Shared and Distributed Memory Parallel Security Analysis of Large-Scale Source Code and Binary Applications*

Dan Quinlan[1], Gergo Barany[2], and Thomas Panas[1]

[1]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory,
{dquinlan,panas}@llnl.gov
[2]Institute of Computer Languages, Vienna University of Technology, Austria,
gergo@complang.tuwien.ac.at

Abstract

Many forms of security analysis on large scale applications can be substantially automated but the size and complexity can exceed the time and memory available on conventional desktop computers. Most commercial tools are understandably focused on such conventional desktop resources. This paper presents research work on the parallelization of security analysis of both source code and binaries within our Compass tool, which is implemented using the ROSE source-to-source open compiler infrastructure. We have focused on both shared and distributed memory parallelization of the evaluation of rules implemented as checkers for a wide range of secure programming rules, applicable to desktop machines, networks of workstations and dedicated clusters. While Compass as a tool focuses on source code analysis and reports violations of an extensible set of rules, the binary analysis work uses the exact same infrastructure but is less well developed into an equivalent final tool.

1 Introduction

The development of security analysis for software can be expected to address larger software projects at an ever increasing depth of detail in the future. These two factors contribute to the growing importance of performance within security analysis. Additionally, the development of more commonly available open source techniques for software analysis is likely to lead to much more software being routinely evaluated for security. The complete automation of software security analysis is still largely illusive, as people are often required to double check reported problems and automated corrections to security flaws are not yet technically possible. However, many specific security rules can be completely automated (e.g. change known insecure library calls to secure ones, like `sprintf` to `snprintf`; or change a `delete var` operation to `delete[] var`, iff `var` represents an array). The techniques for security analysis can range from efficient structural tests on the AST to particularly expensive formal validation techniques (typically considered intractable for large scale applications of many millions of lines of code).

Source code analysis supporting common compiler optimization is typically compromised by the performance requirement of the compilation phase itself, even so rather expensive algorithms may be justified. Commercial settings for security analysis are typically for desktop platforms and represent constrained resources. For security analysis, even rather expensive forms of program analysis (in time and resources - e.g. memory) may be justified. Moreover, the program analysis techniques required for many phases of binary analysis (e.g. alias analysis) can be of particularly high order of complexity. Historically, similar complexity has appeared in numerical methods for the solution of partial differential equations within scientific computing, and has lead the drive for both better algorithms and more advanced parallel computing architectures. Much work specific to numerical algorithms and performance optimization for scientific computing continues to be done at laboratories worldwide. This work has some relevance to large scale security analysis as well.

There are thousands of security rules that could be imagined for software generally. Hundreds are language specific and addressed in a number of existing

*This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

rule sets (SAMATE [9], CWE [8],CERT Secure Code rules [2], QA++ [10], etc). However, many possible rules are domain specific and fundamentally tied to the semantics of individual libraries or are specific to internal details of individual applications.

The correct use of the libraries is critical to their safe use and the compiler itself is only cognizant of the language semantics, not the library semantics. Applications themselves have specifications for how functions and internal data structures are expected to be used; security rules could be extended to even this level of application specific behavior. The incorrect use of both the language and libraries is a common source of security flaws, even within strictly legal code. Note also that numerous issues about even the legality of code (e.g. the one time definition rule in C/C++ [11]) are not checked by any compiler and assumed to be true in large scale applications. Since the encapsulation of all relevant security rules is intractable, we have focused on an extensible solution that can be easily tailored to specific environments, libraries, applications and computer architectures. Within this context the development of simple rules is easy to specify and an evaluation of the rules can be combined to form general or even highly specialized checkers. We address the performance issues of large numbers of rules to check on large scale applications by using both shared and distributed memory parallel computing.

This paper presents recent work on the parallelization of static analysis for easily specified source code patterns. The handling of binaries is similar since the internal tools and infrastructure is identical, but the rule sets for binaries are less well developed than for source code. We have implemented our work within the Compass/ROSE infrastructure. A specific goal of this work has been to layout how more sophisticated rules, which might require more expensive program analysis, would be defined and how they might be leveraged in large scale applications consisting of many millions of lines of code.

2 ROSE

We are implementing our parallel program evaluation within ROSE [12], a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure, which currently can process million line C, C++ and Fortran 2003[1] applications [12].

ROSE permits an attribute grammar based traversal of the AST with trivial attribute evaluation which supports both a simple programming model for users to specify security flaws and more importantly *parallel evaluation*.

For C and C++, ROSE uses the Edison Design Group C++ front-end (EDG) [6] to parse programs. EDG generates an abstract syntax tree (AST) and fully evaluates all types. Translated from the EDG AST, ROSE uses for its internal representation (IR) its own object-oriented AST, SAGEIII, which consists of 240 types of nodes for C and C++. For Fortran 2003 support, ROSE uses the Open Fortran Parser (from Los Alamos National Laboratory), and generates a similar object-oriented AST using many of the same IR nodes as for C and C++, but adding new IR nodes specific to Fortran 90 and Fortran 2003 support.

The 240 types of nodes preserve the high-level C, C++, and Fortran language representation so that no information about the structure of the original application (including comments and templates) is lost, thereby allowing ROSE to perform *sophisticated analysis* as well as *transformations* on the IR.

2.1 Source Code Analyzer

As an analysis infrastructure, ROSE can perform a wide range of analyses of which many are implemented in our Compass tool, cf. Section 3. Analyses may vary from general purpose analyses (that are part of ROSE), such as control flow analysis, data flow analysis, program slicing, etc. to domain-specific analyses (which are part of Compass), such as software quality analysis (e.g. memory issues, pointer issues, expression issues, etc.), data handling analysis (e.g. buffer errors, type errors, numeric errors, etc.), etc., cf. [8].

Tool builders, who do not necessarily have a compiler background, may utilize the Compass/ROSE infrastructure to build their own analysis tools.

In addition to the availability of general purpose analysis within ROSE and specific domain analysis within Compass/ROSE, we are extending the ROSE infrastructure, in collaboration with academic groups, to interface with general analysis tools, including PAG [1], OpenAnalysis [13], as well as analysis tools specifically for automated debugging and security, such as Osprey for measurement unit validation [7], MOPS for finite state machine-based temporal specification checking [3], and coverage analysis tools [5].

2.2 Source-to-Source Translator

The main purpose of the source-to-source capability of ROSE is to allow million line high performance DOE applications to be optimized to run even faster.

[1]Fortran 2003 is currently under development.

For that, algorithms exist within ROSE that either improve the efficiency of source code statically, e.g. loop optimizers, or determine the fastest local code optimization based on a combination of run-time information and static decisions. The latter determines from a permutation of static transformations the best option for a specific machine setting.

Another capability of the source-to-source mechanism that we are currently working on, is to patch faulty implementations. For this, ROSE analyses and translators are applied to automatically detect and fix quality as well as security issues in source code. As a result, the differences in the IR before and after the applied transformations can be released as a source code patch. In this way, the source-to-source translator helps to improve the current implementation.

2.3 IR Generation

The IR in ROSE is easy to extend since it utilizes *Rosetta*, a IR node generator, which is part of ROSE. Rosetta allows to generate the implementation for all necessary IR nodes with all their access functions based on a grammar specification for each IR. In this way, we have recently extended ROSE with 38 Fortran specific IR nodes[2] and also 151 nodes representing the 80x86 instruction set.

The Fortran IR nodes were added to support ongoing work with Los Alamos National Laboratory and Rice University to uniformly handle Fortran in ROSE similar to existing support for C and C++. The 80x86 instruction set, on the other hand, was added as part of internal research to represent binaries in form of an AST, and to allow general analysis on binaries uniformly with source code.

2.4 Binary Analysis

Security analysis cannot be performed on only source code, especially when source code is not available or if compilers are not trusted. For instance, compilers may perform optimizations that remove essential features of a source code implementation. One such operation is known as *insecure compiler optimization* [8]. The authors of this *common weakness evaluation* (CWE) rule present an example where a developer tries to clear a buffer containing a password. Unfortunately, because the buffer is no longer used after it is purposely cleared by the developer, the compiler removes this operation due to machine code optimization. As a result, the buffer stays uncleared in memory and is subject to exploitation.

Detecting an insecure compiler optimization is not trivial and is currently subject to manual expert investigations only. It is difficult to predict the translation of source code to machine code that a compiler does; especially using different compiler optimization flags, compiler versions, operative systems and hardware platforms. Currently, there exist no tools that could automatically determine whether a compiled code matches its source code - not semantically and not even syntactically, i.e. it is not possible to automatically determine whether all operations in the source code are present in the binary (and vice versa).

One attempt to approach the above problem, i.e. to investigate and relate the input (source code) and output (binary) of a compiler, would be to create a machine readable model for both, input and output, and try to compare the models. This is one of the reasons why the ROSE IR node set was extended with an IR node set for 80x86 assembly. This extension enables us to represent a binary in exactly the same form as source code, i.e., in an intermediate representation, which uses the same traversal mechanisms for both source code and binary.

Our binary AST is represented by functions and their corresponding instructions. Each instruction contains operands that are attached in form of an expression tree, just like with the source AST. To match binary instructions with source code statements, one would need to traverse both the source IR and binary IR. This infrastructure enables matching source and binary, however, it does not simplify the hard task of actually matching corresponding nodes from both intermediate representations. The difficulty here is that it is inherent in a compiler's nature to optimize and rearrange instructions (even without additional optimizations enabled) to produce an efficient binary during translation. The rearrangements may be complex enough to make *exact* pattern matching hard.

Nevertheless, matching may be simplified. For instance, in our initial attempts, we have focused on only matching functions between source code and binary. This approach can be used to support a user to answer the question of insecure compiler optimizations, as described above. All that is needed is a combined binary and source code checker that looks for all function calls in the binary as well as in the source code and compares the two sets. Referring to our insecure compiler optimization example, such an analysis would return that the binary is missing a *memset* library call that is purposely present in the

[2] With many IR nodes overlapping with specifications from C and C++.

source code.

Our work on binary analysis is based on *IDA*, an interactive disassembler [4] and includes a number of external collaborations. We utilize this tool in order to parse and pre-analyze binaries. All information retrieved by IDA is stored via Python scripts to an MySQL database, from which our binary AST generator gathers its input. Once the AST of a binary is generated, arbitrary analyses can be performed.

3 Compass

Compass is a source code analysis tool built using ROSE to support the detection of violations of externally easily defined rules in source code. Our focus has been on secure coding rules. Where the rules can be defined on the AST, the rules are simple to specify using the language grammar for C, C++, and even for Fortran. Each rule is specified separately, and thus the rules can be evaluated independently. More complex rules may be defined on either the control flow graph (CFG), system dependence graph (SDG), call graph, class hierarchy graph or combinations of those.

Compass is foremost an extensible open source infrastructure for the development of large collections of rules. A script within Compass (`gen_checker.sh`) automates the generation of source code templates for writing Compass rules; including a Makefile (for Linux), documentation templates, and related files required to define and test new rules. Typically, the code template is instantiated by the definition of a single function defining a user specific security rule. A separate script (`compass_checkin.sh`) submits the new rule, contained in a directory, as a tarball to a Compass repository. Finally, a third script (`compass_submission_setup.sh`) collects all the rules in the Compass repository and inserts them into a pre-existing Compass/ROSE distribution. Steps required to address the scale of hundreds of checkers are completely automated.

In collaboration, Gabriel Coutinho, of Imperial College London, developed the *QRose library* to build a GUI interface for Compass that permits a more conventional look and feel. QRose is a Qt-based GUI library supporting important ROSE abstractions. QRose makes it easy to build ROSE based tools that have full knowledge of source code, ASTs, IR node queries, etc. Compass is an example ROSE project within the ROSE/projects directory of the ROSE distribution.

While most rules are defined on the AST, each may access the control flow graph or other program analysis results as required to express the evaluation of violations of defined rules.

4 Performance Optimization

4.1 Combining checker evaluations into a single AST traversal

The organization of checkers as separately evaluated read only traversals of the AST permits a simple setting for the evaluation of different approaches to parallelization for large scale applications. The first step in the optimization is to organize the *combined* execution of the separate checkers efficiently within one single AST traversal. The effect is a loop fusion over the known semantically independent checkers. Figure 1 demonstrates the performance improvements we obtained in this way; in our particular application the execution time was roughly halved. The subsequent step is then to parallelize the evaluation of the checkers on each IR node.

4.2 Parallelization of Compass

The parallelization of static analysis is the key to achieving good performance in highly complex analysis of very large programs. As desktop systems are increasingly equipped with multiple CPU cores, program analyzers aiming at the best possible use of the available hardware must also become parallel.

Two different approaches at parallelization of Compass have been implemented: The first one uses the *pthreads* library in a shared memory setting, running different groups of checkers on a shared AST; the second approach uses distributed memory and the *MPI* library, running the same checkers on different processors on different distributed parts of the AST. The implementation of our parallelization workis not specific to Compass; it is part of the ROSE infrastructure and usable for general program analyses. Figure 2 shows results of the improvements in execution time by parallelizing Compass using the two methods.

In the shared memory approach a single copy of the AST is present in memory and is traversed by several threads at a time. Each thread executes different checkers on the AST; as they all run in the same address space, no special effort is necessary to combine the final results of the computation. The speedups achieved using this method are mostly limited by the available memory bandwidth as the work of Compass checkers consists mainly

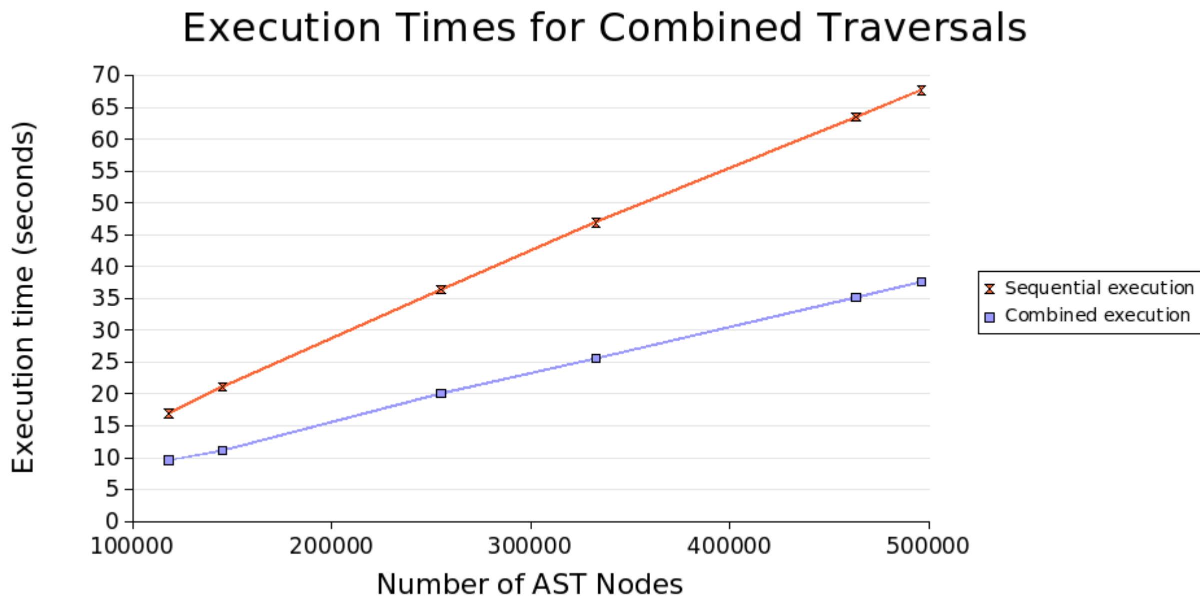

Figure 1: This figure shows the execution times of 58 Compass checkers of various complexities on a collection of programs (where the x axis refers to program size). Sequential execution refers to one traversal of the whole AST for each checker, i.e. 58 traversals; combined execution traverses the AST only once for all checkers. A program with 500K IR nodes corresponds to about a 100K line application in C.

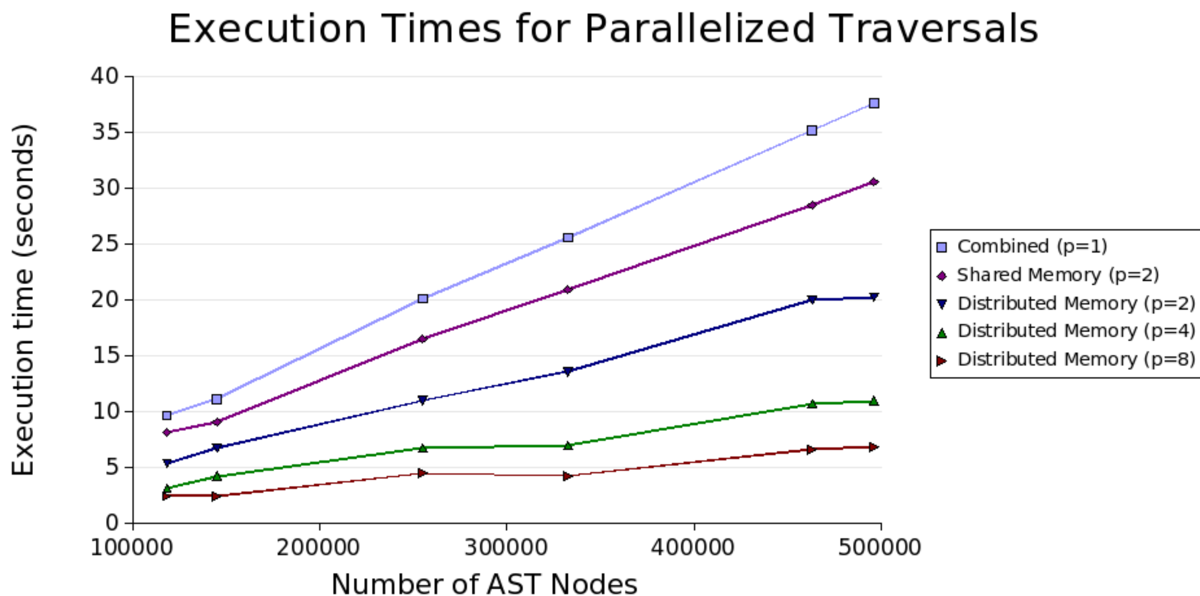

Figure 2: Illustrated is the difference in performance of using different numbers of processors for the shared and distributed memory parallel execution of Compass. For this, we have used the same checkers and input programs as in Figure 1.

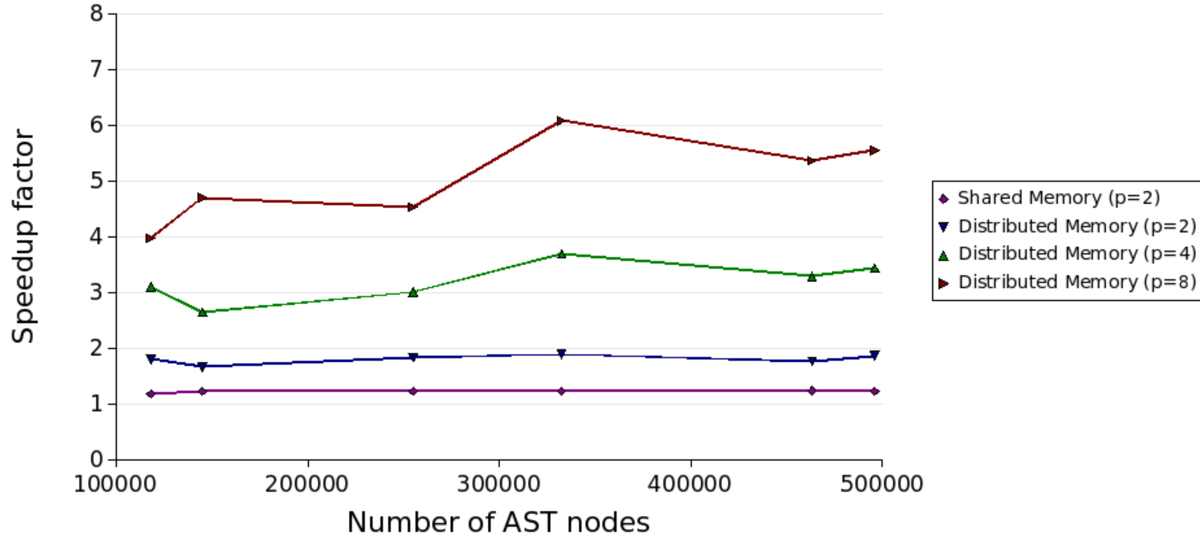

Figure 3: A common metric in the evaluation of parallel computing is *parallel speedup*, defined as the ratio of single-processor execution time to parallel execution time. The optimum speedup factor would equal the number of available processors. This graph shows the speedups of various parallelized versions of Compass checkers on our evaluated programs. The image reveals that the distributed memory approach scales well with the number of processors while the shared memory model is limited by memory bandwidth.

of memory accesses, many of them non-local. Using two processors, shared memory parallelization of Compass yields a speedup of about 20%, depending on the characteristics of the hardware; experiments with unrealistically CPU-intensive applications have shown much better results. As the limiting factor in the speedup is the memory, using more than two CPUs has not proven fruitful since it causes even more contention for memory.

The distributed memory version of Compass runs the same analyses on different parts of the AST that reside in separate address spaces. Currently each process has a copy of the complete AST, but future work will investigate how to partition the AST better, i.e. each process only stores IR nodes necessary for its part of the overall analysis. In our initial work, each process is assigned a number of files to analyze from the global AST. This decomposition of the AST is done in a way such that each process has roughly the same number of IR nodes to analyze in order to balance the loads between processors. This is a somewhat coarse approach and limits the speedups we achieved; for instance, running on 8 nodes yields speedup factors of about 5 to 6 where perfect distribution of the AST would yield the theoretical upper bound of 8, cf. Figure 3 and Figure 4.

For this reason we will in the future partition the AST into function definitions rather than just files, yielding a more fine-grained distribution for better parallel performance.

While read-only analyses are usually usable in shared memory parallelization without any modification, in the distributed memory setting the results of the computations from different nodes must be collected and combined in a single process. For this reason distributed memory analyzers must implement a simple interface consisting of a function for serializing their relevant analysis data and a function for combining such serialized data from distributed sources. The data are then communicated using the MPI library, but this detail is completely hidden from the user.

Our two parallelization approaches can easily (without any additional effort) be combined, provided that a cluster of multicore systems is available. In this case the AST is partitioned as in the distributed memory setting, and the checkers in each process are then run in shared-memory threads on its part of the AST. Preliminary experiments suggest that this combination yields roughly the expected additional 20% speedup compared to the pure distributed memory approach.

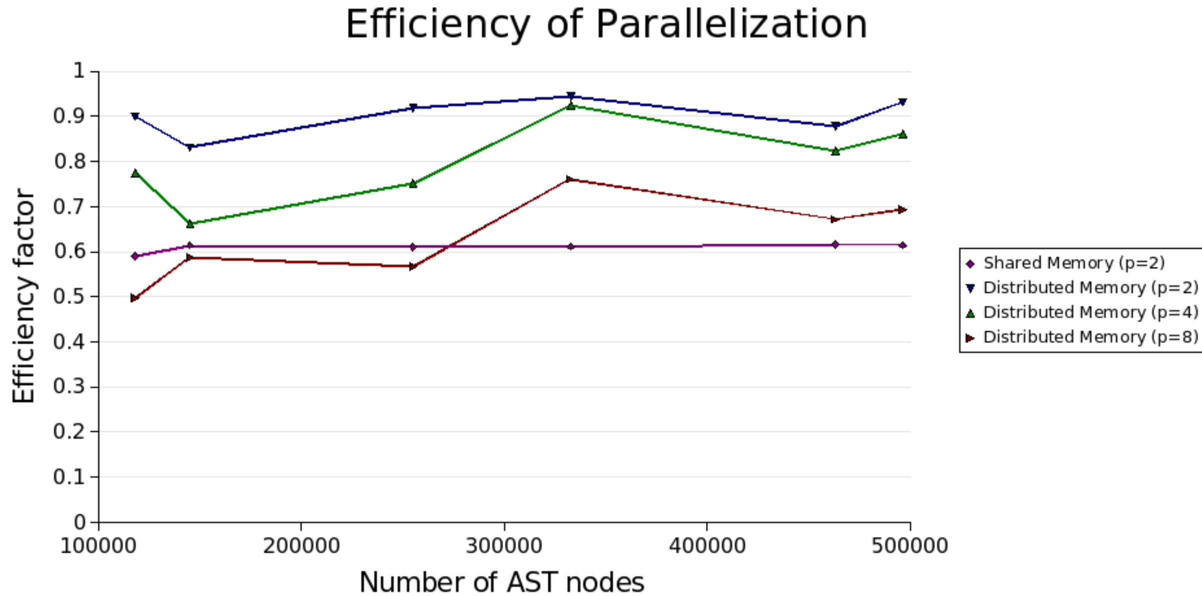

Figure 4: *Parallel efficiency* is a metric for the evaluation of parallel performance results. It is defined as speedup divided by the number of available processors; perfect parallel efficiency would be a constant value of 1.0, corresponding to perfect utilization of available resources. This graph shows the efficiency of various parallelized versions of Compass.

5 Related Work

We are not aware of distributed memory parallel research work for security analysis. When parallel computing was attractive as a new research area (late 1980s) there were a number of papers on general program analysis for shared memory architectures, but it does not appear to have been successful and the emphasis on distributed memory appears to have made the subject overly complex. It can be expected that shared memory parallel work will be attractive with the recent introduction of multi-core processors. This may become attractive and easy to introduce in commercial security analysis tools.

Clearly there are a number of sophisticated security analysis tools focused on moderately large-scale applications. Using function/file summaries many tools can address large scale applications, working on one file at a time. The use of such summaries is however a possible problem for extensible techniques that target user defined rules or are built from accumulated rules built by external groups. Most commercial tools come with a large set of predefined rules and so can tailor their summary information to handle their own rule sets. Except that we know that most commercial tools are extensible, we are unfamiliar with the details or the level of the extensibility provided by commercial tools because we have not had access to the better (and expensive) commercial tools. We expect that all the commercial tools address performance using combined traversals of the AST, but we are unaware of any using parallelism for the analysis (shared or distributed).

6 Conclusions and Future Work

The development of an extensible static analysis tool, build on top of the ROSE source-to-source open compiler infrastructure, has been presented. Compass has provided a useful tool for the analysis of source code and an initial architecture for a future binary analysis tool. We have also presented results on the parallelization of the detection of violations of a moderate collection of rules. Results have been presented for both shared memory (multi-threaded) parallelism and distributed memory parallelism. Importantly, the individual checkers were not modified to accomplish the combined evaluation over a single AST traversal or the shared or distributed memory parallelism.

Techniques developed to support global analysis [11] in ROSE permit a single AST to be formed from the merging of ASTs from separately compiled

files. The resulting whole project AST is memory efficient since it shares IR nodes where possible and results in a memory footprint of about 400MB per million lines of code. This is efficient enough to permit many-million line applications to be held in memory for simple processing. We will in future work use this approach to decompose an AST representing an entire project for use on large multi-processor distributed memory machines (LLNL has parallel machines with tens of thousands of processors). Future work will evaluate the techniques for the fine precision decomposition of the AST and the performance of parallel security analysis on much larger distributed memory computer architectures.

References

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

[2] CERT. Secure Coding Standards, 2007. www.securecoding.cert.org/confluence/.

[3] 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.

[4] DATARESCUE. IDA - Interactive Disassembler, 2007. www.datarescue.com.

[5] 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.

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

[7] 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.

[8] MITRE Corporation. Common Weakness Enumeration, 2007. cwe.mitre.org.

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

[10] Programming Research Group. High-Integrity C++ Coding Standard Manual, 2004. www.programmingresearch.com.

[11] D. Quinlan, R. Vuduc, T. Panas, J. Härdtlein, and A. Sæbjørnsen. Support for whole-program analysis and verification of the One-Definition Rule in C++. In *Proc. Static Analysis Summit*, Gaithersburg, MD, USA, June 2006. National Institute of Standards and Technology Special Publication.

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

[13] 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.