Stacy—static code analysis for enhanced vulnerability detection

Pankaj Lathar1*, Raunak Shah2 and Srinivasa K G1

Abstract: Computer program analysis refers to the automatic analysis of the behavior of a user defined program. An application of program analysis is to determine the quality of source code. Humans are prone to errors and, in most cases, the penalty of deploying low quality code is very high for a large organization. These errors often give rise to potential security vulnerabilities in an application, which could be exploited by malicious users. In this paper, we present Stacy—a tool that statically detects potential security vulnerabilities present in input source code. Static program analysis is the examination of source code prior to its execution. Our tool attempts to predict the behavior of a program before it is deployed. Stacy uses novel techniques to detect the primary sources of vulnerability in the source code of a program and informs the developer.

Keywords: static analysis; control flow graph; uninitialized variables; buffer overflow; memory leaks

1. Introduction

The standard of code quality that is deemed acceptable has risen vastly in recent times. Along with efficiency, quality has gained increasing importance among organizations. Code that has been developed by the user and passed through a compiler does not necessarily adhere to a high coding standard of code quality. This paper presents the algorithms used to detect certain security vulnerabilities that Stacy checks for, as well as its working on real world programs.

ABOUT THE AUTHORS

Pankaj Lathar, currently working as senior faculty in the Department of Information Technology, at CBP Government Engineering College, Jaffarpur since 2010. He has published many papers in International Conferences and Journals of repute. His area of research includes Cloud Computing, M-Commerce and Computers in Education.

Raunak Shah obtained his BE in Computer Science and Engineering at MS Ramaiah Institute of Technology. He is currently working there as a Research Assistant.

Srinivasa K G is currently working as an associate professor in CBP Government Engineering College, Delhi. He received his PhD in Computer Science and Engineering from Bangalore University in 2007. He has published more than 100 research papers in International Conferences and Journals. He is the senior member of IEEE and ACM. His research areas include Data Mining, Machine Learning and Cloud Computing.

PUBLIC INTEREST STATEMENT

Every software application we use today has been created manually by a developer or group of developer. Application code written in any development language needs to adhere to certain syntactical standards governed by a tool known as the compiler. Most programs that adhere to these standards still contain flaws that are overlooked by the compiler. These programs are syntactically astute but logically flawed. Such loopholes can potentially cause drastic security vulnerabilities in a program. Malicious users can make a program perform in ways that it is not meant to. Thus it is important to develop tools that check for these potential flaws, on top of the working of the compiler.

We have created one such tool—Stacy. This paper presents the algorithms used to detect certain security vulnerabilities that Stacy checks for, as well as its working on real world programs.
standard. The quality of code could be low due to defects introduced by the developer that make it vulnerable and reduce the functionality of the code.

Languages like C do not trade performance for safety although it is used for many low level utilities such as the operating system as well as security algorithms that run the Internet infrastructure such as the SSL protocol for secure web browsing. Bugs found in such systems have been used by hackers to break into them (Chess & West, 2007). It is evident that low quality code can introduce security vulnerabilities that may widely affect the functions of an organization.

Trivial mistakes are a part and parcel of the program development process. Most of the time, these mistakes are of not much consequence. The compiler highlights the error, which is fixed by the programmer. However, this cycle of feedback and action does not apply to most security vulnerabilities, which can be overlooked by the compiler and exist unnoticed in the program source code. The longer a defect on the software lies dormant, the more expensive it can be to fix. Static analysis is the analysis of computer code that is performed without actually executing programs. A static code analysis tool automatically checks the source code for compliance with a predefined set of rules given by the organization.

Manual reviewing is a form of manual static analysis. It is a time consuming process and highly prone to human error. To perform it effectively, human code auditors must be fully aware of what type of errors they are searching for before they can rigorously inspect the code. A common practice is for the developer to review their code as it is written. This contributes to detecting errors at an early stage of development. The manual reviewing process is not exhaustive and is prone to human error, especially in larger programs. Thus active research has been applied to automate such actions.

Thus the focus has shifted to automated static analysis techniques. This approach feeds the system with a predefined set of rules that must be adhered to pass the user code as vulnerability free.

In this paper, we define certain quality metrics and present Stacy, a tool that automatically identifies potential security vulnerabilities in input source code based on the identified metrics. Stacy runs as a plug-in for Eclipse IDE. The input source code is expected to be written in C. Results on test cases are also included in this paper.

2. Related work

The debate of dynamic analysis versus static analysis is a never-ending one. In contrast to static analysis, which determines characteristics of a program in all possible execution paths, dynamic analysis determines properties of a program in the execution path taken by the current execution. The usefulness of dynamic analysis is due to two of its properties: dependence on program input and precision of information (Ball, 1993). Static analysis is considered a more thorough examination of input source code. Recently, many applications have combined the properties of static and dynamic analysis to provide a more complete solution (Aiken et al., 2007).

Presence of uninitialized variables often goes undetected in source code, as the compiler ignores such occurrences. They can cause errors in program execution and eventual system collapse. To cope with the used-before-set problem, variables are initialized at compile time, when they are declared, to a predefined value. This ensures program consistency but not program correctness (Nguyen, Irigoin, Ancourt, & Coelho, 2002). LCLint is an advanced C static checker. It detects the use of a value of a location before it is initialized, by defining formal specifications written in LCLint language (Evans et al., 1994).

Jana and Naik (2012) present a technique that uses a combination of source and binary instrumentation. It tracks variables of basic types, individual array elements, and fields of structures. The tool is precise and complete but requires further optimization on the size of instrumentation information.
Memory leaks created by an application can lead to slower execution of the program and eventual unavailability of memory. Algorithms exist to track the flow of values from allocation points to deallocation points using a sparse representation of a program, consisting of a flow value graph (Cherem, Princehouse, & Rugina, 2007). The approach is to denote edges in the value flow graph with guards. These represent branch conditions.

One particular algorithm (Orlovich & Rugina, 2000) assumes the presence of memory leaks and runs a backward heap to the assumption and hence proves its absence by contradiction. This algorithm is effective with routines that manipulate linked lists and trees. This algorithm can be used on incomplete code and can identify the inputs that can cause the leak. It can be used as an interactive tool by the programmer to query particular statements in the program.

In programs with explicit memory management, memory leaks can be detected using a context and path-sensitive algorithm (Xie & Aiken, 2005). This algorithm is based on an underlying escape analysis: any allocated memory in procedure T that is not deallocated in T and does not escape it, is leaked. This algorithm works effectively even on dynamically allocated memory. This algorithm is scalable and its use of Boolean constraints ensures the detection of memory leaks with low rate of false positives. It is also computationally intensive and analyses functions in parallel as long as there are no dependencies.

Clouseau (Heine & Lam, 2003, 2006) is a leak detection tool that uses a notion of pointer ownership. It tracks variables responsible for freeing heap cells and implements the analysis as an system based on ownership constraint. Saturn (Xie & Aiken, 2005) reduces the problem of memory leak detection to a Boolean satisfiability problem, and then uses a SAT-solver to identify potential errors.

Buffer overflow attacks are an important and incessant security problem. Several run-time solutions to buffer overflow attacks have been proposed like StackGuard (Cowan et al., 1998) and Software Fault Isolation yet, buffer overflow attacks remain a problem. Much of this may be due to the lack of awareness of the extent of the problem and the availability of practical and efficient solutions. There are well-founded reasons why the run-time solutions are not acceptable in some environments. Run-time solutions always incur some overhead performance penalty. Another problem with solutions at run-time is that while they may be able to detect or avoid a buffer overflow attack, they instead turn it into a denial-of-service attack. On detecting a buffer overflow, there is often no other way to recover other than execution termination.

Static checking detects likely vulnerabilities before deployment thus overcoming these problems. Detecting buffer overflow vulnerabilities by analyzing code in general is an undecidable problem. Nevertheless, it is possible to produce useful results using static analysis (Larochelle & Evans, 2001).

The most common occurrence of buffer overflow is the runtime stack overflow, as a general practice for developers is to use stack allocated arrays (Dahn & Mancoridis, 2003). This allows the attacker to modify the control flow of the program by writing outside the bounds of an array onto a return address on the run-time stack. If the arrays are positioned differently in the heap at compile time, no attack succeeds. Not to mention, repositioning the buffers to the heap should disorder the heap memory, enough to avoid many heap overflows as well. A tool called Gemini repositions stack allocated arrays at compile time using TXL. Though a different language, TXL, is being used, it can be used as the base algorithm and made compatible with C. The major advantage of this implementation is that the semantics of the program is preserved.

LCLint (Larochelle & Evans, 2001), a static analysis tool, uses source-code comments to detect points of buffer overflow occurrences. This tool has the disadvantage of difficulty in creating additional information. To perform static analysis with this tool, the programmer has to add information other than source code to various sections. Accuracy of the information is assumed by
the tool. This method is effective if the added information is correct. However, it is an onerous task to insert accurate information in this manner, especially in the case of a large program.

Since LCLint is designed to analyze specific functions, its ability to detect points of buffer overflow occurrences is limited. Moreover, this method has the disadvantage of providing insufficient information: even if this tool can detect cases of buffer overflow, it is extremely difficult to isolate a program structure that may cause buffer overflow.

Stacy serves as an engine that makes checks using deep path analysis. The system checks one function at a time—allowing developers to quickly analyze if the function they are working on has any security issues. The results can be observed in Eclipse IDE; via a plug-in. Issues pointed out to developers early in the development cycle are less expensive to correct. Stacy does not require all header files or any dependencies to be checked.

Further, our system, unlike many others, does not require any inputs from the developer, in the form of comments—instead performing a deeper path analysis.

3. Control flow analysis

Control-flow analysis is a commonly used technique for static code analysis. The program flow is depicted as a directed Control Flow Graph (CFG). A CFG is directed graph that is used to represent blocks of code in the form of nodes, the control dependencies in the form of directed edges, starting with an entry node and concluding with the exit node at the end point of the program. A simple CFG is shown in Figure 1. The CFG for given input source code must completely illustrate all possible execution paths the program may take. Any traversal from an entry node to an exit node, through the graph, represents a valid execution path of the program.

The CFG along with control flow information is represented by an abstract syntax graph representation such as Abstract Syntax Tree (AST) (Söderberg, Ekman, Hedin, & Magnusson, 2013). In an AST, each node of the tree denotes a construct occurring in the source code. A parser reads the source code and produces an abstract syntax tree, which models all of the structural information contained in the source code.

Our tool analyzes a program by creating a CFG from the given input source code, as mentioned above. Each node represents a relevant piece of code for our analysis, and directed edges from a node represent the possible paths program execution may take from that node. Our algorithms traverse the graph in a depth first manner (Figure 2).

4. Techniques adopted

4.1. Detecting the use of uninitialized variables

A compiler does not check for the use of uninitialized variables in source code, yet the flow of execution of a program is dependent of the values of the variables it manipulates. Variables with garbage values could cause incorrect execution of the program. The consequences may affect the global running state of the program, as in the case of returning an uninitialized variable or setting a global variable with an uninitialized variable.

This section of the paper demonstrates how Stacy detects the incorrect use of uninitialized variables.
All variables must be declared in a program. The compiler flags the use of any variables that have not been declared as an error. In C, all variables can only be declared at the start of a new scope. A variable ceases to exist beyond the end of the scope in which it was declared. The memory in which held the variable and all its metadata is available for reuse.

Variables that are simply declared at the start of a scope are allocated memory but their value is not initialized. They contain garbage values that cannot be determined. These variables are said to be uninitialized.

Variables that are declared in a scope hold values that are required for the correct execution of that block of code. The incorrect use of a declared variable goes unnoticed and the effect it has on the program is untraceable.

```
int main()
{
    int i, ifVar, globalVar;
    if (ifVar=3)
    {
        if (ifVar==3)
        {
            int i=3;
            globalVar=i;
        }
        else{
            globalVar=i;
        }
    }

    globalVar=i;
}
```
Certain statements, which are related to a conditional statement, may be located within the scope associated with the conditional statement. Variables declared within this scope are local to it, and cease to exist outside of the scope. Figure 3 depicts an incorrect initialization, along with the CFG that would be created by such a program.

The algorithm used by Stacy to detect the use of uninitialized variables is shown in Figure 4, which performs a traversal of the CFG created during the parse. Stacy maintains two data structures for any given input source code: a variable symbol table; which contains the list of variables declared and an initialization symbol table. Both the symbol tables are scope driven—i.e. at any point of traversal, it is possible to access the state of the system at that point. In order to do this, the current state is retained at every point of possibility for a new state.

An assignment of one variable, denoted as an LHS variable, by another variable, denoted RHS variable, in the source code must ensure its validity by observing its presence in both the symbol tables.

A node may contain more than one outgoing edge, which causes a branch in the CFG. On reaching such a node, a new scope begins and Stacy saves the current state of the two tables. The parent state is restored at the end of any scope. This serves two purposes: firstly, local variables declared in a scope cease to exist at the end of the scope. Secondly, a global variable may be initialized within a scope, but incorrectly used thereafter in the program. Hence the structures must be reset before traversal proceeds.

Stacy keeps track of whether a variable has been initialized in all possible paths at every point, in which case its use in the program would be valid at a later stage of traversal. Thus only variables whose initialization affects the global state of the program may be used as assignment variables.
4.2. Detecting the presence of potential memory leaks

Low-level programming languages such as C provide manual memory management and require explicit deallocation of program structures by programmers. As a result, memory leaks represent a standard cause of errors in such languages. Dynamically allocated blocks of memory are referenced by pointers during execution. In case such a block is still referenced by one or more reachable pointers at the end of the execution, fixing the leak is often quite simple as long as it is known where the block was allocated. If, however, all references to the block are over-written or lost during the program’s execution, only knowing the allocation site is not enough in most cases. Memory leaks are difficult to detect since the only indication is through high consumption of memory, resulting in slower execution speed. For long-running applications, the system may eventually run out of memory due to this problem.

This section describes how Stacy detects potential memory leaks present in the given source code.

Among other properties, Stacy differentiates nodes in the CFG, created during the parse of input source code, based on their type. Stacy assigns a unique representation in the CFG to any statement in the source code that allocates memory to a pointer variable by the use of a dynamic memory allocation function. It uses a special structure to represent all such nodes present at any point of time. This special structure contains information about the variable that was assigned memory by a given statement in the source code. Stacy keeps track of every structure with a member that determines whether the region of memory that was dynamically allocated to a variable has been specifically deallocated by the developer.

In the simple case, all pointer variables are assigned to memory with the memory allocation functions and deallocated using the “free” function.

Figure 5 depicts an example of a potential memory leak, along with the CFG that would be created by such a program.

Generally, a pointer does not point to only one location in memory throughout the course of the program. It may point to different memory locations and different pointers may point to the same location by virtue of assignment. This brings out the essence of pointers and Stacy takes it into consideration in its algorithm.

Thus, the structure described above is modified to contain a list of indices. Each index represents a variable that would point to the memory location at any point of time, during execution of the program. Deallocation of a region in memory, referring to it by any of the pointer variables that point to it, is universally accepted.

At the end of the program execution, all dynamically allocated memory locations must be freed to prevent the presence of memory leaks. The path taken by the program during execution will not be known to Stacy during analysis; hence the solution is to deallocate all memory that was allocated during the course of the program, treating every possible execution path separately.
This follows a similar procedure to the one explained in the previous section. On encountering a node with multiple outgoing edges in the CFG, the current state of the system is saved. The parent state is restored at the end of the scope, while any properties of the current state that affect the parent state are retained.

At the end of the traversal, Stacy expects all dynamically allocated memory locations to be specifically deallocated by the programmer in all possible execution paths of the program.

Figure 6 depicts the algorithm we have just presented.

### 4.3. Detecting the presence of potential buffer overflows

Buffer overflow refers to data overflowing into a region not intended by the developer. Buffer overflows account for approximately half of all security vulnerabilities. Programs written in C are particularly more susceptible to buffer overflow attacks. It provides a loophole for attackers to exploit—from allowing them to interfere with regular operation to providing full control over the process.

Whether a buffer overflow emerges or not in program runtime, is dependent upon inputs of the executable program.

Stacy uses an algorithm to detect buffer overflows statically. Static detection ensures the detection process is exhaustive. Runtime analysis of buffer overflows present in input source code provides, at best, partial detection of overflows.

Figure 7 shows an example of a potential buffer overflow, along with the CFG created by such a program.

```c
TraverseForMemoryLeaks(struct node* graphNode)
Input: startNode in CFG
Output: Potential memory leaks
currentNode = graphNode;
if currentNode is dynamically allocated memory do
    create newStructure;
if currentNode is conditional node do
    push(stateOfSystem)
if (currentNode->type==lhs) do
    push(currentNode) onto existing structure variable
if currentNode frees dynamically allocated memory do
    push(current state) to tracking list
if (currentNode->next==NULL)
    return;
while (currentNode->next) do
    check for common initializations between current path and previous path
    TraverseForMemoryLeaks(currentNode->next)
```
A variable that denotes an array has the ability to hold a sequence of elements of the same type. Similarly, a pointer variable may point to the first of a list of elements of the same type. Any element in the sequence can be accessed via an index or using pointer arithmetic.

An access to an index that is not a valid, i.e. an index whose value is outside of the range of permissible values, gives rise to a buffer overflow.

Stacy attempts to determine the safety of a variable that can be used as the index in any array access. A safe variable may be used as the index to an array without the possibility of creating a buffer overflow.

To establish the safety of a variable, its value must have been inspected at the beginning of the current scope. Inspection of a variable, in terms of a selection or iterative statement, ensures that the developer is aware of the state of the system when performing an array access. Stacy assumes no errors by the developer in terms of evaluation of the variable, i.e. an inspected, and therefore safe, variable is in the valid range for array access.

Initially, all variables are assumed to be unsafe for usage as index variables.

For every scope, Stacy tracks safe and unsafe index usage. Variables declared in a scope are demarcated as unsafe by default. At every new scope, the current state of the system is saved and Stacy internally has access to the ancestral state of safety.

```c
int main()
{
    int a[10], b;
    b = 5;
    if (b < 5) {
        int c;
        int [c] = 5;
    }
}
```

TraverseForBufferOverflow(struct node *graphNode)
Input: startNode
Output: Potential buffer overflows present in source code
Maintain 2 symbol tables - safe and unsafe; both are scope based

Variables in a conditional node should be added to the safe list for all paths associated with it
A variable that is assigned to, should be made unsafe
If a variable at a node has sub-variables, check for all sub-variables to be safe.
If any sub variable is unsafe, mark it for potential buffer overflow
Only variables that are tracked as safe, either in the current scope or any ancestor scope, are valid for use as array indexes. Stacy is constantly inspecting and updating the state of safety of the system.

Stacy treats inspection of a variable followed by its assignment as a special case. Assignment supersedes inspection, and subsequently the variable is tracked as unsafe in the current state and in every ancestral state following its assignment.

Figure 7 depicts the algorithm we have described above (Figure 8).

5. Architecture
Figure 9 depicts the architecture and data flow of the entire system. The developer writes source code in Eclipse. At any time during the development life cycle, a developer can instantiate static analysis on the source code. Input source code is first converted into its corresponding CFG. The required analysis is performed on this form of the source code, and the results of analysis are returned to the developer.

6. Applications
Enterprise security is focused on the application layer. Since the enterprise perimeter is almost completely impenetrable to malicious users, these individuals focus on exploiting weaknesses in enterprise applications. Static code analysis is a security tool that an enterprise can use to identify vulnerabilities in code before the application is deployed. Stacy reviews source code line by line to detect security vulnerabilities before the code is released into production. Performing this analysis early in the Software Development Life Cycle diminishes the cost of correction to enterprises. It also increases the efficiency of the development process.

Many large enterprises use vendor written code or third party software in their products. This code may be tested for security flaws to the organization before it is embedded into the product.

Static analysis may also be used in the development of mobile and web applications. Developing mobile applications is a challenging task: developers need to support their app on multiple platforms but only have limited resources to deal with. This results in an increasing use of cross-platform development frameworks that allow developing an app once and offering it on multiple mobile platforms such as Android, iOS, or Windows.

A major challenge in this cross platform model is to ensure the quality of apps. Besides the usual sources of errors, mobile apps are susceptible to a number of specific errors. Independent
developers develop a majority of apps. Static program analysis would be helpful to building secure and high quality mobile apps.

Android applications run on mobile devices that have limited memory resources. Although Android has its own memory manager with garbage collection support, many applications currently suffer from memory leak vulnerabilities. These applications may crash due to out of memory error while running.

Internet applications have become one of the most important communication channels between various kinds of service providers and clients. More services are provided via the World Wide Web on a daily basis. Research efforts to create technologies and standards that meet the requirements and expectations of today’s applications and users is being heavily undertaken. Thus the negative impact of security flaws in such applications has grown as well.

7. Test cases
This section presents functional testing performed on programs in the runtime environment of Stacy. Stacy is provided as a plug-in to Eclipse IDE. All input source code was compiled in the Eclipse framework before Stacy performed its analysis on it.

The objective of these tests was to ascertain the correctness of our system as well as to observe the need to carry out static analysis on commercial embedded applications. While independent developers may be susceptible to ignoring such deficiencies in programs, even developers in larger organizations tend to overlook the need for high quality code.

We tested Stacy against the source code of three popular, open source embedded applications—a dual boot loader, a portable vnsprintf implementation and Mongoose, and embedded web server/network library. All three applications are popular and mature, having been in development for many years and with multiple contributors. Initially, we did not expect to find any security bugs. However this was not the case.

The results of the analysis are summarized in Table 1.

In the boot loader, Stacy noticed two instances of initialization by global variables that are uninitialized in a certain path of the CFG. One such error caught is shown in Figure 10.

In the vnsprintf application, Stacy flags one case of a potential buffer overflow, shown in Figure 11.

Mongoose is a fully deployed embedded web server or network library. Stacy was run across the source code of different C files. It flagged 3 cases of potential buffer overflows and 3 cases of potential memory leaks, as show in Figure 12.

On performing analysis manually at the flagged lines, we notice that the errors raised are accurate and potentially harmful to the system. It proves that even fully deployed applications are susceptible to these types of errors.

| Table 1. Defect distribution in tested applications |
|---------------------------------------------------|
| **Boot Loader** | **Vnsprintf implementation** | **Mongoose** |
| Use of uninitialized variable | 2 | 0 | 0 |
| Potential memory leak | 0 | 0 | 3 |
| Potential buffer overflow | 0 | 1 | 2 |
Figure 10. Line 207 flagged as use of uninitialized variable. Variable “SPDR” has not been initialized in every path of the CFG that leads to this line.

Figure 11. Line 101 flagged as potential buffer overflow. Variable “len” is not safe for array indexing in the do-while construct.
8. Conclusion
In this paper, we have presented lightweight techniques that our tool, Stacy, uses to statically analyze input source code. We have identified the three most important and common causes of quality degradation in source code—usage of uninitialized variables, presence of memory leaks and presence of buffer overflows—and created Stacy to detect the same using novel techniques.

With growing competition among enterprises to deploy quality applications, with minimal time allotted for development, automatic analysis techniques are becoming increasingly important. It is essential to perform analysis early in the development life cycle to reduce the costs of fixing bugs. Stacy helps in this regard as a fully working application is not needed to run the analysis, as in the case of dynamic analysis. Stacy provides enterprises with a lightweight tool to perform accurate vulnerability detection.

Funding
The authors received no direct funding for this research.

Author details
Pankaj Lathar1
E-mail: pankajlathar@gmail.com
Raunak Shah2
E-mail: raunakshah13@gmail.com
Srinivasa K G1
E-mail: kgsrinivasa@gmail.com

1 Department of Information Technology, CBP Government Engineering College, New Delhi, India.
2 Computer Science and Engineering, M.S. Ramaiah Institute of Technology, Bangalore, India.

Citation information
Cite this article as: Stacy—static code analysis for enhanced vulnerability detection, Pankaj Lathar, Raunak Shah & Srinivasa K G, Cogent Engineering (2017), 4: 1335470.

References
Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hawkins, P., & Hockett, B. (2007). An overview of the saturn project. In Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (pp. 43–48).
Ball, T. (December, 1993). The concept of dynamic analysis: ACM letters on programming languages and systems.
Cherem, S., Princehouse, L., & Rugina, R. (2007). Practical memory leak detection using guarded value-flow analysis. New York, USA. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.
Chess, B., & West, J. (2007). Secure programming with static analysis: Upper Saddle River, NJ: Pearson Education.
Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., ... Zhang, Q. (January, 1998). Automatic detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium.
Dahn, C., & Mancoridis, S. (2003). Using program transformation to secure C programs against buffer overflows. In WCRE ’03 Proceedings of the 10th Working Conference on Reverse Engineering.
Evans, D., Guttag, J., Horning, J., & Tan, Y. M. (1994). LCLint: A tool for using specifications to check code. In ACM SIGSOFT Symposium on Foundations of Software Engineering (pp. 87–96).

Heine, D. L., & Lam, M. S. (2003, June). A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In Proceedings of the ACM Conference on Program Language Design and Implementation, San Diego, CA.

Heine, D. L., & Lam, M. S. (2006, May). Static detection of leaks in polymorphic containers. In Proceeding of the International Conference on Software Engineering, Shanghai, China.

Jana, A., & Naik, R. (2012, October). Precise detection of uninitialized variables using dynamic analysis - Extending to aggregate and vector types. In 2012 19th Working Conference on Reverse Engineering, Kingston, ON.

Larochelle, D., & Evans, D. (2001, August). Statically detecting likely buffer overflow vulnerabilities. In Proceedings of the 10th USENIX Security Symposium.

Nguyen, T. V. N., Irigoin, F., Ancourt, C., & Coelho, F. (2002). Automatic detection of uninitialized variables. International Conference on Supercomputing, Ecole des Mines de Paris.

Orlovich, M., & Rugina, R. (2000). Memory leak analysis by contradiction. In: Proceedings of the 2000 International Conference on Compiler Construction, Berlin, Germany.

Söderberg, E., Ekman, T., Hedin, G., & Magnusson, E. (2013). Extensible intraprocedural flow analysis at the abstract syntax tree level. Science of Computer Programming, 78, 1809–1827. https://doi.org/10.1016/j.scico.2012.02.002

Xie, Y., & Aiken, A. (2005, September). Context- and path-sensitive memory leak detection. In ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal.