

---

# A Scalable Mixed-Level Framework for Dynamic Analysis of C/C++ Programs

---

Philip J. Guo  
Stephen McCamant

PGBOVINE@CSAIL.MIT.EDU  
SMCC@CSAIL.MIT.EDU

MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA

## 1. Introduction

Many kinds of dynamic program analyses, such as run-time data structure repair (Demskey et al., 2004) and invariant detection (Ernst, 2000), are interested in observing the contents of data structures during execution. There is a large number and variety of programs written in C and C++ which make compelling and practical real-world test subjects for these analyses. However, it is often difficult to create robust and scalable implementations of these analyses because the traditionally-preferred approach of source-code modification suffers from many limitations. Its complement, a binary-based approach, is often inadequate as well because it cannot access source-level information about data structures which these analyses require.

To overcome the limitations of previous approaches, we have built a framework based on a *mixed-level* approach which uses both binary and source-level information at run time. Our framework enables the implementation of dynamic analyses that are *easy to use* — working directly on a program’s binary without the need to deal with source code, *robust* — avoiding output of garbage data and crashes caused by unsafe memory operations, and *scalable* — operating on programs as large as 1 million lines of code.

We have used this framework to build a value profiling tool named Kvasir, which outputs run-time contents of data structures to a trace file for applications such as invariant detection, and compared it against its source-based predecessor to show the advantages of our mixed-level approach.

## 2. Limitations of a Source-Based Approach

A source-based approach works by inserting extra statements into the target program’s source code so that when the instrumented code is compiled and executed, the program runs normally and outputs the desired data, usually to a trace file for further analysis. Here are some limitations:

1. **Usability:** A source-based tool is often difficult to use on real-world programs because a user must instrument every source file and then compile the instrumented source into a binary. It can be time-consuming

to figure out exactly which files to instrument and compile because many real-world programs have their source code spread throughout multiple directories and utilize sophisticated compilation configurations.

2. **Robustness & Scalability:** It is difficult to make a robust and scalable source-based tool for programs written in a memory-unsafe language like C or C++ due to dynamic memory allocation and pointer manipulation. It is impossible to determine run-time properties of dynamically-allocated data at the time the source code is instrumented. For instance, an `int*` pointer may be uninitialized, initialized to point to one integer, or initialized to point to an array of unknown size. In order for a source-based tool to insert in statements to observe the values which this pointer refers to at run time, it must maintain metadata such as initialization state and array size. This can be difficult to correctly implement, as it involves transforming all code related to pointer operations to also update their metadata.

The complexities of parsing C (and especially C++) code presents another barrier to scalability. The parser must properly account for various dialects and standards (e.g., K&R, ANSI) and complex source-code constructs (e.g., function pointers, C++ templates) which are likely to arise in many real-world programs.

Previous research in our group used a source-based approach to implement the Dfec value profiling tool (Morse, 2002). Our experiences with using and debugging Dfec confirm the aforementioned limitations.

## 3. Mixed-Level Approach and Framework

When implementing Kvasir, the successor to Dfec, we adopted a new mixed-level approach which combines binary-level instrumentation with source-level constructs extracted from a program’s debugging information. The framework we developed can be utilized to build a wide range of dynamic analyses, of which Kvasir is the first. It consists of three parts: dynamic binary instrumentation using Valgrind (Nethercote & Seward, 2003), memory safety checking using the Valgrind Memcheck tool,

and data structure traversal using debugging information. These components work together to provide a framework which overcomes limitations of source-based approaches:

1. **Usability:** A user of a tool built upon this framework simply needs to run it on a binary file compiled with debugging information. There are no worries about which source files to instrument or compile. We utilize Valgrind to re-write the target program's binary to insert the appropriate instrumentation instructions at run time. In order to gather source-level information about variable names, types, and locations, we use the debugging information compiled into the binary.
2. **Robustness & Scalability:** In order to ensure that an analysis does not crash the target program or output garbage values, we utilize the Valgrind Memcheck tool to keep run-time metadata about which bytes of memory have been allocated and/or initialized. This metadata is updated during every machine instruction that operates on memory. Memcheck's memory checking is far more robust than the source-based metadata approach because it works directly on the binary level, thus avoiding source-level complexities.

Furthermore, working on the binary level greatly improves robustness and scalability because the semantics of machine instructions are much simpler than that of source code. The source-level description of data has a complex structure in terms of pointers, arrays, and structures, but the machine-level representation is as a flat memory with load and store operations.

When source-level information is required, though, instead of gathering it from the source code, we utilize debugging information, which is much easier to parse and more dialect and language-independent. With such type, size, and location information, our framework allows tools to observe the run-time values of variables, arrays, and recursively traverse inside of structs. Memcheck allows tools to discover array sizes at run time and determine which values are initialized, thus preventing the output of garbage values.

#### 4. Evaluation of Kvasir Value Profiling Tool

Kvasir, built upon our mixed-level framework, works by instrumenting the program's binary to pause execution at function entrances and exits. At those times, it traverses through data structures to output their contents to a trace file. It works on many large C and C++ programs such as `povray`, `perl`, `xemacs`, and `gcc`. Kvasir provides selective tracing functionality to control which variables to observe and at which times the observations should be made, which is useful for tracing selected parts of large programs. For example, a CSAIL research project on data structure

repair (Demskey et al., 2004) used Kvasir to trace the evolution of flight information data structures in the air traffic control program CTAS (1 million lines of code) and a map in the server for the strategy game Freeciv (50,000 lines).

Kvasir has two main limitations: First, Kvasir only works on x86/Linux programs because Valgrind only supports x86/Linux. However, most open-source C/C++ programs can be compiled for this platform. Secondly, programs running under Kvasir suffer a performance slowdown of around 80 times during normal execution without any output. There is an additional slowdown proportional to the amount of trace data outputted. However, by selecting a small subset of functions and variables to trace, it is possible to run Kvasir on large, interactive GUI programs such as CTAS at reasonable levels of speed and responsiveness.

Kvasir's usability, robustness, and scalability far surpass that of its source-based predecessor Dfec, which could only work without crashing on small to mid-sized programs, and usually only after hours of work massaging the source code so that Dfec could accept it as input. The largest program that Dfec worked on was `flex` (12,000 lines), but only after weeks of effort to modify its source code. In contrast, we can download `flex`, compile it with debugging information, and run Kvasir on it, all within half an hour.

#### 5. Future Work

Both our mixed-level framework and the Kvasir tool are under active development, especially to improve C++ support. If you have a research project that requires a tool to observe or modify data structures within large C or C++ programs at run time and would like to use our framework or the Kvasir tool, please contact us. We plan to add new features based on user interests and requirements.

#### References

- Demskey, B., Ernst, M. D., & Rinard, M. (2004). Automatic inference and enforcement of data structure consistency constraints (in preparation).
- Ernst, M. D. (2000). *Dynamically discovering likely program invariants*. Doctoral dissertation, University of Washington Department of Computer Science and Engineering, Seattle, Washington.
- Morse, B. (2002). A C/C++ front end for the Daikon dynamic invariant detection system. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA.
- Nethercote, N., & Seward, J. (2003). Valgrind: A program supervision framework. *Proceedings of the Third Workshop on Runtime Verification*. Boulder, Colorado, USA.