collects and outputs information according to a single structure, that of the C language, so no complex translation is required. It is architecture independent: it was relatively easily ported to several operating systems and architectures. The compiler optimizes the instrumentation along with the rewritten program, yielding relatively small run-time overheads (see Section 4.4.2 for measurements). However, some other benefits of a source-based approach are not fully realized in Dfec: some remaining dependencies on nonstandard library and compiler features limit its portability, and debugging its source-code output is difficult.

4.2.2 Limitations of Dfec

Though some of Dfec's shortcomings could be attributed to design choices we would make differently in retrospect, it also suffers from limitations that are virtually unavoidable in a source-based analysis.

Because of the complexity of tracking memory use at the source level, Dfec does not correctly handle the full diversity of input language constructs. The particular problems we found are results of Dfec's design, but other source-rewriting strategies would likely have similar complexities. For instance, we encountered problems triggered by `typedef`ed `void *` pointers, and by ternary `?:` operators in which one argument is a string literal. No one such problem is insurmountable, and in theory, the supply of such problems is finite, but the complexities of source processing make it hard to solve all of them.

Language-level complexity was also a major obstacle in the efforts to extend Dfec to support C++ input programs. Even discounting the difficulties of parsing, C++ is a much more complex language than C at the source level. Though C++ support was originally a goal for Dfec, and the last months of Dfec's development focused on that goal, it was very challenging to make it work robustly. For instance, one source of problems was interactions between the templates used to represent smart pointers, and other uses of templates in the original program.

Dfec also suffers, somewhat less severely, from three other problems that tend to affect source-based analyses. First, though Dfec re-uses an existing C++ parser, the AST interface alone is rather complex, making maintenance difficult. Second, because Dfec is unable to rewrite the system libraries to track their memory usage, Dfec contains special interface stubs for standard functions such as `strcat` that properly update the pointer and memory metadata. For programs that interact with more substantial libraries, the work of creating such stubs would be prohibitive. Third, because Dfec works by processing source code that must then be compiled, it is cumbersome to use, especially for large programs. Particular care is needed for programs that consist of multiple compilation units, and the rewritten code must be compiled with a different (C++, not C) compiler and linked with an additional library.

Dfec does not work "out of the box" on C/C++ programs of reasonable size, and even a complete source-based reimplementation would not resolve many of its most debilitating limitations. This experience motivated the development of a new Daikon front end based on a mixed-level approach.

4.3 Mixed-level approach: Kvasir

Kvasir (named after the Norse god of knowledge and beet juice) is a C/C++ front end for Daikon based on the mixed-level approach. Kvasir instruments and executes the target program's binary in one step without using the source code. Kvasir works "out of the box" on programs of significant size (see Section 4.4.1), and its scalability and performance surpass those of the invariant detector itself.

4.3.1 Implementation of Kvasir

Kvasir is built on the Fjalar mixed-level toolkit. It uses Fjalar's binary rewriting API to instrument the target program's executable, inserting value tracing hooks to execute its own code. It utilizes the recursive data structure traversal functionality provided by Fjalar (Section 3.3.4) to read and output the values of all relevant variables and expressions. It uses Fjalar's memory safety API to avoid crashing or returning nonsense during a memory read, and unlike Dfec, does not need to maintain any pointer metadata because Fjalar can determine array sizes and pointer validity at runtime.

4.3.2 Advantages of Kvasir

Kvasir's use of binary analysis has three interrelated advantages. First, it is precise to the bit level; for instance, Kvasir can print the initialized bits but not the uninitialized ones in a bit vector. Second, the precision is not diminished when libraries are used, since code is treated uniformly regardless of its origins. Third, and most importantly, the binary memory tracking is conceptually simple, allowing a simple and robust implementation.

Kvasir avoids complexity by not depending on details of the target program's source language. Kvasir's memory tracking is designed with respect to a simple machine model, rather than a complex language semantics. While the sophisticated structure of debugging information accounts for much of Kvasir's complexity, it is still much simpler than the original source from which it is derived (many source-level aspects are abstracted away by the compiler) and is well-encapsulated within the Fjalar toolkit.

The best example of this reduced dependence on source complexities was our experience in adding support for C++ to Fjalar and Kvasir (at first, the implementation only supported C). This primarily required additional debugging information parsing code to handle object-oriented features such as member functions and static member variables, and in all required only 4 days of work. It would likely be just as easy to support other gcc-compiled languages such as Ada, Fortran, Objective-C, and Java.

A final advantage of Kvasir is that it is easy to use. Running a program under Kvasir's supervision is a single step that involves just prepending a command to the normal program invocation. For a program that is normally run as

```
./program -option input.file
```

a user would instead use the command

```
kvasir-dtrace ./program -option input.file
```

It is rarely necessary to modify a program in order to use it with Kvasir, even for large programs that use many language and system features (see Section 4.4.1).

4.3.3 Limitations of Kvasir

Like the Fjalar toolkit upon which it is built, Kvasir currently supports only x86/Linux systems (see Section 3.3.1).

Another limitation is that a program producing trace data with Kvasir can take 1000 or more times longer to run than the program without instrumentation. This slowdown is a combination of two factors: First, even when not producing any trace output, there is a significant overhead (of 50–100 times) which includes Valgrind's dynamic translation (designed for extensibility rather than efficiency), the memory usage tracking performed by Memcheck, and other bookkeeping related to tracing, such as deciding whether to trace a function at all. Second, because the tracing performed by Kvasir is very detailed, an even larger amount of time is required to print the trace; much of this expense simply reflects I/O. The expense of tracing is of course proportional to the amount of output desired; in the experiments of Section 4.4.2, it represented an additional slowdown factor of 20–80 times. While this slowdown seems large compared to less detailed kinds of dynamic analysis, it is largely unavoidable for producing so much data. Since Kvasir usually produces data faster than Daikon can process it, improving Kvasir's performance would not markedly improve the performance of the Kvasir–Daikon invariant detection system; thus, doing so has not been a priority.

Program	Lang.	LOC	Dfec	Kvasir	Lackwit	DynComp
md5	C	312	Y	Y	Y	Y
rijndael	C	1,208	Y*	Y	Y	Y
bzip2 1.0.2	C	5,123	Y*	Y	Y	Y
flex 2.5.4	C	11,977	Y**	Y	Y	Y
make 3.80	C	20,074	N	Y	I†	Y
xtide 2.6.4	C++	23,768	-	Y	-	Y
groff 1.18	C++	25,712	-	Y	-	Y
civserver 1.13.0	C	49,657	-	Y	Y	Y
povray 3.5.0c	C++	81,667	-	Y	-	Y
perl 5.8.6	C	110,809	-	Y	I†§	Y
xemacs 21.4.17	C	204,808	-	Y*	N†§	Y*
gcc 3.4.3	C	301,846	-	Y*	I†	Y*

Y	runs to completion	-	did not attempt
I	incomplete run	†	fatal error
N	failed to run	§	parse failure (≥1 file)
*	requires minor modifications reasonable for users to make		
**	requires major modifications that would deter most users		

Table 1: Scalability tests for open-source Linux C and C++ programs (Lackwit does not support C++). LOC is non-comment non-blank lines of code.

4.4 Experimental results

This section compares Dfec and Kvasir in terms of scalability and performance. Scalability was measured by the sizes of programs that Dfec and Kvasir could successfully process and the amount of human effort, if any, required to do so. Performance was measured by the slowdown factors relative to the runtime of the uninstrumented programs.

4.4.1 Scalability

Table 1 shows a dozen open-source C and C++ Linux programs on which Kvasir ran successfully, producing valid trace output for Daikon to analyze. Dfec only ran on the four smallest programs.

Dfec usually works without target program modifications on small programs such as `md5` which use few language or library features. Programs of greater size and complexity require source code modifications either so that Dfec will accept them or, more often, so that Dfec's output is a legal program. For all of the programs except `md5`, we needed to add casts to satisfy the stricter C++ type rules. For `bzip2`, Dfec's special treatment of the `void*` type failed to be triggered by a type `BZFILE*` when `BZFILE` was a typedef for `void`. We resolved this by directly replacing `BZFILE` with `void`. For `flex`, we replaced string literals in ternary `?:` operators by variables initialized to those literals in order to resolve an ambiguity related to operator overloading in Dfec's output. For `make`, we spent several hours trying to bypass Dfec's usual pointer transformations to match the memory layout required by Unix environment variables, without success.

In contrast, Kvasir runs on both C and C++ programs of up to 300 KLOC (non-comment non-blank) with rare occasional modifications required by unusual constructs. For `xemacs`, we renamed

						Val-	Mem-	Kvasir
Program	Time	Dfec	Kvasir	Daikon	Comp	grind	check	main()
md5	0.14	310	240	500	260	2.3	15	18
rijndael	0.19	690	5000	2200	3000	7.6	38	86
bzip2	0.18	1100	3500	12000	28000	5.2	28	46
flex	0.41	780	1800	2400	750	14	49	99
Average		720	2600	4300	8000	7.3	33	62

Table 2: Slowdown for programs that were successfully processed by Dfec, Kvasir, and DynComp. The "Comp" column is for the DynComp tool (Section 5). All numbers are slowdown ratios, except that the base runtimes are given in seconds. All tests were run on a 3GHz Pentium-4 with 2GB of RAM.

one of two sets of functions generated by two compilations of a single C source file to avoid having two otherwise indistinguishable functions. For `gcc`, we supplied an extra `--with-gc=simple` configuration parameter to specify a garbage collector that works with the standard `malloc` function.

The majority of the scalability problems of Dfec come from limitations of a source-based approach. In general, larger programs are more likely to contain complex source code constructs and interactions with libraries and system interfaces, which are more difficult to properly handle at the source level than at the binary level.

4.4.2 *Performance*

Both Dfec and Kvasir ran on the order of 1000 times slower than the uninstrumented target program (see Table 2), but most of the overhead was due to writing trace data to files at every program point. The trace files ranged from several hundred megabytes to several gigabytes. For the larger programs of Table 2, `bzip2` and `flex`, we configured both Kvasir and Dfec to skip global variables and to print only the first 100 elements of large arrays.

Dfec was somewhat faster than Kvasir because Dfec produces instrumented source code that can be compiled with optimizations to machine code, while Kvasir performs run-time binary instrumentation with Valgrind, which both consumes run time and also yields less-optimized code (see Section 4.3.3 for details).

The three rightmost columns of Table 2 show the components of Kvasir's slowdown caused by its implementation as a Valgrind tool built on top of Memcheck. The "Valgrind" column shows the $\sim$10$\times$ slowdown of Valgrind running on the target program without any custom instrumentation. The "Memcheck" column shows the $\sim$30$\times$ slowdown of the Memcheck tool. The "Kvasir main()" column shows the $\sim$60$\times$ slowdown of Kvasir when running on the target program but only outputting the trace data for one function, `main()`. This measures the overhead of bookkeeping related to tracing, including the overhead of Fjalar, without the data output slowdown. The rest of Kvasir's slowdown above this factor is caused by the output of trace data for Daikon and is approximately linear in the size of the trace file. Both Dfec and Kvasir share this unavoidable output slowdown.

We believe that we could reduce the value profiling overhead. However, we have not spent any significant effort on optimizing Dfec or Kvasir, because they usually produce trace output faster than Daikon can process it, so neither is the performance bottleneck in the entire invariant detection system. (Daikon's performance is also roughly linear in the size of the trace, but has additional super-linear factors including the number of variables per function [21].) Given its other advantages, Kvasir's performance overhead is completely acceptable.

5. VALUE PARTITIONING CASE STUDY

This section presents a second analysis built using the mixed-level approach and Fjalar toolkit. The analysis determines when two variables hold values that are of the same abstract type; that is, it partitions values according to their abstract types.

5.1 Requirements

C programmers often use a single concrete representation (such as `int`) to hold data of multiple distinct abstract types. For instance, consider the following simple example code:

```
int main() {
  int year = 2005;
  int winterDays = 58;
  int summerDays = 307;
  compute(year, winterDays, summerDays);
}

int compute(int yr, int d1, int d2) {
  if (yr % 4)
    return d1 + d2;
  else
    return d1 + d2 + 1;
}
```

The three variables in `main` all have the same C representation type, `int`, but two of them hold related quantities (numbers of days), as can be determined by the fact that they interact when the program adds them, whereas the other contains a conceptually distinct quantity (a year). The abstract types 'day' and 'year' are both represented as `int`. A richer type system would permit the programmer to give these variables distinct types, which would make the code's intention clearer, prevent errors, and ease understanding. Value partitioning automatically infers a richer set of abstract types from the few representation types that programmers often utilize.

Applications of this analysis include assisting in finding abstract data types, detecting abstraction violations, locating sites of possible references to a value, and aiding other program analysis tools [20]. For example, we have applied the technique as a pre-processing step [10] before dynamic invariant detection (Section 4.1.1), improving both the speed and the precision of the subsequent analysis by indicating variables that need not be compared to one another.

5.2 Source-based static analysis: Lackwit

The Lackwit tool [20] performs a static source code analysis to determine when two variables hold values of the same abstract type. It applies a non-standard type system to the C program, then performs an ML-style polymorphic type inference over that type system; essentially, any two variables whose values may interact via a program operation such as + or = are given the same type. The analysis output does not reflect the underlying polymorphic type system.

Though a static type inference technique is scalable in terms of performance (general type inference has poor worst case behavior, but performs well on programs with limited use of higher-order functions), the Lackwit tool is impractical to use on realistically-sized systems. Library source code is often unavailable or uses difficult-to-analyze constructs such as the definition of malloc, inline assembly, system calls, etc. Thus, hand-written summaries are needed for library functions; while these exist for a subset of the standard library, it would be prohibitive to write them for other libraries. (Lackwit can run without library summaries, as in the case study below, but its results are then incomplete.) Though Lackwit is sound with respect to a large subset of C, this subset does not cover all the features used in real programs: it may miss interactions that result from some kinds of pointer arithmetic, and it does not track control flow through function pointers. On the other hand, Lackwit's conservatism has the danger of producing imprecise results with fewer abstract types than would be correct.

Lackwit also has a number of flaws, related to its implementation as a source-based tool, that limit its usefulness. It does not support all of the language features used in the current Linux standard libraries, does not support queries for function return values, fails to parse certain constructs, and on large programs often fails with a segmentation fault or assertion failure (see Section 5.4).

5.3 Dynamic analysis

We propose a dynamic approach for computing whether two variables hold values of the same abstract type at particular program points, such as procedure entries and exits. We call such variables *comparable* at that program point. (This analysis could work on a wide spectrum of granularity, ranging from measuring abstract types on a program-wide basis to a single-line basis, but we have chosen program-point granularity because interesting variables such as function parameters have program-point scope.) The analysis conceptually computes abstract types for values, then converts the information into sets of comparable variables at each program point (called *comparability sets*). Two values have the same abstract type if they interact by being arguments to the same program operation such as `+` or `=`. This is a transitive notion; in the code `a+b; b+c`, the values of `a` and `c` have the same abstract type. Whether two variables hold values of the same abstract type can change from moment to moment in the program as assignments and other operations are performed. Two variables are comparable at a program point if they held values of the same abstract type during any execution of the program point. This notion need not be transitive.

In addition to solving the scalability problems of Lackwit's static approach, a dynamic approach has the potential to produce more precise results, for two reasons. First, it need not apply approximations of run-time behavior but can observe actual behavior. Second, like typestate analyses [27], it permits the same variable to hold values of different types at different points in the program. For instance, in this code:

```
int apples, oranges = 10, tmp;
apples = oranges;
/* A */
apples = 5;
tmp = apples;
tmp = oranges;
/* B */
```

variables `apples` and `oranges` are comparable (have values with the same abstract type) at A, but are not comparable at B. This precision is important in real code that re-uses temporary variables or registers, and is also a way of achieving context-sensitivity.

As described above, the dynamic type inference consists of performing a value partitioning, then translating that information to variable comparability sets before being communicated to a human or another analysis. We describe a value partitioning analysis at a high level, then discuss three implementation approaches.

The dynamic value analysis maintains, for each value, a tag representing its abstract type. It associates a fresh abstract type with each new value. For a primitive representation type such as `int`, new values are instances of literals, values read from a file, etc. Use of a fresh abstract type for each instance of a literal, and propagation of abstract types along with values through procedure calls, provides perfect context-sensitivity. Only values of primitive types get tags; structs and arrays are treated as collections of primitive types. Each program operation on two values unifies their abstract types, using an efficient union-find data structure, and gives the result the same abstract type.

5.3.1 *Implementation approaches*

This section discusses binary, source, and mixed-level approaches to the variable comparability problem.

It would be impractical to first perform a purely binary-based analysis to record value flow and interaction during execution and then run a source-based postprocessing step. It is not adequate to perform the value analysis and then map the final results (the last abstract values seen) to source information, because previous variable–value associations are lost by that time. In the above example, a binary-based analysis would need to record that the memory location corresponding to `apples` has been overridden with two distinct values so that the source-based postprocessing step can produce the correct results. This is not scalable to larger programs because of the excessive space required to store all values which each variable ever referred to during an execution.

It would also be impractical to utilize a source-based analysis due to the complexities of recording value flow on the source level. Metadata would need to be kept along with each variable and updated at each operation. Tracking the flow and interaction of values in memory and registers is more easily accomplished on the binary level, but it is necessary to update comparability sets of variables throughout the course of the analysis.

Thus, a mixed-level approach is the most practical implementation because it can can simultaneously utilize both binary and source-level information to determine the comparability of variables such as `apples` and `oranges` along with the abstract types of their values at both points A and B during execution without the need to maintain extraneous data.

5.3.2 *Mixed-level approach: DynComp*

We have used the Fjalar mixed-level toolkit to implement a dynamic value partitioning tool called DynComp. It integrates a binary-level analysis to partition values with a language-level analysis to group variables into comparability sets based on abstract types of all values ever stored in those variables.

The binary-level value analysis operates on the whole program; library code is covered transparently. It associates a fresh abstract type with each new value, identified by a unique tag for each byte of a value. This analysis is implemented (similar to Memcheck's V-bits) by keeping a 32-bit integer tag along with every byte in memory and every register. It uses Fjalar's binary rewriting API to insert operations to create tags when new values are created in the program, copy tags around in memory and registers when values are loaded and stored, and unify the sets of tags during relevant operations between values.

The language-level analysis is performed only on the parts of the program for which results are requested; these are the only parts of the executable for which DynComp requires debugging information. It would be impractical to implement this as a postprocessing step because the result depends on the mapping between variables and values which changes throughout execution. For instance, a variable may be used to store values of different abstract types. Thus, the abstract type information that is maintained for values must be integrated with variable information each time a program point is executed. In order to accommodate this, the language-level analysis keeps comparability sets for variables at each program point and merges the value abstract type information into those sets at each execution of the respective program point. This analysis is implemented by using Fjalar's recursive data structure traversal

Program	Lackwit	DynComp
md5	1.85	1.25
rijndael	2.01	2.27
flex	30.91	15.61

Table 3: The average size of the comparability set for a randomly-chosen initialized, non-pointer variable, averaged over all program points executed by DynComp.

API to observe the tags of the values of all relevant variables at each program point (similar to Kvasir, except that it observes the tags, not the actual values). During execution, the variable comparability sets are unified based upon the state of the value partitions (abstract types). At the end of execution, these comparability sets are reported to the user or another analysis tool.

The Fjalar toolkit enabled us to build a fully functional first version of the DynComp tool in a few weeks. Fjalar provided a robust and scalable framework for DynComp, permitting us to focus on algorithm design instead of implementation complexities. We are currently comparing a variety of algorithms for the run-time mapping of value partitions to variable comparability.

5.4 Experimental results

As shown in Table 1, DynComp performs just as well as Kvasir in the scalability tests, mainly due to the fact that they are both built with the Fjalar toolkit. Once we extended Fjalar to work with C++ programs, it did not take much work to extend DynComp to C++. Section 4.4.1 describes the minor changes that were made to `xemacs` and `gcc` to get DynComp to work on it.

The "Comp" column of Table 2 shows the slowdown factor of programs running DynComp. The performance of Lackwit and DynComp are not directly comparable, because the former is a static tool and the latter is dynamic; each is faster in certain circumstances. The main overhead of DynComp comes from the maintenance of tags at the binary level and also the algorithm to update variable comparability sets at each program point.

Table 3 shows that, in general, DynComp generates smaller variable comparability sets because, unlike Lackwit, it does not make approximations about what might be comparable for any possible execution; it reports what is comparable for a particular execution. (The table omits `bzip2` and `civserver` because Lackwit successfully generated databases of abstract types but crashed when we queried the database for the comparability sets of all variables.) In `rijndael`, Lackwit generated smaller comparability sets, but upon manual inspection of the output, we found that its output was erroneous. (We have not yet carefully examined its output for the other programs.) For example, Lackwit reported that two function parameters were not comparable when in fact they were. Although the parameters did not directly interact within that function, the values held by those parameters were copied around to various structs, and these structs were passed by pointer to two other functions and then copied to local variables before the values interacted via a comparison operation. There is no theoretical reason why a source-based static analysis cannot find the correct results in this complex scenario, but implementation complexities of tracking value flow on a source level through layers of structs and pointer indirection make this a difficult task. In contrast, DynComp found the correct results, thanks in part to the relative simplicity of tracking value flow on a binary level.

6. RELATED WORK

Much of the Fjalar infrastructure is devoted to tracking uses of memory; as noted in Section 3.3.2, this is a requirement for a rich dynamic analysis of non-memory-safe programs. We described both a source- and a binary-based approach to the problem. Most memory tracking analysis aim to detect memory errors in C programs.

Representative recent source-based work is by Xu et al. [30], who rewrite C programs to maintain pointer metadata in data structures separate from those of the target program. Although their approach scales up to programs as large as 29 KLOC, it suffers the problems inherent in all source-based approaches: development challenges with parsing C source code, difficulty in supporting additional languages such as C++, and the inability to handle complex language constructs such as integer-to-pointer casts, certain types of struct pointer casts, and the use of custom memory allocation functions. Earlier work includes Safe-C [3], which uses fat pointers to store metadata, and CCured [18], which analyzes the C program to reduce the cost of dynamic checking.

The best-known dynamic memory analysis is Purify [14], which performs ahead-of-time binary instrumentation so that the program maintains bits indicating whether each byte of memory is allocated and initialized, and checking them before uses. Memcheck [23], which we use, is similar but is accurate to the bit level and employs a just-in-time compiler. Many similar tools exist with some or all of the capabilities of these tools; for example, another popular approach is using special system libraries (e.g., `malloc` and `free`).

Binary analysis and editing frameworks include ATOM [25], EEL [16], Etch [22], DynamoRIO [5], and Valgrind [19]. These are low-level tools intended for use in binary transformations that improve performance or security, so they make no accommodation for communicating information to a software engineer, much less in terms of source level constructs. We extended Valgrind to do so.

A debugger designed for interactive use provides some of the same capabilities as the Fjalar toolkit. The debugger can stop execution at arbitrary points in the execution, and print the values of arbitrary source-level expressions. Invalid pointer accesses cause a debugger warning, not a segmentation fault. Some software engineering tools have been built on top of the gdb [26] debugger (for example, [1, 13, 4, 7]). Our own experiments are consonant with previous experience: gdb is not an automated dynamic analysis, nor is it designed for extensibility or serious scripting, and it imposes unacceptably high run-time overheads. Furthermore, it provides no memory tracking, which is a key requirement for a software engineering tool.

The DynComp value partitioning work can be viewed as a limited special case of dynamic slicing. Slicing [28] is a technique for reifying a program's data- and control-dependences; a backward slice of a particular statement or expression indicates all the other statements or expressions whose computation can affect the given one. Static slicing approximates this relation, and dynamic slicing [2, 15, 29] computes it exactly for a given computation. In the general case, dynamic slicing amounts to maintaining a full execution trace of a program, and much dynamic slicing research focuses on how to collect and maintain this trace information efficiently.

7. CONCLUSION

A dynamic analysis for software engineering tasks needs to report its results in terms of the original source program, for use by a human or by another tool. However, a binary analysis has advantages that make it attractive as an implementation technique for a simple, robust tool, especially for memory-unsafe languages such as C and C++. Furthermore, it is often useful to simultaneously have access to both source and binary-level information during an analysis.

Most previous dynamic analysis techniques work at the source level, work at the binary level, or weakly couple the two levels, for instance by postprocessing results to convert them from the binary to the source level. However, certain problems do not fit neatly into one level or the other, or require information from both levels in order to compute precise and useful results.

We propose to apply hybrid source–binary analyses, which we call *mixed-level analyses*, in order to obtain the important advantages of both binary- and source-based approaches, and to obtain results that cannot be practically achieved using either alone. We are not aware of previous research that recognizes the value of integrating source and binary information throughout the duration of a dynamic analysis.

The Fjalar binary instrumentation toolkit allows tools to be built using the mixed-level approach. We evaluated the toolkit in four ways. First, we demonstrated the toolkit's flexibility and language-independence by extending it from C to C++ in few days. Second, we used it as the basis for two tools, one for value profiling and one for value partitioning. These distinct uses indicate that the toolkit is easy to use, general, and robust. Third, we performed a case study of two implementations of value profiling for C and C++. One uses a traditional source-based approach, and one uses the Fjalar toolkit; the latter was both faster to build (even counting toolkit construction time) and much more scalable and robust. We discuss and compare experience with the two implementations, yielding insight into the strengths and weaknesses of the source-based and mixed-level approaches. Fourth, we performed a case study of two implementations of value partitioning (dynamic type inference). Fjalar allowed us to quickly build the DynComp value partitioning tool, which is more scalable and precise than a tool built using a source-based static approach. In sum, the mixed-level approach implemented in Fjalar enables the quick construction of a wide range of dynamic analysis tools, allowing tool builders to focus more on high-level design and less on platform or language-specific implementation details.

Acknowledgments

Jake Cockrell and Ben Morse implemented Dfec. Robert O'Callahan implemented Lackwit and discussed dynamic value partitioning with us. This research was supported by DARPA, EDG, NASA, and NSF.

REFERENCES

[1] H. Agrawal. Towards automatic debugging of computer programs. Technical Report SERC-TR-103-P, Software Engineering Research Center, Purdue University, Sept. 1991.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, White Plains, NY, June 20–22, 1990.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, pages 290–301, June 1994.

[4] R. Biddle, S. Marshall, J. Miller-Williams, and E. Tempero. Reuse of debuggers for visualization of reuse. In *ACM SSR'99*, pages 92–100, May 1999.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *FDDO*, Dec. 2001.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, Mar. 1999. `http://www.jilp.org/vol1/`.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, May 2005.

[8] Edison Design Group. *C++ Front End Internal Documentation*, version 2.28 edition, Mar. 1995. http://www.edg.com.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[10] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[11] Free Software Foundation. GNU binary utilities. `http://www.gnu.org/software/binutils/`.

[12] Free Standards Group. The DWARF debugging standard. `http://dwarf.freestandards.org/`.

[13] M. Golan and D. R. Hanson. DUEL — a very high-level debugging language. In *Winter 1993 USENIX Conference*, pages 107–117, Jan. 1993.

[14] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, Jan. 1992.

[15] J. R. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. Technical Report 1174, University of Wisconsin – Madison, Madison, WI, Aug. 26, 1993.

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, June 1995.

[17] B. Morse. A C/C++ front end for the Daikon dynamic invariant detection system. Master's thesis, MIT Dept. of EECS, Aug. 2002.

[18] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, pages 128–139, Jan. 2002.

[19] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *RV*, July 2003.

[20] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[21] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE*, pages 23–32, Nov. 2004.

[22] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *USENIX Windows NT Workshop*, Aug. 1997.

[23] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX 2005 Technical Conference*, Anaheim, CA, Apr. 2005.

[24] J. Seward, N. Nethercote, J. Fitzhardinge, et al. Valgrind. `http://valgrind.org/`.

[25] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *PLDI*, pages 196–205, June 1994.

[26] R. M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9th edition, 2002.

[27] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, SE-12(1):157–171, Jan. 1986.

[28] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[29] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM TOPLAS*, 17(2):197–216, Mar. 1995.

[30] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE*, pages 117–126, Nov. 2004.