Flow-insensitive Static Analysis for Detecting Integer Anomalies in Programs

Dipanwita Sarkar Muthu Jagannathan Jay Thiagarajan
Ramanathan Venkatapathy

Center for Software Excellence, Microsoft Corporation
{dipas,muthuj,jaythia,ramanv@microsoft.com}

Abstract. This paper describes a static analysis algorithm to detect potential integer anomalies in software. Integer anamolies take place when arithmetic operations on integer values yield new values that cannot be represented in the range for the integer type. Two common integer anomalies are integer overflow and integer underflow. Unexpected behavior can result if an attempt is made to represent a value outside the range of the integer type. Such anomalies in integers representing buffer sizes can lead to serious buffer overruns that compromise the security of a system.
In this paper, we present a flow-insensitive static analysis algorithm that detects such missing integer range validations. We walk the AST, build a constraint graph that records range relationships between variables, and use this to ensure that all interesting uses of integers have been verified. Although the analysis is not sound or complete, its performance is significantly better than a flow-sensitive approach. We ran the analysis on approximately 50 MLOC from future versions of Microsoft products. We successfully uncovered and fixed over 2000 such anomalies with an overall noise rate of as low as 6.76 percent.

1 Introduction

Integer anamolies take place when arithmetic operations on integer values yield new values that cannot be represented in the range for the integer type. Two common integer anomalies are integer overflow and integer underflow. An integer overflow can make the result of an addition smaller than its parts. An integer underflow can make the result of a difference greater than either of the parts. Unexpected behavior can result if such a range check is missing and an attempt is made to represent a value outside the range of the integer type. Obtaining a smaller integer where a larger one is required and vice-versa can result in buffer access violations that may often lead to security holes in the program [15, 21, 20].

Integers are used in programs to represent various entities such as disk quota, account balance, etc. However, its use in representing buffer sizes turns out to be the most interesting and critical one from the perspective of computer security. A buffer typically represents a range of accessible memory locations. Operations

on a buffer should not access memory locations outside the buffer bounds. Such bound checks on buffers can be programatically captured by range checks on integers that are used to access locations in the buffer or to specify its size.

There have been CERT [5, 6] and Microsoft Security Bulletin [17] evidences of such vulnerabilities in, the SSH1 protocol caused due to a remote integer overflow exploit, in Sun Microsystems XDR library and in Microsoft's Internet Explorer, where a malicious bitmap caused an integer overflow, allowing remote execution of arbitrary code. Integer overflow has been wellknown but never thoroughly studied. The regular C++ compiler and tools like LCLint [12] detect such anomalies for all integer casts and arithmetic respectively. However, to our knowledge, there has not been any static analysis tool that focuses the analysis on finding integer anomalies that compromise the security of a system.

The problem space is vast, as integers come in many varieties, with different limited ranges and in signed and unsigned combinations of those. Besides this, the presence of casting rules makes the problem quite complex. Integers are so common in programs that it would be infeasible to report overflow or underflow in every integer whose bounds are not properly checked.

In this paper, we have reduced the problem space by focusing on certain integer overflow scenarios that can compromise the security of the system. Such security-critical vulnerabilities typically arise when an integer overflow or underflow takes place in an integer that is used in the context of memory allocation, memory access or loop termination.

Integer anomalies often occur when there is a mismatch between the programmer's intent and the program's execution. Through our analysis of the source program, we try to infer the programmer's intent by capturing ordering relationships between integer expressions. An *ordering relationship* between integer expressions can be of any of the following types: *greater than*, *less than*, *equal to* and a combination of those. We use a *constraint graph* with the nodes as integer expressions and the edges as the ordering relationship between the nodes, to represent this. Some examples of integer expressions and their corresponding ordering relationships are shown in Fig. 1. These ordering relationships can include some unsound ordering assumptions made by the programmer, based on the idea that the integer will never overflow or underflow. Some examples of such unsound assumptions are shown in Fig. 2.

Expression	Ordering Relationship(s)
x – 1	(x - 1) < x
x + 1	(x + 1) > x
x + y	(x + y) > x, (x + y) > y
x – y	(x - y) < x

Fig. 1. Ordering Relationships.

Ordering Assumption	Why unsound
x +1 > x	**(x + 1)** may overflow to a value less than **x**
x - 1 < x	**(x - 1)** may underflow to a value greater than **x**
x + y > x	**(x + y)** may overflow to a value less than **x**
x + y > y	**(x + y)** may overflow to a value less than **y**
x - y < x	**(x - y)** may underflow to a value greater than **x**

Fig. 2. Unsound Assumptions.

We find the defects that result from such assumptions by trying to verify these ordering relationships and identifying a relationship that remains unverified. Verification can be implicit, explicit or some combination thereof. A bounds check on an integer in one of the program instructions is an example of explicit verification. A bounds check on an *equal* integer is an example of implicit verification. Each integer expression node has a property called *bound status* that can be *unbound*, *partially bound* or *completely bound*.

Our approach is flow-insensitive. We perform a recursive tree-walk on the AST representation of the program. The constraint graph is created as we recur down into the subtrees. The constraints on the edges are verified and the edge properties updated as we return from the recursion. Finally, we traverse the constraint graph to update the node properties and identify unverified edges. The presence of an unverified edge corresponds to the presence of a software defect. In order to scale to millions of lines of production code, our analysis is local to a function. However, we make use of source level annotations [14, 1, 7, 13, 23] to communicate the bounds information from the callers to the callees, at callsites. This method is not only scalable but also effective in helping us detect common but dangerous integer anomalies in code.

The contributions of this paper can be summarized as follows:

- We frame the problem in a way that reduces the problem space and finds the most critical defects first.
- We provide a unique way of capturing the programmer's intent and assumption about integers in the program.
- We present a scalable solution by restricting ourselves to a flow-insensitive local analysis.
- We have applied this technique to a variety of future Microsoft products and successfully uncovered and fixed over 2000 such anomalies in 50 MLOC, with an overall noise rate of as low as 6.76 percent. Our experiments show that this is a practical, scalable, and accurate solution.

The remainder of the paper is organized as follows. In Section 2, we provide a motivating example for integer anomaly detection. In Section 3, we describe our static analysis algorithm used to detect potential integer anomalies. In Section 4

we describe our analysis infrastructure and the annotations used to aid the analysis. Results demonstrating the yield and scalability of our approach are presented in Section 5. In Section 6, we review related work. We conclude by summarizing the advantages of our approach in Section 7.

2 Example

In this section, we show with an example, how an integer overflow can lead to a buffer overrun with serious consequences.

The purpose of class `CImageParser` in the code excerpt in Fig. 3 is to parse image files. One of its methods, `CImageParser::InitColorTable` initializes the color table `m_pColorTable`, with the image data. The method takes two arguments, `numColors`, which represents the number of colors in the image file and `pImageData`, which represents the `BYTE` stream of the image file. The method first calculates the size of memory needed for the color table, allocates memory for the color table and then initializes it with the image data. If the method is called with a very large untrusted value for `numColors` say, `0xFFFFFFFF`, the integer expression $numColors * sizeof(structRGB)$ will overflow and the computed size of the color table `tableSize` will be much smaller than the expected size. As a result, when the memory allocation takes place, a color table of a smaller size than required, is allocated. On subsequent intialization of the color table, a buffer overrun will take place. Assuming that the image file being read is from an untrusted source like network, such a buffer overrun could lead to arbitrary remote code execution.

The integer anomaly described in Fig. 3 can be detected with the algorithm described in Section 3. The algorithm detects the integer anomaly by constructing a constraint graph for the integer expressions in the function. The nodes in the graph represent the integer expressions in the function and the edges represent the arithmetic ordering relationship between the integer expressions. There are three types of arithmetic ordering relationships, greater than, less than and equal to. Nodes corresponding to the integer expressions used in memory allocation or buffer access contexts are marked specially. If there is an explicit or implicit check in the program for the arithmetic ordering relationship between integer expressions, the edge connecting their corresponding nodes is marked as *verified*, otherwise it is marked as *unverified*. Integer anomaly in a program exists if there is an unverified edge in the constraint graph that is built for that program. Fig. 4 represents the constraint graph for the code excerpt in Fig. 3. The unverified edge in the graph between the $numColors$ node and $numColors * sizeof(structRGB)$ node identifies the integer anomaly in this program.

3 Algorithm

In this section we present an algorithm for detecting integer anomalies and suggest a remedy for the same. First, we walk through the abstract syntax tree

```
struct RGB
{
    unsigned short red;
    unsigned short green;
    unsigned short blue;
};

// Class to parse an image file.
class CImageParser
{
    struct RGB *m_pColorTable;
    ...

public:
    void InitColorTable(unsigned long numColors, BYTE *pImageData);
    ...
};

// Method to initialize color table with untrusted image data coming from a network.
void CImageParser::InitColorTable(unsigned long numColors, BYTE *pImageData)
{
    // Calculate the amount of memory to be allocated for the color table.
    unsigned long tableSize = numColors * sizeof(struct RGB);

    // Allocate color table.
    m_pColorTable = (struct RGB*)malloc(tableSize);

    // Initialize the color table.
    for(unsigned long i = 0; i < numColors; i++)
    {
        m_pColorTable[i].red   = (*(unsigned short*)pImageData++);
        m_pColorTable[i].green = (*(unsigned short*)pImageData++);
        m_pColorTable[i].blue  = (*(unsigned short*)pImageData++);
    }
}
```

Fig. 3. Motivating example for integer anomaly detection.

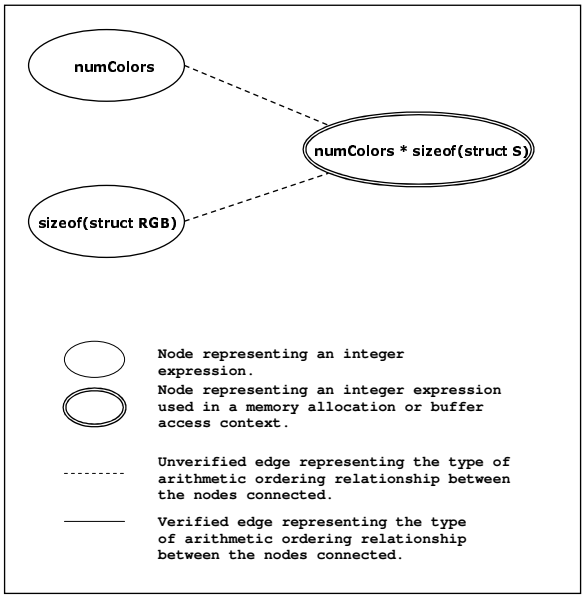

Fig. 4. Constraint graph for the example in Fig. 3.

of the program to build the constraint graph. Second, we apply these arithmetic constraints to determine which integer expressions have been checked for bounds and which edges are considered to be verified. Finally, we walk through the constraint graph and find unverified edges to detect integer anomalies.

3.1 Building Constraint Graph

A constraint graph of integer expressions is built to capture the ordering relationship between the integer expressions in a program. The nodes in the graph represent the integer expressions. Each node has a name for the expression and a set of properties. The properties for an integer expression are *sign*, *bound status* (e.g., unbound, partially bound and completely bound), *operation type* (e.g., addition, subtraction, multiplication and division) and an *usage context* (e.g., memory allocation size and buffer access index). The edges in the graph represent the arithmetic ordering relationship between the nodes. The properties of an edge are *relationship type* (e.g., greater than, less than and equal to) and *verification status* (e.g., verified and unverified) that tells if the ordering relationship has been verified or not. Each node can have a *set of edges* for each of the three types of ordering relationship mentioned above (e.g., greater-than set, less-than set and equal-to set). The analysis is targeted towards finding anomalies in buffer size expressions. Hence it assumes that the programmer's intent was to consider these sizes as positive.

The constraint graph is built by recursively walking through the abstract syntax tree of a program, considering only the integer expressions. Some examples of integer expressions are binary arithmetic operations like addition, subtraction, multiplication, division, unary arithmetic operations like pre-increment, pre-decrement, post-increment, post-decrement, relational operations like less than, greater than, equal to, assignments and casting operations.

For each binary arithmetic operation, the graph contains three nodes, two for the operands and one for the operation. The type of the edge that is constructed between the nodes is set based on the arithmetic operation being performed. For example, for the integer expression $a + b$, nodes for a and b are built first, followed by a node for $a + b$. Then two edges, one connecting a to $a + b$ and another connecting b to $a + b$ are constructed. The edge between a and $a + b$ is added to the greater-than set of node a and to the less-than set of node $a + b$. Similarly, the edge between b and $a + b$ is added to the greater-than set of node b and to the less-than set of node $a + b$.

For each unary arithmetic operation, the graph contains two nodes, one for the operand and one for the operation. The type of the edge between the nodes is set based on the arithmetic operation being performed. Example, for the integer expression $--a$, a node for a is built first, followed by a node for $a - 1$. Then an edge connecting a to $a - 1$ is constructed. This edge is added to the less-than set of node a and to the greater-than set of node $a - 1$. The postfix increment and decrement operations are treated in the same way as their prefix increment and decrement counterparts.

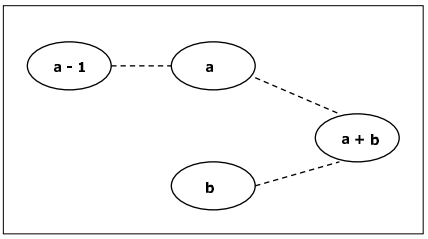

Fig. 5. Constraint graph for $a + b$ and $--a$ integer expressions.

For each relational operation, the bound status of both the operands is updated from unbound to partially bound or from partially bound to completely bound. For example, for the relational operation $a < a + b$, the bounds status of both a and $a + b$ nodes are updated. The default bounds status of a node when built is unbound.

For assignment operations, nodes for the operands are built first, followed by an equal-to edge between them. The bounds property of the right hand side operand is transfered to that of left hand side operand.

For cast operations, typically one node for the operand is built. However, if the cast operator is a size-changing or sign-changing operator, two nodes are built, one for the operand and one for the cast operation. If there are multiple cast operators involved, only the final cast operator is considered to determine whether it is a size-changing or sign-changing cast.

As the constraint graph is built, the bound status of the integer expression at each node is checked. If the integer expression is bound in any way, the bound status of the node is updated accordingly. For example, if the integer expression is an enumerator, the bounds property of the node is updated to be completely bound. If the integer expression is used in a memory allocation or buffer access contexts, the usage property of the node is also updated accordingly.

For example, the constraint graph for the code excerpt in Fig. 3 is shown in Fig. 4.

3.2 Applying Arithmetic Constraints

The constraint graph is walked and the arithmetic constraints based on the implicit ordering relationship between the expressions are applied. The result of applying a constraint is that the ordering relationship on the edges are implicitly verified and the bound status of the expression is updated. The constraint rules are described in Table 6. A *sub-expression* of an expression is represented by the expressions in the lesser-than edge set. A *super-expression* of an expression is represented by the expressions in the greater-than edge set.

Constraint Rule	Result of Applying Rule
1. All the sub-expressions of an expression are fully bound.	The edges between the sub-expressions and the expression are verified and all the equal expressions of the expression get the same bounds property as the expression.
2. One of the sub-expressions has a verified edge to the expression.	All the sub-expressions have a verified edge to the expression.
3. An equal edge between two expressions has been verified.	Both the expressions have the same sign and bound properties.
4. All the edges to a super-expression are verified.	The super-expression is fully bound.
5. Two expressions are fully bound and there is an unverified edge between them.	The edge is verified.

Fig. 6. Constraint rules.

For example, when you apply the arithmetic constraints to the constraint graph in Fig. 4, there is no change to it.

3.3 Defect Detection

Once the partial order graph is built and the arithmetic constraints are applied, detecting the integer anomalies is straightforward. Integer overflow issues are

detected by iterating through all the nodes in the graph with usage property as memory allocation or buffer access. From each expression, the sub-expressions in the less-than edge set are iteratively walked. This is done recursively until all the sub-expressions of the expression are reached. If the edge to any sub-expression is unverified, then the ordering relationship of the expression doesn't hold and an integer overflow is detected.

Detecting integer underflow issues in an expression is similar to detecting integer overflow issues. In that case the super-expressions in the greater edge set are walked.

In the example graph shown in Fig. 4, the context of usage of expression $numColors * sizeof(structS)$ is memory allocation size. This expression has two sub-expressions $numColors$ and $sizeof(structS)$ and the less-than edges to them are unverified. This shows that there is a potential integer overflow in the memory allocation size $numColors * sizeof(structS)$.

3.4 Remedy

An integer overflow defect in a program can be avoided by adding a simple check like the one shown below.

```
if (numColors > (UINT_MAX/sizeof(struct S)))
   return;
```

This explicit check in the program changes the bound status of the $numColors$ node and $numColors * sizeof(structS)$ node from unbound to partially bound. Since the lower bound of an unsigned integer is fixed, in this context, being partially bound is the same as being completely bound. Applying constraint rule no. 5. in Fig. 6, the verification status of the edge connecting $numColors$ and $numColors * sizeof(structS)$ is set to *verified*. Then applying constraint rule no. 2. in Fig. 6, the verification status of the edge connecting $sizeof(structS)$ and $numColors * sizeof(structS)$ is set to *verified*. The constraint graph after applying the implicit arithmetic constraints is shown in Fig. 7. Since all the sub-expressions of $numColors * sizeof(structS)$ have verified edges to it, no integer overflow issue will be found.

4 Implementation

This integer anomaly detection algorithm is implemented in C++ using the PREfast AST infrastructure [16]. PREfast is a static analysis tool that exposes the parse trees after resolving the types and symbols for a given program. It provides a helpful AST interface that can be used to iterate through and query properties of the abstract syntax trees, symbols and types of symbols in a program. Other than parsing infrastructure, PREFast also provides a suite of static

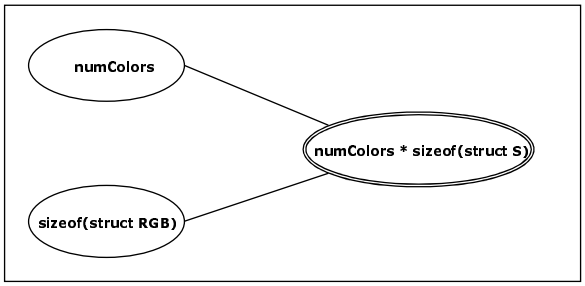

Fig. 7. Constraint graph after the integer overflow check has been added to the program in Fig. 3.

analysis modules that analyze C and C++ code to detect a class of errors not easily found by a typical compiler. Our analysis runs as one of PREfast's static analysis modules. In order to scale to millions of lines of production code, our analysis module runs locally on a function. However, we make use of source level annotations [14, 1, 7, 13, 23] described in this section, to communicate the bounds information from the callers to the callees, at callsites.

The programmer can use **__in_bound** annotation on an integer parameter to a function, if it is known to be bound without the actual range being known at compile-time. Any arithmetic involving it and compile-time constants will not result in an integer overflow defect. In this case, there is no need to add any integer overflow check to the function.

The programmer can use an **__in_range(const1, const2)** annotation if the range of the integer is known at compile-time. The annotation provides information to the algorithm that the parameter is in a fully bound state on entering the function.

Similarly, we have **__out_range** and **__out_bound** annotations for reference parameters and the return value of a function.

One of the main goals of this algorithm is to find integer overflow and underflow defects in the size parameter of memory allocation function calls. In order to bias the analysis toward this class of defect, we use **__allocator** annotation to mark those functions that allocate memory and thus are more prone to serious buffer overrun resulting from integer overflow and underflow defects.

In case none of the above annotations are sufficient for a particular scenario, the programmer can also use **__assume_bound** annotation to simply declare that the integer is completely bound. The usage of these annotations is shown in Fig. 8.

5 Results

Fig. 9 shows the number of lines of code for future versions of some Microsoft Products, the number of integer anomalies detected by our analysis, the number

```
void InitTable(__in_bound unsigned int numEntries, BYTE *ptr);

void InitTable(__in_range(0,5) unsigned int numEntries, BYTE* ptr);

__allocator void* GetMemory(unsigned int size);

__assume_bound(numColors);
```

Fig. 8. Examples of Annotations in Source Code.

of false positives reported and the percentage noise for each product. As seen from this table, the average noise rate for 50 MLOC is considerably low.

Fig. 10 shows the number of lines of code for the first three products followed by the comparision of the time taken to compile the sources and the time taken to run our analysis on them. As it is seen, the analysis time is comparable or less than the compile time.

The compilation and the analysis (32-bit executable) were both done on a 4 x Dual Core machine with 16 GB RAM (2GB per proccessor) running 64-bit Windows 2003 Server Enterprise Edition operating system.

Our analysis is limited by the availability of context-sensitive information, such as calls to string-length functions whose return value is expected to be completely bound. Lack of this kind of context-sensitive information is one of the sources of the resulting noise. However, noise resulting from this can be suppressed by adding annotations to such functions.

6 Related Work

6.1 Software validation tools

In recent years, research on software validation via static analysis has given rise to several static analysis tools like PREfix [4], SLAM [2] [3] and ESP [10], that are used to improve code quality on large commercial codebases. Most commercially used tools have to scale for millions of lines of source code. As a result, they often have to compromise the completeness and soundness of their approach. PREfix [4] is a successful global software validation tool, which truncates the search space to achieve scalability. LCLint [12] uses a mix of type system extensions and local dataflow analysis to report integer anomalies in all arithmetic operations in the the program. Such a tool may be more complete but also too noisy and infeasible for use on developer desktops. The goal of our integer anomaly tool is to prioritize defect detection and find the most critical defects first. It achieves this goal in a scalable fashion by restricting the analysis to function bodies, by focusing the analysis on integer usage in memory allocation, access and loop termination contexts, and by disregarding the flow of control for control structures like loops, branches etc.

Size in MLOC	No. Anomalies	No. False Positives	Noise Rate
2.1	90	0	0.00%
8.2	85	8	8.60%
1.8	102	38	27.14%
5.4	205	5	2.38%
4.4	395	28	6.62%
6.6	194	5	2.51%
2.2	141	31	18.02%
1.8	68	8	10.53%
9.5	308	27	8.06%
4.6	247	25	9.19%
1.7	82	0	0.00%
1.9	95	3	3.06%
0.3	9	0	0.00%
0.7	47	2	4.08%
3.7	163	2	1.21%
55.0	2231	182	6.76%

Fig. 9. Table showing noise rate of the integer anomalies detected in some future Microsoft products.

Size in MLOC	Compile Time in mins.	Analysis Time in mins.
2.1	14:02	10:34
8.2	45:49	40:07
1.8	8:58	6:26

Fig. 10. Table showing the comparison of compile time and analysis time for the first three products.

6.2 Integer Range discovery

In 1978, Patrick Cousot developed a method to automatically discover linear restraints among variables of a progeam [9]. He statically determined linear equality or inequality relations between variables of a program and used this information for abstract interpretation and to aid optimizations like common subexpression elimination and constant propagation.

Program analysis algoritms often not only need to discover integer constraints but also need an efficient way to solve them. Zhengdong Su and David Wagner presented a set of novel polynomial time algorithms for solving some general classes of integer range constraints [18]. For the purposes of our analysis, we do not require to compute the actual integer range. We only determine whether the integer has a lower bound, upper bound or both.

6.3 Buffer Overrun detection

Our analysis detects integer overflows and underflows which are precursors to some serious buffer overruns. There have been other tools that focus only on buffer overrun detection like the one developed by David Wagner [22], espX and CGS [19]. CodeSurfer [11] is another code understanding tool that is used along with a constraint generator, taint analyzer and constraint solver to detect buffer overruns. Analysis for array bounds checking and buffer overrun detection require to relate the array or buffer to its size. Chin and Khoo have developed a generic framework for inferring the sizes of size-type input paramters to a function [8]. We either encode the knowledge of buffer allocation, access and sizes in our analysis or we communicate such information via annotations.

7 Conclusions

Integer anomalies in buffer size expressions result in buffer overruns, a serious security threat. In this paper, we have described a flow-insensitive static analysis algorithm to detect such critical integer anomalies in source code. The algorithm scales to millions of lines of production quality code and the time taken to analyze is comparable to the time taken to compile. The analysis has been successfully used to uncover and fix over 2000 integer anomalies in 50 MLOC, from future versions of several Microsoft products.

References

1. K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.
2. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

3. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
4. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7):775–802, 2000.
5. CA-2001-35 CERT Advisory.
6. CA-2003-10 CERT Advisory.
7. B. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy,*, 2002.
8. Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 62–72, 2000.
9. Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints between variables of a program. In *5th Annual ACM Symposium on Principle of Programming Languages*, 1978.
10. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.
11. Vinod Ganapathy et.al. Buffer overrun detection using linear programming and static analysis. In *CCS03*, 2003.
12. David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.
13. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, /2002.
14. Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.
15. Michael Howard and David LeBlanc. Writing Secure Code, Second Edition, 2002.
16. et.al. Jim Larus. Righting software. In *IEEE SOFTWARE published by IEEE Computer Society*, 2004.
17. MS04-025 Microsoft Security Bulletin.
18. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, 2004.
19. Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, 2004.
20. John Viega. Building Secure Software, First Edition, 2001.
21. John Viega. Secure Programming Cookbook for C and C++, First Edition, 2003.
22. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, 2000.
23. Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *10th ACM Conference on Computer and Communications Security*, 2003.