IJARSE
ISSN 2319 - 8354

# SAAC-SECURITY PRESERVING ARTIFICIAL INTELLEGENT AUTOTUNED COMPILER- A Survey

**A.Yaganteeswarudu[1], P.Surekha[2]**

[1]*Computer Science and Engineering Sreenidhi Institute of Science & Tech (Autonomous) Hyderabad, (India)*

[2]*Computer Science and Engineering Dr.MVSIT Mangalore (India)*

## ABSTRACT

*Compilers are generally supposed to make your code as efficient as possible – while compilation finding errors and converting to executable code.Todays most important challenging feature of programming is security. Programmers need to provide more security to the programmes developed. And while it may be well-established behavior, there is always danger when the behavior of code is opaque to the coder. Programmes need to be self improvement which is the facility provided by artificial intelligence. Analyzing the equal implementation details for performance is the ongoing research area.*

*Closer to the compiler are development frameworks, some of which have begun to make safe behavior the default. Frameworks like Django and Grails, for instance, will provide some protection against XSS and CSRF attacks on the front end, or SQL injection attacks on the back end. Autotuning compiler measure execution time and compare and select the best-performing implementation. By using recursive self improvement technique of the artificial intelligence concept it should improve the design of constituent software's and hardware's.*

***Keywords: Autotuning; Artificial Intelligence; Stak Buffer; Deterministic Finite Automata;***

## I. INTRODUCTION

The challenge with frameworks and compilers is that the lower you go in the technology stack, the less context you have to make decisions. When you write code, you know to a large degree who will be using it and what they will be trying to do. You can put security logic in exactly the appropriate places based on the use cases.

Current architectures and compilers continue to evolve bringing higher performance, lower power and smaller size while attempting to keep time to market as short as possible. Typical systems may now have multiple heterogeneous reconfigurable cores and a great number of compiler optimizations available, making manual compiler tuning increasingly infeasible. Furthermore, static compilers often fail to produce high-quality code due to a simplistic model of the underlying hardware.

Compilers may be used to compile themselves. As compilers are more optimized they can recursively recompile themselves by using artificial intelligence and so be faster compiling.

Autotuning is related to hardware (and hardware-software) design space exploration. The process of analyzing various functionally equivalent implementations to identify the one that best meets objectives. Many codes spend the bulk of their computation time performing very common operations. Autotuning is used to enhance performance without requiring low-level programming of the application.

In this paper iam proposing the concept of producing compiler with the features security, artificial intelligence concept self recursive method and Autotuning.

## II. SECURITY PRESERVING

The following may be the attcks an attacker can do  Buffer overrun vulnerabilities

a. Stack-based: Stack-smashing attack
b. Heap-based: Function pointers, C++ virtual pointers, Exception handlers (CodeRed)

a. Smashing the Stack

- To overflow (automatic) stack buffer, one would need:
  - Shellcode, i.e. characters representing machine code (obtain from gdb, as)
  - Memory location of injected shellcode (typically buffer address)
- Can approximate to make up for lack of precise information
  - nop instructions at the beginning of the shellcode
  - overwrite locations around 0(%ebp)with shellcode address
- suid installed programs. Shellcode: shell, export xterm display

b. Heap-Based Attacks

- Function pointer
  - Higher address: function pointer
  - Lower address: buffer
- C++ Pointer to vtable
  - Higher address: virtual pointer
  - Lower address: buffer

A compiler doesn’t know whether a function call involved direct user input, and could potentially introduce huge slow-downs by attempting to, say, check for buffer overruns everywhere; but if it could figure out or be told the right context it could provide an additional layer of security without any developer interaction. As security awareness grows across the industry, the basic tools and infrastructure required to produce code will need to remove the risk of damaging side-effects in optimization.  As they do so, let’s hope they take the opportunity to explore solving the difficult problems of security.

Compiler-assisted securing of programs at runtime Via added runtime checks as part of function invocations and add protection code such that protect what: control data in stack frames , What from: most stack-smashing attacks.

## III. ARTIFICIAL INTELLIGENT COMPILER

Recursive self-improvement is the speculative ability of a strong artificial intelligence computer program to program its own software, recursively. This is sometimes also referred to as *Seed AI* because if an AI were created with engineering capabilities that matched or surpassed those of its human creators, it would have the potential to autonomously improve the design of its constituent software and hardware. Having undergone these

IJARSE
ISSN 2319 - 8354

improvements, it would then be better able to find ways of optimizing its structure and improving its abilities further. It is speculated that over many iterations, such an AI would far surpass human cognitive abilities. A limited example is that program language compilers are often used to compile themselves. As compilers become more optimized, they can re-compile themselves and so be faster at compiling. However, they cannot then produce faster code and so this can only provide a very limited one step self-improvement. Existing optimizers can transform code into a functionally equivalent, more efficient form, but cannot identify the intent of an algorithm and rewrite it for more effective *results*. The optimized version of a given compiler may compile faster, but it cannot compile *better*. That is, an optimized version of a compiler will never spot new optimization tricks that earlier versions failed to see or innovate new ways of improving its own program. Seed AI must be able to understand the purpose behind the various elements of its design, and design entirely new modules that will make it genuinely more intelligent and more effective in fulfilling its purpose.

"Theory of computation" it is a branch of mathematics which tells that the given problem can be solved or not and if yes then which algorithm will give best result. Automata Theory is a branch of Theory of Computation which tells us about the machine (theoretical model of computer) and their automaton. Since compilers based on the grammar of the language which it takes as input, compiler having number of phases around 6 phases , first one is lexical analysis which uses finitie automata , second phase is syntax analysis which uses parser like LALR etc .

Now come to Artificial Intelligence , cellular automata is an algorithm of Bio Inspired AI is an application of Automata Theory.

Elevator works on principle of Deterministic Finite automata.To reach particular floor

## IV. AUTOTUNING

Autotuning is related to hardware (and hardware-software) design space exploration

The process of analyzing various functionally equivalent implementations to identify the one that best meets objectives.

*Compiler-based Autotuning*

Parameters and variants arise from compiler optimizations

Parameters such as tile size, unroll factor, prefetch distance

Variants such as different data organization or data placement, different loop order or other representation of computation

• Beyond libraries

Can specialize to application context (libraries used in unusual ways)

Can apply to more general code

• Complementary and easily composed with application level support

## V. CONCLUSION

We show how to compile high-level programs with security by smashing the stack. We believe this approach provides a safer, more reliable alternative to security design. In this paper we have shown that self recursive improvement which optimizes the code faster. In this paper mostly focused on structure of systems and expressing/generating search space. As we gain experience with adaptive compilation, we hope to learn enough about the behavior of the optimizations and their interactions to allow the compiler to perform all or part of the search analytically.

## VI. ACKNOWLEDGMENT

The author would like to thank SNIST, CSE department HOD and colleagues for the help with editing this paper.

I thank Dr.Aruna Varanasi for their many valuable and thoughtful suggestions.

## REFERENCES

[1]. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In 10th ACM Conference on Computer and Communications Security, pages 220–230, 2003.

[2]. P. Laud. Semantics and program analysis of computationally secure information flow. In 10th European Symposium on Programming (ESOP 2001), volume 2028 of LNCS. Springer-Verlag, Apr. 2001.

[3]. S. Callanan, D. J. Dean, and E. Zadok. Extending gcc with modular gimple optimizations. In Proceedings of the GCC Developers' Summit'2007, 2007.

[4]. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005), pages 29–46, November 2005.

[5]. J. Ullman. Principles of database and knowledge systems. Computer Science Press, 1, 1988.

[6]. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In Proceedings of the Conference on Machine Learning, 2000.

[7]. John Backus. The history of Fortran I, II, and III. In Wexelblat, editor, History of Programming Languages, pages 25–45. Academic Press, 1981.

[8]. Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In Proceedings of the ACM SIGPLAN 99 Conference on Language Design and Implementation, May1999.

[9]. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching _xed dimensions. J. ACM, 45(6):891{923, 1998.

[10]. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, Mar. 1995.

## About Author

| | |
|---|---|
| | A.Yaganteeswarudu,Working as assistant professor in CSE department in Sreenidhi institute of science and technology (Autonomous), Ghatkesar, Hyderabad, Andhra Pradesh. Having 6 Years of experience in teaching. Completed M.Tech in SJCET, Kurnool in 2012. |
| | Surekha Pinnapati,Completed BE at PDACE Gulbarga,Completed M.Tech at BIET Davangere,Having 2 years of experience in teaching.Currently working in Dr.MVSIT. |