Using data flow analysis for detecting security vulnerabilities

Andrey Belevantsev and Oleg Malikov
Institute for System Programming
Russian Academy of Sciences
Russia, 109004, Moscow, B. Kommunisticheskaya, 25
Phone: (+7-095) 912-07-54, Fax: (+7-095) 912-15-24
{abel,malikov}@ispras.ru

Abstract. This paper addresses the issues of using data flow analysis approach for detecting certain types of programming errors in the source code. The primary motivation of this work is that programming errors like a missing array or buffer bounds checking may lead to security vulnerabilities in programs. The article is devoted to the detection of *buffer overflow*, *unchecked input usage*, and *memory leak* errors.

A new approach for static detection of these kinds of errors is described. The approach combines forward data-flow intraprocedural alias and integer range analyses as well as flow-sensitive and partly context-sensitive interprocedural analysis. The algorithm is based on the notion of an *abstract memory location* (AML). The rules for computing attributes of AMLs as well as number of heuristics used to improve the algorithm are discussed.

The suggested approach has been implemented for checking C programs. The paper reports on evaluating the tool with a number of open source programs.

1. Introduction

Security vulnerabilities is a very important class of programming errors. According to the reports of SecurityFocus.com more than 3000 vulnerabilities have been discovered during 1.5 year starting from 01/2000. The most common are buffer overflow (23% to 50% [10, 13]), format string and tainted input errors.

A great number of already known security errors originates from the C language unsafe features and its standard library functions, e.g. unsafe pointer and array dereferences, pointer arithmetic and string handling routines. As it was noted in [10], even "safe" string primitives, such as `strncpy()`, encourage possible off-by-one bugs.

There are several approaches to classify the software security errors. A profound analysis of the fundamental taxonomies is given in [1] and [5]. We accept a widely used definition of the security vulnerability as a program error that causes incorrect program operation if a user feeds specially crafted data into the program. We distinguish between errors and corresponding vulnerabilities focusing on programming errors detection in particular. It often is a difficult problem to qualify an error as the security vulnerability. Moreover, an error, which is not vulnerable, still may turn to be a severe bug in the code.

It is known that software vendors allocate much resources to audit their code base for various kinds of possible security vulnerabilities. Manual code inspection and fixing is an expensive and time consuming solution to the problem. Existing publicly available tools for the automatic detection of programming errors are either not designed especially for finding security flaws, thus missing to report important error classes, or yield too many false positives (more than 90%). Several companies claim to possess such tools [14, 11], but use them internally in their auditing services. Many researchers are making a considerable effort towards automated static analysis tool for detecting security flaws with a relatively low number of false warnings [6, 10].

There is a number of tools and libraries, which are intended to detect attempts to exploit vulnerabilities at run time, as well as a number of hardware features aimed at protecting the system from hacking attempts. However, these features can either be bypassed or incur unacceptable running cost. The preferable way of anti-hacking protection is to fix all possible security vulnerability errors in the source code.

The source code programming errors can be detected either during static analysis or at run time during debugging, testing, etc. To detect programming errors during debugging or testing one has to provide input data that spots an error, which is a challenging task mainly because the exploitation of security vulnerabilities often needs non-trivial input data. For these reasons static analysis approach should be used.

Static analysis approach has its own limitations. Namely, it is impossible to detect all the errors in a program without false warnings. The false positives to the total number of detected errors ratio is a measure of quality of the static analysis algorithm. In our opinion, in order to achieve the best quality of analysis the algorithm should model operation of the program as close as possible. This includes tracking use of program data memory, memory allocations, buffer accesses as well as tracking integer calculations in the program, for example, string lengths or array indices. The semantics of the standard library functions is also to be modeled.

1.1. Overview

The goal of our research is to develop new methods for automatic detection of various kinds of programming errors in existing (or legacy) programs. Another goal is to develop a tool for automatic detection of programming errors using this new approach. The tool should be applicable to real-world programs (hundreds and thousands of KLOC) and have reasonable false positives to the total number of detected errors ratio.

In this paper we concentrate on the buffer overflow, format string and memory leak programming errors. **Buffer overflow** errors occur as a result of missing or insufficient array bounds checking. **Format string** errors are results of using the unchecked input data as an actual parameter passed to the formatting input/output functions of the C standard library (`printf`, `fprintf`, etc). **Memory leaks** occur when a heap object is not deallocated and all the pointers to that object are lost in the running program.

The main contribution of the paper is a new method for finding certain kinds of programming errors, which combines interprocedural range and alias analysis. We show that the method can be practically used for error detection by implementing it in a prototype tool. In our opinion this is the first attempt to apply such combination of static analysis methods to the problem of automatic detection of programming errors.

Our approach uses the notion of *abstract memory location* (AML). An AML represents any object in memory, allocated either statically (variables, string constants) or dynamically (heap memory). Each AML possesses a number of attributes that reflects its properties useful for error detection. These attributes include, e.g *points-to* set (i.e., the set of AMLs to which the AML may point to); *size* (amount of bytes that occupied by the AML); or *length* (the length of the string stored in the AML).

The error detection process consists of two stages. In the first stage a classical iterative forward data-flow analysis algorithm [7] is used. The algorithm calculates the attributes of AMLs in every program point. It consists of several components as follows:

- Flow-sensitive intraprocedural analysis. This component makes both alias and integer range analysis, depending on the attributes of an AML it is calculating.
- Partially context-sensitive interprocedural analysis. We use a number of experimentally established tradeoffs to regulate context sensitivity of the analysis.
- Support for the major functions of the C standard library. The purpose of this part of the algorithm is to provide the information about the impact of the operational environment on the analyzed program.

The second stage performs verification of the attributes and issues warnings when certain conditions are not satisfied. The following warnings can be reported: buffer overflow, format string error, memory leak, use of undefined pointer value, dereferencing `NULL` pointer.

We have implemented the approach in the prototype tool for checking C programs. The preliminary evaluation of the tool on a number of open source programs shows promising results. Namely, in 4 packages we've analyzed the true positive rate ranges from 22.6% to 70.6%.

The rest of the paper is organized as follows. Section 2 introduces basic concepts of our analysis. Section 3 follows with a detailed description of the algorithm for attribute calculation. Section 4 describes the process of warnings generation. Section 5 continues with evaluation results of the implemented tool, Section 6 discusses related works. Section 7 gives the conclusion.

2. Basic concepts

In this section we describe the fundamental concepts and data structures used in our approach. First we define the lattices applied for tracking all integral computations performed in the program. We do not support floating-point calculations, because they rarely result in buffer overflow errors. Then we continue with a detail description of abstract memory locations and data structures concerned.

2.1. Data lattices

To model the integral types of the C language we use the integer lattice. The value set of the lattice consists of unlimited negative and positive integer numbers, $-\infty$, ∞, **undef**, **any**, and integer ranges $[l, h]$, $(-\infty, h]$, $[l, \infty)$, where l, h are integers. The integer lattice type is further denoted as $M\_Integer$.

The approach of using integer ranges for tracking integral values in the program is also used in [3, 10]. Considering [10], we add "undefined" value **undef** and "overdefined" value **any** to the value set of the lattice. The **undef** value is assigned to a local variable at the location of its definition, if the definition has no initialization. The **any** value means that the value of the variable is completely indeterminable by static analysis algorithm. By definition $(-\infty, \infty) \equiv$ **any**.

We define the standard set of operations with integers and extend them to handle **undef** and **any** values. We also define the join ($\sqcup$) binary operation used in program locations where the control flow merges from several alternatives. We follow the [3] definition of join operation of two integral ranges as *bounding box* of these ranges.

2.2. Abstract memory locations

Each object, which may potentially exist during program execution, has the corresponding abstract memory location (AML) object during program analysis. AMLs for objects are created "lazily" when the corresponding variable declaration point of the program is analyzed. AMLs are created for static and automatic variables, string constants, dynamic data structures (allocated by `malloc()` and friends)and temporary variables of the intermediate representation of the program.

An AML is created for an aggregate type as the whole, as well as AMLs for individual fields of the aggregate type. These AMLs are linked together using the *overlap* AML attribute. For an array object two AMLs are created, the first represents the pointer to the array, whereas the second is the array itself. This allows us to handle array and pointer dereferences uniformly. Array elements are not distinguished and no separate AMLs are created for array elements. This is a trade-off decision to decrease resource and time consumption of the analysis. However, our experiments show little impact on the quality of analysis.

We use *(AML, offset)* to refer to an AML with the offset *offset*. Offsets have the *M_integer* type. By $AMLSet = \{(AML, \mathit{offset})\}$ a set of such pairs is denoted.

An AML contains both static and dynamic attributes. The values of static attributes remain unchanged during the lifetime of the AML, whereas dynamic attributes are associated with some point of the program and make sense only in conjunction with it. There is a special AML **null**, which is referenced by any `NULL` pointer.

The static attributes of an AML are listed below:

- *size* is the amount of bytes occupied by the AML. For dynamically created AMLs the size is calculated using the corresponding parameter(s) of the memory allocation function, when it is possible. In this case the size may change during iterations of the analysis, as well as after the call to `realloc()`. In other words, the size of the AML is the same for all the points of the program, but may be adjusted dynamically.
- *overlap* is an *AMLSet* that contains the AMLs that overlap the current one. The attribute is used for specifying the layout of subobjects in the object of the aggregate type. Creation of subobject AMLs is done implicitly during creation of an aggregate AML, but this set can also be defined more precisely among the iterations. It happens due to impossibility of the correct determination of subobjects of dynamically created aggregate AML.
- *label* denotes the name of the AML.
- *type* denotes the type of the AML.
- *var* denotes the original variable that initiated the creation of the AML.

The dynamic attributes of an AML (denoted by *AMLAttributes*) are as follows:

- *aset* is an *AMLSet* of the objects which the AML may point to (i.e, points-to set).
- *len* is *M_integer* denoting the length of the string contained in the AML.
- *value* is *M_integer* denoting possible values of AMLs of integral types.
- *input* is a boolean attribute that shows whether the AML is "tainted", i.e. its value depends on user input.
- *alive* is a boolean attribute denoting that the memory allocated for this AML was not freed.

Note that the dynamic attributes of an AML are always bound to a specific program point.

3. Calculation of attributes

This section contains detailed description of the rules for an AML attribute calculation. Both intra- and interprocedural stages of the algorithm are discussed.

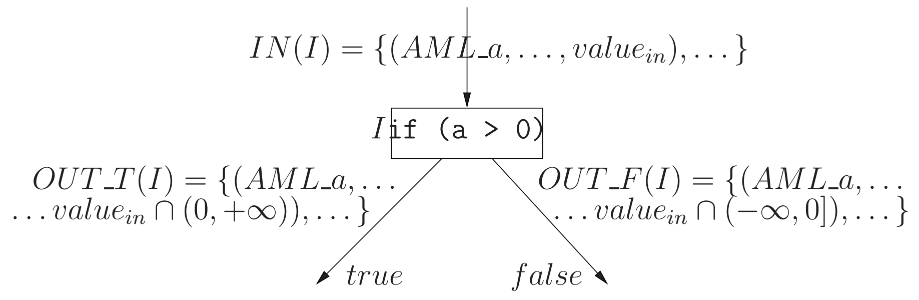
$IN(I) = \{(AML\_a, \ldots, value_{in}), \ldots\}$
I if (a > 0)
$OUT\_T(I) = \{(AML\_a, \ldots \ldots value_{in} \cap (0, +\infty)), \ldots\}$
$OUT\_F(I) = \{(AML\_a, \ldots \ldots value_{in} \cap (-\infty, 0]), \ldots\}$
$true$
$false$

Figure 1: An example of the flow function calculation

There is also a number of heuristics used to improve speed and convergence of the analysis presented.

3.1. Intraprocedural analysis

The algorithm operates on the medium level of the intermediate representation of the program, generated by the C front-end. We use medium-level representation in form of quadruples $\langle opcode, res, op_1, op_2\rangle$. Structured statements are converted to conditional and unconditional gotos. So-called *pseudoregisters* are used to hold temporary values.

The algorithm annotates each instruction I of the program with additional attributes. We denote the type of these attributes as *AML2Attr*. Each attribute of the type *AML2Attr* is a set of pairs $\{(AML, AMLAttributes)\}$. The instruction attribute $IN(I)$ is a set of attributes that are available at the point before the instruction I. If I is a single-exit instruction, $OUT(I)$ is the set of attributes available at the point after the instruction I.

When I is a conditional jump instruction, two sets *OUT_ T(I)* and *OUT_ F(I)* are defined and they contain the attributes effective when the condition holds and when the condition fails respectively. Similarly, if the instruction I has several exits, all of them have different attribute sets. In the Fig. 1 an example is given. AML_a is an AML corresponding to the variable a. At the entry of the instruction I this AML has the attribute $value = value_{in}$. At the exits of I the attributes will be as follows: $value_{out\_t} = value_{in} \cap (0, +\infty)$, $value_{out\_f} = value_{in} \cap (-\infty, 0]$.

$IN(I)$ and $OUT(I)$ (*OUT_ T(I)*, *OUT_ F(I)*) are connected by the flow equation:

$$\begin{aligned}
&OUT(I) = Flow(I, IN(I)), \text{ or}\\
&(OUT\_T(I),\ OUT\_F(I)) = Flow(I, IN(I))\\
&IN(I) = \bigsqcup_{p \in Pred(I)} OUT(P)
\end{aligned}$$

Joining of *AML2Attr* sets is performed as follows. For each $x = (AML, Attrib1)$ taken from $OUT(P)$, if $IN(I)$ does not contain $y = (AML, Attrib2)$ with the same AML, x is added to $IN(I)$. Otherwise, $Attrib1$ and $Attrib2$ are joined.

The flow function is defined for each instruction of the intermediate representation and is the core of the attribute calculation algorithm. For example, access via pointer instructions modifies *aset* and/or *len* attributes, arithmetic instructions modify *value* attributes, etc. Conditional jump instructions also modify attributes.

The function call instructions are handled in a special way. If the function being called is a standard library function (for example, `strcpy`), a special routine is called, which simulates the behavior of this library function and generates the output attribute set $OUT(I)$ from the current input attribute set $IN(I)$, i. e. handling of the standard library functions is always context sensitive. If the function is not a predefined function, yet its body is available, the analysis of the function is not performed immediately, and the current output attribute set $OUT(I)$ is used. It is the duty of the interprocedural analysis framework to analyze the function and to update its output attributes. If the callee is not a predefined function and its body is not available, the analysis algorithm has to use conservative assumptions concerning the function semantics.

Special support for loops is also implemented in order to ensure algorithm convergence and improve its precision. The loop counter variable is detected and its value range is set according to loop condition. All the further changes of the attributes of the loop variable are prohibited. This heuristics work for loops of simple structure, and still complex loops may not converge. A number of other tricks is used in this case.

3.2. Interprocedural analysis

A context-insensitive program call graph is constructed at the beginning of the analysis. Initially the call graph does not contain edges corresponding to call-by-pointer instructions, such edges are added when the corresponding points-to sets are computed.

The current implementation features only partially context-sensitive interprocedural analysis. On each iteration each function is analyzed only once with joint context (join of all the input attribute sets at all the call sites), thus the analysis is mostly context-insensitive. However, calculation of the flow function for a call instruction does some additional passes to make analysis more sensitive. For example, all the dynamic AML attributes, which cannot exist at the current path in the call graph, are killed.

The interprocedural analysis uses the worklist approach. Initially the worklist of functions to be analyzed contains all the dangling nodes of the call graph, i.e. all the functions which are not called from any other function. On each iteration each function f in the worklist is analyzed once as follows:

1. All the input sets $IN(I)$ of all the call sites of f are joined. Let IN_1 be the result of join operation.

2. All the pairs $(AML, Attributes)$ in IN_1 are replaced with $(AML_1, Attributes)$. Here AML corresponds to an actual argument of the function at some call site, AML_1 corresponds to the respective formal parameter of the function. If several AMLs map onto the same AML_1, their attributes are joined and the result is assigned to the attributes of AML_1. The attributes of the actual parameters without an AML (for example, constants) are also joined to the attributes of the corresponding formal parameters. Let IN_2 be the resulting set of attributes.

3. The pairs $(AML, Attributes)$ are removed from IN_2 for all the AMLs, which are not directly or indirectly accessible from the current scope. Such attributes are useless during the analysis of f. Let IN_3 be the result of the operation.

4. Forward iterative intraprocedural data-flow analysis is performed with a set of input attributes IN_3 for the f function.

5. The pairs $(AML, Attributes)$ are removed from the output set of attributes of f for all the AML, which are not directly of indirectly accessible outside the f.

6. If the intraprocedural analysis performed at the step 4 has changed attributes of any instruction within f, then all the callers of f are marked to be reanalyzed on the next iteration. A callee of f is marked to be reanalyzed, if input attributes of the corresponding call instruction have changed.

3.3. Additional analysis heuristics

We have implemented a number of additional analysis heuristics to increase the analysis speed, reduce memory usage, and improve the precision of the analysis. These features are listed below.

- Loop widening and narrowing [3] is performed to ensure convergence of the analysis in typical situations.

- Context clipping is performed at the points of conditional jump instructions. The context is updated on both exits according to the condition tested.

- Attributes of the source and the destination AMLs of *transfer* instructions are linked instead of copying. A transfer instruction assigns attributes of a source AML to attributes of a destination AML without changes. This feature allows effective cutting of AML attributes later.

- The exits of the function are distinguished. The contexts at the function exit points may differ from each other providing different attribute sets to the callers. When the result of the function call is tested, the condition is propagated back to the function and the appropriate return statement is chosen, thus selecting specific attribute set. This is a trade-off, which allows increasing context-sensitivity without impacting on the performance.

  The code snippet below is an example of this heuristic. On the “then” branch of `if` (line 14) the value of `p` is undefined, but on the “else” branch (line 15) `p` points to the object of size 10.

```c
int f(char ** p) {
    char * pp = malloc(10);
    if (pp == NULL)
        return 0;
    else {
        *p = pp;
        return 1;
    }
}

int main() {
    char *p;
    if (f(&p) == 0)
        return 0;
    p[1] = 0;
    ...
}
```

4. Error detection

In two previous sections we described the attribute computation algorithm. The programming errors are detected using the computed AML attributes. For each instruction of the program the input context is checked and the attribute values, which can indicate programming errors, are determined.

Buffer overflow warning messages are generated as a result of comparing *size* attribute of the destination AML with *offset* attribute of pointers to this AML. If the offset is negative or greater than the size of the AML, buffer overflow warning is issued.

Detection of memory leaks is implemented as follows. At the end of each function all the pairs $(AML, AMLAttributes)$, which have *alive* attribute set to *true*, are checked. AML must correspond to some dynamically allocated data structure. If such pair is to be removed from the output sets according to step 5 of interprocedural analysis, then all the points of memory allocation for this AML are marked as potential sources of memory leaks, and the corresponding warnings are issued.

Potential format string errors are detected at the call sites of the corresponding library functions. The format argument of the function is checked for taintedness. The argument is tainted, if the *input* attribute of the corresponding AML is set to *true*. In such cases warning messages are also issued.

Dereferencing of undefined or possibly `NULL` pointers is also checked and appropriate warning messages are generated.

5. Experimental results

We have implemented our approach as a prototype tool for checking C programs. The tool uses the C compiler and is implemented in the framework of the Integrated Research Environment (IRE) developed at Moscow State University. The compiler translates C code to the medium-level intermediate representation. The IRE contains a number of standard and original analysis and transformation algorithms. All of them operate over the intermediate representation. The IRE provides graphical user interface and project support.

For understanding features of our prototype implementation, consider this example.

```c
#define BSIZE 32

void copy_string(char *dst, const char *src,
  int *cur, unsigned cnt) {
  int i = 0;
  for (; i < cnt; i++)
        dst[*cur + i] = src[i];
  *cur += i;
}

void fix_middle(char *dst, int *cur) {
  dst[(*cur)++] = '␣';
  dst[(*cur)++] = '=';
  dst[(*cur)++] = '␣';
}

void do_cat(char *dst, const char *src1,
  const char *src2, int *cur, unsigned cnt) {
  copy_string(dst, src1, cur, cnt);
  fix_middle(dst, cur);
  copy_string(dst, src2, cur, cnt);
}

int main(void) {
  char str1[BSIZE];
  char str2[BSIZE];
  char *str3;
  int i = 0;
  j = sizeof(str1) + sizeof(str2) + 3;
  str3 = (char*) malloc(j);
  if (str3) {
    do_cat(str3, str1, str2, &i, sizeof str1);
    printf("%d␣%d\n", i, j);
    str3[i] = 0;
  }
  return 0;
}
```

Table 1: Evaluation results for our tool

Application	Source LOCs	True positives	Total warnings	% of true positives
bftpd-1.0.24	3389	19	60	31.7%
lhttpd-0.1	1141	7	19	36.8%
pcre-3.9	9102	7	31	22.6%
surfboard-1.1.8	823	12	17	70.6%

It contains buffer overflow in line 32. Index variable `i` is changed through calling `do_cat()` function when passing its address as a parameter. Both calls of `copy_string()` increase the variable by `BSIZE`, and `fix_middle()` increases it by 3. Then the value of this variable equals the allocated size for `str3` buffer. Thus, an off-by-one error occurs. Note that neither FlexeLint [15], nor ITS4 [9] and RATS [12] detect this error.

We delivered preliminary evaluation of the tool on a number of open source programs. Several examples are taken from [6]. Evaluation results are summarized in the Table 1. For each package we show the size of its source code as reported by the `wc` tool, total number of warnings generated, a number of true positives and the percentage of true positives.

There is relatively high percentage of false positives due to complex evaluations of array indices, which our analysis is not yet able to cover. Other major causes of false warnings include insufficient handling of calls by pointer, insufficient standard library support and imprecision of the analysis in certain situations.

6. Related work

There are many methods and tools applied for automatic detection of programming errors. We have only mentioned approaches based on program verification (see [8] for an overview) and run-time checking such as [4]. Below we focus on the existing static analyses and tools.

There are tools that use lexical source code analysis to detect security errors (for example, [9, 12]). The main advantage of such tools is the speed of the analysis. However, lexical analysis tools lack accuracy and gives a very large number of false warnings.

Static analysis tools of the `lint` family such as Splint and FlexeLint [16, 15] use semantic analysis and simple forms of data flow analysis to detect many common C coding errors. The tools can detect buffer overflow and format string errors in simple situations, but more complex situations are either missed or yield false positives.

Bush et al. [2] use static analysis for finding common dynamic programming errors, such as dereferencing `NULL` pointers, memory leaks, etc. The analyzer simulates execution paths by modeling a memory state of a program and identifying inconsistencies. Models for system library functions are provided with the analyzer, whereas models for user functions are generated from the source. The tool does not aim at finding security errors such as buffer overflow and format string errors, whereas our approach is designed to detect these kinds of errors too. Both approaches detect memory leak and `NULL` pointer dereferencing errors.

Wagner et al. [10] develop static range analysis technique for finding buffer overflows. The approach consists in generating constraints for modeling string and buffer operations and then solving the constraints with range analysis. The tool implemented finds one real error in 10 warnings generated. The main limitations of the approach are insufficient handling of aliases (both pointers and unions) and flow- and context-insensitivity. Our approach handles pointer operations with flow sensitivity. That allows us to detect buffer overflows resulting from primitive pointer operations, which is not performed in [10]. However, we use some ideas from [10] and [3] for our implementation of integer range analysis.

Livshits and Lam [6] proposed an alias analysis to be used expressly for error detection. The approach introduces a new representation, called IPSSA, for capturing pointer dereferences and procedure calls in SSA form. The algorithm presented is a hybrid approach, using context- and path-sensitive analysis in the key situations and fast imprecise analysis for all other references. However, error detector used in [6] is limited to discover buffer overflows resulting from putting user supplied string to a static buffer. Our approach performs tracking of integral values of array indices and string lengths. That allows us to detect buffer overflows resulting from handling strings as character arrays, which was not the aim of [6]. Nevertheless we think that the assumptions we have made to speed up the analysis and suppress false errors are to some extent similar to those of [6].

7. Conclusions

This paper describes an approach for using data flow analysis for detecting programming errors, especially those error kinds, which may lead to security vulnerabilities (buffer overflow and unchecked input usage errors). Our algorithm is based on flow- and partially context-sensitive interprocedural alias and integer range analyses.

We use our approach in a tool for automatic detection of security vulnerabilities, which is currently at the stage of prototype implementation. Preliminary evaluation of the tool yields the true positive rate from 22.6% to 70.6% on 4 tested packages, showing feasibility of our approach.

We may also state that the assumptions we have made to improve the precision of the tool match the semantics of many functions analyzed.

Future challenges for us include scaling up of the analysis to larger programs, as well as eliminating known drawbacks and further improving the precision of the tool. We also plan to implement the support for annotation language and perform thorough evaluation of the features introduced by annotations.

References

[1] T. Aslam. A Taxonomy of Security Faults in the Unix Operating System. M.S. thesis, Purdue University, 1995.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Proceedings of Software Practice and Experience*, pages 775-802, 2000.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[4] E. Haugh and M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2003

[5] I. Krsul. Computer vulnerability analysis. Ph.D. thesis, Purdue University, West Lafayette, IN, May 1998.

[6] V. Benjamin Livshits and Monica S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, September 2003.

[7] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997, 3rd ed.

[8] K. Rustan, M. Leino. Extended Static Checking: a Ten-Year Perspective. In *Proceedings of the Schloss Dagstuhl Tenth-Anniversary Conference*, Springer-Verlag, 2001

[9] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[10] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of 7th Network and Distributed System Security Symposium*, Feb. 2000.

[11] @Stake, `http://www.atstake.com`

[12] RATS checker, `http://www.securesoftware.com`

[13] Bugtraq mailing list and vulnerability database, `http://www.securityfocus.com`.

[14] Secure Software, `http://www.securesoftware.com`

[15] Flexelint static checker, `http://www.gimpel.com`

[16] Splint static checker, `http://www.splint.org`