UCRL-CONF-222122

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++

Dan Quinlan, Richard Vuduc, Thomas Panas,
Jochen Haerdtlein, Andreas Saebjoernsen

June 15, 2006

Static Analysis Summit 2006
Gaithersburg, MD, United States
June 29, 2006 through June 29, 2006

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++

Dan Quinlan[1], Richard Vuduc[1], Thomas Panas[1], Jochen Härdtlein[2], and Andreas Sæbjørnsen[3]

[1]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, {dquinlan,richie,panas2}@llnl.gov
[2]Department of Computer Science, University of Erlangen-Nuremberg, Germany, haerdtlein@cs.fau.de
[3]Department of Physics, University of Oslo, Norway, andsebjo@student.matnat.uio.no

Abstract

We present a compact and accurate representation of a whole-program abstract syntax tree, and use it to detect a specific security vulnerability in C++ programs known as a *One-Definition Rule* (ODR) violation. The ODR states that types and functions appearing in multiple compilation units must be defined identically. However, no current compiler can enforce ODR because doing so requires the ability to see the full application source at once; where ODR is violated, the program is incorrect. Moreover, a lack of ODR enforcement makes a program vulnerable to the so-called *VPTR exploit*, in which an object's virtual function table is replaced by malicious code. Our representation of the whole program preserves all features of the source for analysis and transformation, and permits a million-line application to fit entirely in the memory of a workstation with 1 GB of RAM.

1 Introduction

Most whole-program analyses use some form of summarization, at the loss of analysis precision, since analysis time complexity is often super-linear. The traditional unit to analyze and summarize is a procedure since it does not require the compiler to see the full source at once [33]. However, suppose we provide the compiler with a complete view of the entire program. Then, the compiler may freely choose any convenient unit regardless of procedure or module boundaries, and thereby control the size, contents, and context of the program fragment to analyze [19, 35, 24]. Such techniques permit focused and efficient analyses of customizable precision. For software security assurance, improvements in precision raise the level of assurance we can guarantee.

We describe a scalable whole-program analysis that requires the full source to verify a fundamental assumption that all C++ compilers make but no compiler checks. This assumption is the *One Definition Rule* (ODR) [4], which essentially states that a C++ program is only legal if type and function definitions appearing in multiple source files are defined identically (Section 2). Code violating ODR is not legal and may not be translated to a correct executable. Nevertheless, no compiler verifies ODR because each compiles only a subset of an entire program at one time, under *separate compilation*; as it happens, only a whole-program analysis of the full source can be used to verify ODR.

A lack of ODR enforcement enables the *VPTR exploit*, a virtual function table attack [31]. Though not yet widely used, this exploit can be implemented as a simple insider attack, particularly in collaborative or open-source projects [29] (Section 3). Its use is expected to grow as defenses against stack smashing techniques mature [28]. Checking ODR is an essential preventative measure.

We implement basic support for whole-program analysis in the form of a compact and accurate abstract-syntax tree representation of an entire program (Section 4). We can store a million-line application in the memory of a single workstation having 1 GB of RAM without losing any of the information present in the original source. We achieve memory-efficiency for C and C++ programs by merging common declarations (typically appearing in header files) that might otherwise be stored redundantly for each source file. Our representation complements existing whole-program analyses by providing a simple, high-level view of the complete source from which those analyses can be derived.

We are developing this work using ROSE, an open infrastructure for building compiler-based source-

to-source analysis and transformation tools [32] (Section 4). For C and C++, ROSE fully supports all language features, preserves all source information for use in analysis, and permits arbitrarily complex source-level translation via its rewrite system. Although research in the ROSE project emphasizes performance optimization, ROSE contains many of the components common to any compiler infrastructure, and thus supports the development of general source-based analysis and transformation tools. This paper summarizes aspects of ROSE especially relevant to security analysis research (Section 5).

2 One-Definition Rule (ODR)

This section summarizes the essential features of the one-definition rule (ODR). The ODR states that templates, types, functions, and certain entities can only be defined "once," in a sense made precise in the ANSI/ISO C++ Standard [4, Sec. 3.2, pp. 23–24]. Three of the main conditions of the ODR are:

1. Within a single translation unit (a source file and its headers), there may be at most one definition of any variable, function, class type, enumeration type, or template.[1] All compilers verify this condition.

2. Within the entire program, there may only be one definition of every *non-inline* function or object; an inline function must be defined in every translation unit in which it is used, with all such definitions being identical as described in Condition 3 below. Because compilers typically process only one translation unit at a time, the C++ standard does not require that compilers check this condition.

3. Some entities, including class types, enumeration types, inline functions with external linkage, and various template entities, may be defined in more than one translation unit *provided* the definitions are "identical." The C++ standard lays out the meaning of identical precisely; one notable property is that two definitions must "consist of the same sequence of tokens" to be considered the same [4, p. 24]. We use this token-based property in our ODR checker. Like Condition 2 above, compilers typically do not or cannot verify whether multiple definitions are identical as laid out by the C++ standard.

[1] There may, however, be multiple non-defining declarations, such as function prototypes, "extern" variable declarations, forward class declarations.

Listing 1: **main.cc**–A simple program

```
1 int main () {
  extern void runModule (void); // Module to call
3 runModule ();
  return 0;
5 }
```

A legal C++ program must obey the ODR. However, because the standard assumes that a compiler will see only one translation unit at a time (Condition 1), it does not require that a compiler detects violations across translation units.

The linker can partially verify ODR by detecting, for instance, multiple definitions of *non-inline* functions and global variables (Condition 2). However, *inline* function ODR violations cannot be detected; these violations require a whole-program analysis.

3 VPTR Exploit

The *VPTR exploit* replaces an object's virtual function table pointer ("VPTR") with one containing malicious code [31]. The simplest technique redefines the existing definition of an inline virtual function; since a typical compiler does not see the whole program, it cannot enforce the ODR to catch instances of this exploit. This form is most easily implemented as an insider attack, which could occur in a collaborative software development environment as demonstrated by the 2003 Linux kernel backdoor [29]. Moreover, the exploit is an instance of more general *pointer subterfuge* attacks [28].

Listings 1–3 show a program containing the vulnerability. In Listing 1 at line 3, the program executes a routine defined in an external module. That routine creates two stack-allocated objects, `a` and `b`, both of type **Derived**, at line 13 of Listing 3. The **Derived** type inherits from an abstract base class (**Base**), implements the virtual method, **Derived**`::run`, at line 7, and declares a 1-byte datum at line 8. However, because the `run` method is virtual and defined as (implicitly) inline, we must redefine the method in every translation unit in which it is used, albeit with the same definition (see Condition 3 in Section 2). If the compiler cannot enforce this condition, an attacker can re-implement the method in another translation unit to execute arbitrarily different code.

We implement a basic VPTR exploit in Listing 4. This code is a separate module that defines another malicious version of **Derived**`::run()` in lines 6–9. Most compilers, including GCC, assume ODR holds

Listing 2: **Base.hh**–An abstract base class

```
class Base {
public:
  virtual ~Base (void) {}
  virtual void run (void) = 0;
};
```

Listing 3: **Module.cc**–An innocuous module

```
1 #include "Base.hh"

3 // Derived class, intended to be private to this module.
class Derived : public Base {
5 public:
  Derived (void) { buf_[0] = 'a'; }
7 void run (void) { buf_[0] = 'z'; }
  char buf_[1];
9 };

11 // Public interface to this module.
void runModule (void) {
13 Derived a, b; // Two instances on the stack
  Base *pa = &a, *pb = &b;
15 pb->run (); // Expect b.buf_[0] == 'z'
  pa->run (); // Expect a.buf_[0] == 'z'
17 }
```

and simply choose the first one encountered at link-time. That is, when compiling with

```
g++ main.o Module.o ViolateODR.o ...
```

the compiler chooses **Derived**`::run()` from Listing 3, whereas in

```
g++ main.o ViolateODR.o Module.o ...
```

it chooses the implementation from Listing 4. Moreover, if the application uses shared or dynamically-loaded libraries, the malicious module need only appear first in the shared library path to be executed.

VPTR exploits have more sophisticated forms, as shown in Listing 5. This example builds on the basic exploit in Listing 4 by violating ODR and then using buffer-overrun techniques to rewrite the VPTR directly. The first step on line 15 of this alternative **Derived**`::run()` has the same behavior as Listing 3 at line 7, perhaps to make the code appear to behave safely. However, it then executes additional malicious code in lines 16–17.

These additional lines use the fact that a derived object often stores not just its data, but the VPTR appropriate for that object's type. For example, the `a` and `b` stack-allocated instances of **Derived** declared

Listing 4: **ViolateODR.cc**–Basic VPTR exploit

```
1 #include <iostream>
#include "Base.hh"
3
class Derived : public Base { // Class violating ODR
5 public:
  void run (void) {
7   std::cout << "*** Hostile takeover ***"
          << std::endl;
9 }
};
11
Derived d; // Instantiate to get malicious 'Derived'
```

on line 13 of Listing 3 might appear on the stack as shown in the left-half of Figure 1. Each object has its 1-byte datum, `buf_[0]`, plus a hidden 4-byte VPTR. When line 15 of Listing 3 invokes our malicious `run()`, it does so on data allocated and laid out according to the definition of **Derived** in Listing 3. Lines 16–17 of Listing 5 use platform-specific knowledge of how this data is laid out to write beyond the bounds of the data and, in this case, into the VPTR of the next object on the stack, as illustrated in the right-half of Figure 1. The new VPTR is simply the address of a compatible VPTR for the **Attacker** class defined in Listing 5. The **Attacker** class contains another malicious implementation of `run()`. This additional form of the VPTR exploit builds on the ODR violation, so checking ODR helps defend against VPTR exploits generally.

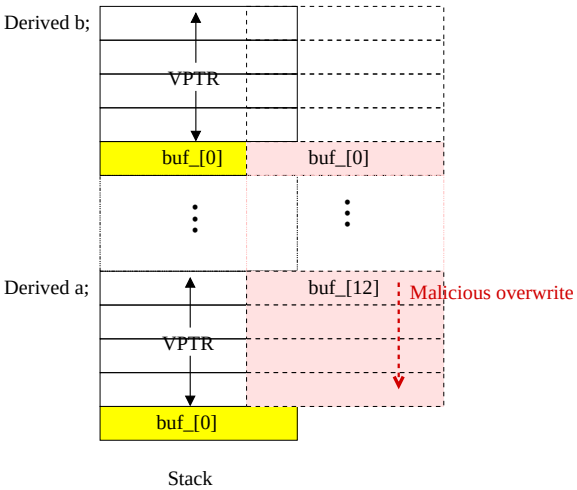

Figure 1: **VPTR exploit**. The attacker implements the alternative version of **Derived**`::run()` shown in Listing 5 such that executing `b.run()` overwrites `a`'s VPTR.

Listing 5: **Attacker.cc**–A malicious module

```
#include <iostream>
#include "Base.hh"

class Attacker : public Base { // More malicious code
public: void run (void) {
    std::cout << "*** vtable overwritten! ***"
              << std::endl;
    // ... Do malicious things here ...
  }
};

class Derived : public Base { // Class violating ODR
public:
  void run (void) {
    buf_[0] = 'z'; // Looks normal, but see below...
    Attacker x; // Instantiate to get a vtable to inject
    *((unsigned *)(buf_+12)) = *((const unsigned *)(&x));
  }
  char buf_[16]; // Buffer used to overwrite vtable
} d; // Instantiate to get malicious 'Derived'
```

4 A Whole-Program Analysis to Detect ODR Violations

Whole-program analysis is typically implemented using procedure summaries or by embedding information into the object files to use whole-program context at link-time. Summarization is necessary to mitigate the impact of super-linear analysis time costs, and procedures are a convenient unit. However, a compiler or analysis tool should be free to analyze any useful, arbitrarily partitioned unit of the program, given a complete and accurate view of program context [35, 36, 24]. This need motivates our whole-program abstract syntax tree representation.

Below, we describe this representation as implemented in ROSE, an open and extensible infrastructure for building customized source-to-source analysis and transformation tools. A typical ROSE-based tool looks like a traditional compiler, with a front-end that generates an object-oriented abstract syntax tree (AST), a "mid-end" performing custom analyses and/or transformations to the AST, and a back-end to unparse the possibly modified AST back into source code. This section outlines recent work to extend the AST to allow the creation of a single, compact AST for the entire program. ODR violations appear during the construction of such a whole-program AST. For more information on the complete ROSE architecture, including features relevant to security analysis, see Section 5.

4.1 Overview of the whole-program representation and ODR test

ROSE's intermediate representation (IR), SAGEIII, stores all high-level information from the source code, sufficient to reproduce the original source code completely. The IR is space-efficient by design since we target large-scale physics applications of 100 KLOC per file and up. Current workstation memory capacities are also quite large (commonly 2–4 GB and greater), and so are better able to support representations of applications consisting of hundreds of files. For greater space savings, we share parts of the AST (subtrees) that are determined to be identical. This test for matching subtrees is where we check ODR, since identical definitions will by construction be shared across multiple files in the AST.

ROSE routinely compiles million-line applications file-by-file. In round numbers, these applications have on the order of 1000 files containing 75K lines contributed from header files and 1K lines of source code in the source file. The effective 76K lines of code generates an AST with about 500K IR nodes. Merging the 75K lines over each of the 1000 files thus saves 75 million lines of code from being represented redundantly in the AST. Using a 250 KLOC program, we have estimated that a million-line application will fit into approximately 400 MB of memory after merging header files. The AST holding the million-line application can also be saved to and loaded from disk using a custom ROSE-specific binary file format; on current single-processor desktop machines, writing one of these binary files to disk takes roughly 30 sec and reading less than a minute. Simple traversals of the whole AST in memory take only a few seconds. Thus, the representation is compact and efficient to operate on once constructed.

We perform the *ODR test* by unparsing candidate subtrees and verifying an exact match. Since ROSE can optionally normalize whitespace and optionally strip comments and preprocessor directives, simple string matching verifies token-by-token equivalence of the original code as required by ODR.

4.2 Whole-program AST construction

Given the ASTs from separate translation units, we merge them as follows:

1. Build an extended *mangled name map*
   The matching process is based on an extended form of name mangling that is common for handling C++ types, variables, and functions. In short, we traverse all declarations in the global scope and all namespace scopes, and for

each declaration, generate and store each declaration's unique name (*i.e.*, extended mangled name) into an STL map. The map's key is the unique name, and its value is a pointer to the associated IR node. (There are a number of details that we omit for simplicity.)

2. Build a *replacement multimap*
   The AST is traversed a second time to match the unique names generated from declarations with keys in the *mangled name map*. All matches are recorded, and a map of pairs of IR node pointers is generated (the IR node of the match and the IR node associated with the matching key from the *mangled name map*). The *ODR test* (see end of Section 4.1) is applied and must pass to be included in the *replacement multimap*.

3. Fixup AST and build the *subtree delete list*
   Using the *replacement mutimap* we traverse the AST again and find all pointers to IR nodes and using the pointer to the IR node as a key we look them up in the *replacement multimap*. If found, we replace the pointer to the key with the pointer to the value obtained from the multimap using the key and the replaced pointer value is added to the *subtree delete list*. All IR nodes that are shared via the merge process are explicitly marked as shared in the AST.

4. Delete redundant subtrees
   To save space we cannot remove redundant subtrees in the modified AST; we iterate over the delete list (which points to redundant subtrees) and remove all the nodes in each subtree.

4.3 Merged AST example

Figure 2 (top) shows the AST for the three source files shown in Listings 1, 3, and 5, with AST subtrees colored by file. The ASTs from the files are not shared. Figure 2 (middle) shows the AST after the merge process, here the diamond shaped IR nodes of the AST indicate that those IR nodes are shared. To be shared, the declaration at the root of the subtrees had to generate the same internal name (in C++ this includes standard name mangling plus a number of other language specific details) *and* the subtrees had to pass the ODR test of equivalence. Figure 2 (bottom) shows the parts of the AST which had the same internal name, but which failed the ODR test. These pairs of subtrees represent the ODR violation that enables a successful VPTR exploit.

5 The ROSE Infrastructure

We are implementing our security analysis work within ROSE, a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure for optimizing large-scale (1 MLOC or more) DOE applications [32]. The ROSE framework enables tool builders who do not necessarily have a compiler background to build their own source-to-source translators. The current ROSE infrastructure can process C and C++ applications, and we are extending it to support Fortran90.

ROSE provides several components to build source code analyzers and source-to-source translators. The C++ front-end generates an object-oriented abstract syntax tree (AST) as an intermediate representation. The AST preserves the high-level C++ language representation so that no information about the structure of the original application (including comments and templates) is lost. This feature permits accurate analysis and the ability to regenerate the original source from the AST. The back-end unparses the AST into source code. The ROSE tool builder creates a "mid-end" to analyze or transform the AST; ROSE assists by providing a number of mid-end components, including graph visualization tools, a predefined traversal mechanism, an attribute evaluation mechanism, transformation operators to restructure the AST, program analysis support, and a number of performance optimizing transformations. ROSE also provides support for annotations whether they be contained in pragmas, comments, or separate annotation files.

Though the traditional emphasis in the ROSE project is on performance optimization, these basic components are well-suited to building software security analysis tools. A recent position paper discusses how ROSE supports the related area of automated program testing and debugging [30].

5.1 Front-end

We use the Edison Design Group C++ front-end (EDG) [13] to parse C and C++ programs. EDG generates an AST and fully evaluates all types. We translate the EDG AST into our own object-oriented AST, SAGEIII, based on Sage II and Sage++ [7]. SAGEIII is used by the mid-end as an intermediate representation. Full template support permits all templates to be instantiated in the AST. The AST passed to the mid-end represents the program and all the included header files. SAGEIII has 240 types of IR nodes, as required to represent the original structure of the application fully.

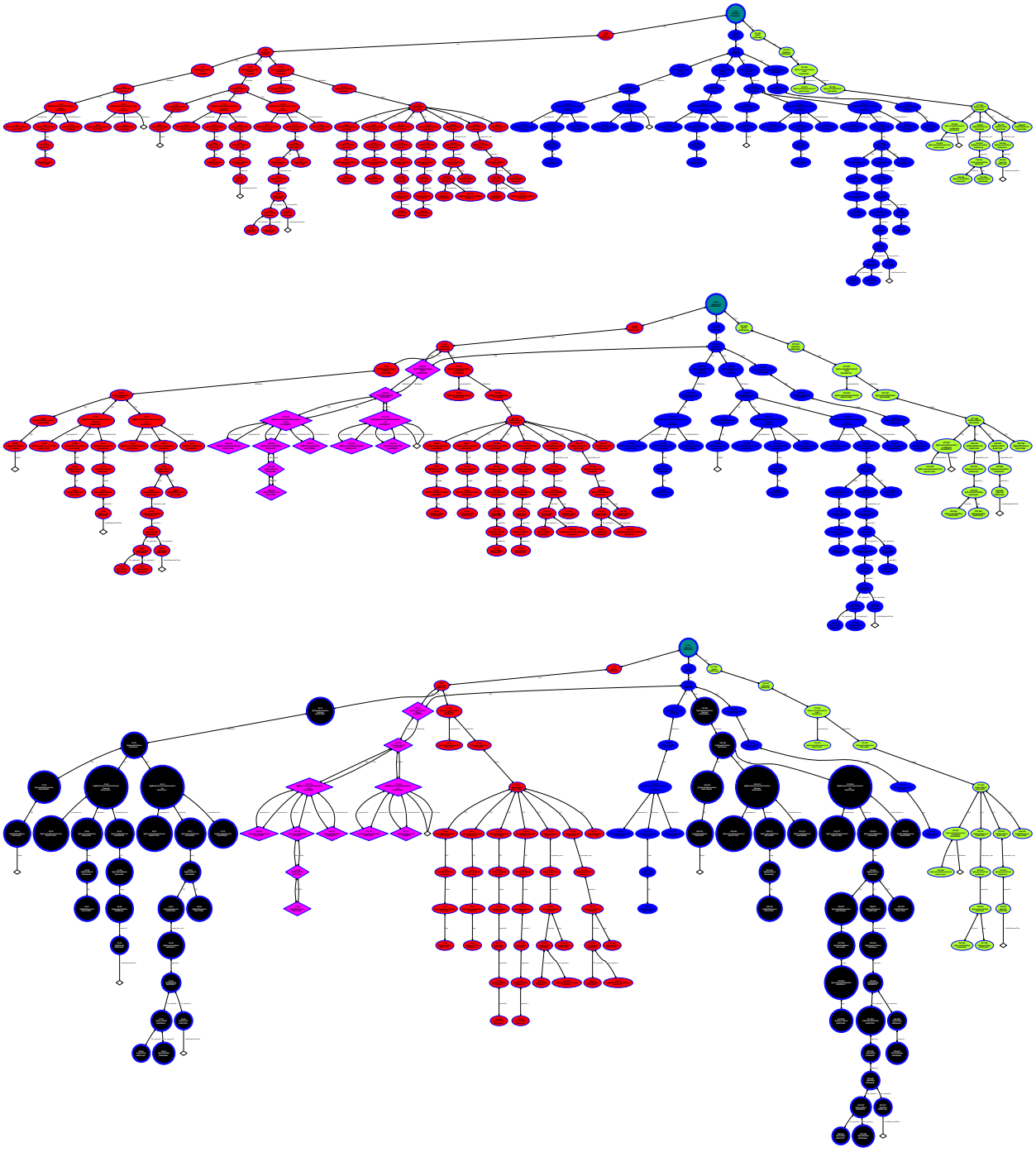

Figure 2: (*Top*) The AST before merging Listings 1 (right-most subtree in light green), 3 (left-most subtree in red), and 5 (middle subtree in blue). (*Middle*) The AST after merging. The **Base** class definition, included by Listings 3 and 5, is shared, as indicated by the magenta subtree with double-edges between diamond-shaped nodes. (*Bottom*) The merged AST, with the two **Derived** class definitions that violate the ODR shown by the subtrees with black circular nodes.

5.2 Mid-end

The mid-end permits analysis and arbitrary restructuring of the AST. Results of program analysis are accessible from AST nodes. The AST processing mechanism computes inherited and synthesized attributes on the AST. An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. Transformation operators can be built using the AST processing mechanism with AST restructuring operations.

ROSE internally implements a number of forms of procedural and inter-procedural analysis, withm uch of this work in current development. ROSE currently includes support for dependence, call graph, and control flow analysis. In collaboration with academic groups, we are extending the analysis infrastructure to interface with general analysis tools, including PAG [2] OpenAnalysis [34], as well as analysis tools specifically for automated debugging and security, such as Osprey for measurement unit validation [22], MOPS for finite state machine-based temporal specification checking [9], and coverage analysis tools [12].

To support whole-program analysis, ROSE has additional mechanisms to store analysis results persistently in a database (*e.g.*, SQLite), to store ASTs in binary files, and to merge multiple ASTs from the compilation of different source files into a single AST (without losing project, file and directory structure).

ROSE also provides debugging facilities, such as AST traversals and coloring, and may be used with visualization tools to aid reverse-engineering [25].

5.3 Back-end

The back-end unparses the AST and generates C++ source code. Either all included (header) files or only source files may be unparsed; this feature is important when transforming user-defined data types, for example, when adding generated methods. Comments are attached to AST nodes (within the ROSE front-end) and unparsed by the back-end. Full template handling is included with any transformed templates output in the generated source code.

6 Related Work

Whole-program analysis has traditionally been applied in performance optimization contexts [5, 35], but has recently also been used to find bugs and detect security flaws using global dataflow analyses [6, 18, 20, 14]. Our techniques complement earlier work by providing the basic infrastructure for accurately representing the source of an entire program, from which we could implement these other analyses. In the case of C++, this representation allows us to verify compliance with ODR, an important but never fully-enforced correctness condition.

Our whole-program AST is closest in spirit to the whole-program control flow graph representation proposed by Triantafyllis, *et al.* [35]. However, we essentially unify the source itself; a whole-program CFG could be easily constructed from this representation.

Atkinson and Griswold advocate on-demand generation of any representations needed for a particular analysis [5]. By contrast, we assume the exponential trends in workstation memory capacity [1] and the need for source-to-source transformation to justify generating and storing the whole-program AST.

A number of compiler infrastructures can or do perform whole-program analyses. GCC developers are adding unified cross-module representations and precompiled header support in order to provide inter-module analysis, particularly for C programs [23, 8]. Our AST merge and file I/O mechanisms are similar in spirit, though we currently provide full support for C and C++, as well as an intermediate representation that accurately represents the source. Among other open C or C++ infrastructures [16, 3, 10] and C++ static analysis infrastructures [37, 17], our complete source-level whole-program representation is unique.

7 Conclusions and Future Work

Our basic support for whole-program analysis enables any number of security analyses with complete context. The analysis we present for checking compliance with ODR to avoid VPTR exploits is just one example; the basic mechanisms permit any number of other global analyses, including whole-program pattern matching [15], region formation [35], and hybrid static/dynamic whole-program path analyses [24], among others. We will develop analyses for additional problems in collaboration with other research groups (*e.g.*, the SAMATE project [26]).

An important issue in software security analysis is how to present analysis results to users [21]. A simple textual representation of security issues is often insufficient because it is difficult to understand the context to the problem under investigation. We are investigating this problem using flexible and unique visualization techniques [27, 25].

We show an example of a program visualization in Figure 3. The program is an 80 KLOC scien-

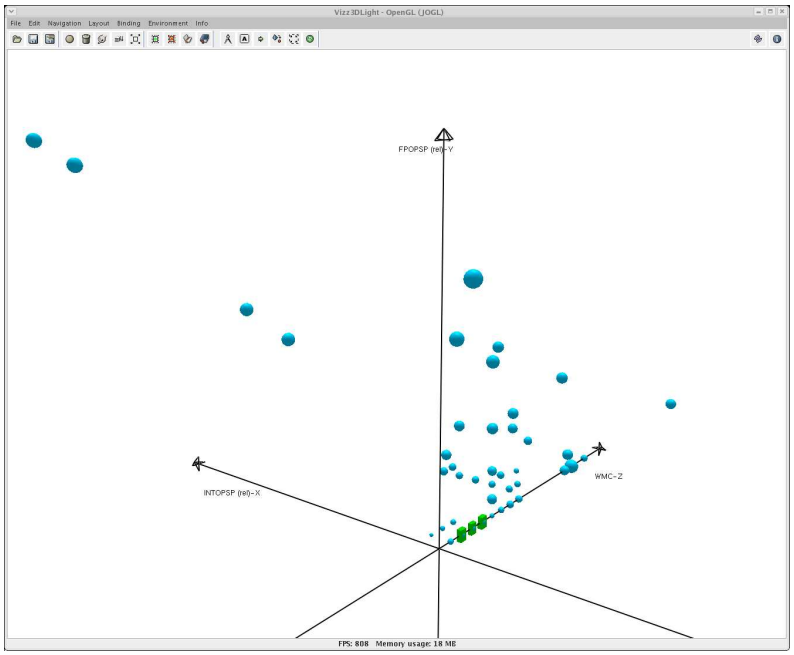

Figure 3: Visualizing security problems in source code.

tific C code, and we plot each function (shown by a sphere) according to its mathematical operations complexity, *i.e.*, the number of floating-point operations along the y-axis and the number of integer operations along the x-axis. The size of each function is equivalent to the relative size of each sphere. Furthermore, the McCabe's Cyclomatic complexity measure [11] is represented on the z-axis.

The application-specific vulnerabilities are shown by green boxes, which indicate possible program overflow problems. These vulnerable functions do not appear along either the x- or y-axis. Thus, we can infer that these vulnerable functions do not occur within the essential scientific kernels, *i.e.*, within functions that make heavy use of floating-point or integer calculations. Indeed, the problem areas for this program occur entirely within the program setup. We are pursuing this and other techniques to help users better understand security analysis results.

References

[1] International Technology Roadmap for Semiconductors, 2005. public.itrs.net.

[2] AbsInt, Inc. PAG: The Program Analysis Generator, 2006. absint.com/pag.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.

[4] ANSI/ISO. *The C++ Standard: Incorporating Technical Corrigendum 1*, volume BS ISO/IEC 14882:2003. John Wiley and Sons, 2nd edition, 2003.

[5] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. International Conference on Software Engineering*, Berlin, Germany, March 1996.

[6] T. A. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. Principles of Programming Languages*, January 2002.

[7] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[8] P. Bothner. GCC compile server. In *Proc. GCC Summit*, 2003.

[9] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2004.

[10] S. Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[11] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[12] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, 2003.

[13] Edison Design Group. EDG front-end. edg.com.

[14] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc.International Conference on Verification, Model Checking, and Abstract Interpretation*, Venice, Italy, 2004.

[15] E. Farchi and B. R. Harrington. Assisting the code review process using simple pattern recognition. In *Proc. IBM Verification Conference*, Haifa, Israel, November 2005.

[16] F. S. Foundation. GNU Compiler Collection, 2005. gcc.gnu.org.

[17] D. Gregor and S. Schupp. Making the usage of STL safe. In J. Gibbons and J. Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming*, volume 243 of IFIP Conference Proceedings, pages 127–140. Kluwer, July 2002.

[18] S. Z. Guyer, E. D. Berger, and C. Lin. Detecting errors with configurable whole-program dataflow analysis. In *Proc. Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[19] R. E. Hank, W. mei W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. November 1995.

[20] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proc. Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004)*, December 2004.

[22] L. Jiang and Z. Su. Osprey: A practical type system for validating the correctness of measurement units in C programs. In *Proc. International Conference on Software Engineering*, Shanghai, China, May 2006.

[23] G. Keating. Inter-module analysis in GCC. In *Proc. GCC Developers' Summit*, Ottowa, Canada, June 2005.

[24] J. R. Larus. Whole program paths. In *Proc. Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, May 1999.

[25] W. Löwe and T. Panas. Rapid construction of software comprehension tools. *Intl. Journal of Software Engineering and Knowledge Engineering: Special Issue on Maturing the Practice of Software Artefacts Comprehension*, 12(54), 2005.

[26] National Institute of Standards and Technology. SAMATE–Software Assurance Metrics and Tool Evaluation, 2006. samate.nist.gov.

[27] T. Panas. *A framework for reverse engineering*. PhD thesis, December 2005.

[28] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, August 2004.

[29] K. Poulsen. Thwarted linux backdoor hints at smarter hacks, November 2003. securityfocus.com/news/7388.

[30] D. Quinlan, S. Ur, and R. Vuduc. An extensible open-source compiler infrastructure for testing. In *Proc. IBM Haifa Verification Conference*, volume LNCS 3875, pages 116–133, Haifa, Israel, November 2005.

[31] Rix. Smashing C++ VPTRs. *Phrack*, May 2000. phrack.org/show.php?p=56&a=8.

[32] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[33] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, pages 189–234. 1981.

[34] M. M. Strout, J. Mellor-Crummey, and P. D. Hovland. Representation-independent program analysis. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 2005.

[35] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proc. Conference on Programming Language Design and Implementation*, Ottowa, Canada, June 2006.

[36] T. Way, B. Breech, and L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *Proc. Conference on Parallel Architectures and Compilation Techniques*, pages 24–33, Philadelphia, PA, USA, September 2000.

[37] D. Wilkerson. OINK: A collection of composable C++ static analysis tools, 2005. freshmeat.net/projects/oink.

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. w-7405-eng-48