Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking

Lei Wang, Qiang Zhang, PengChao Zhao
Computer School, Beijing University of Aeronautics and Astronautics, China
wanglei@buaa.edu.cn, zhangqiang_buaa@cse.buaa.edu.cn, zhaopengchao@sse.buaa.edu.cn

Abstract

Ensuring the correctness and reliability of software systems is one of the main problems in software development. Model checking, a static analysis method, is preponderant in improving the precision of vulnerabilities detection. However, when applied to buffer overflow and other bugs, it is hard to automatically construct the model for detecting the vulnerabilities. To address this problem we propose an approach that combines constraint based analysis and model checking together. We trace the memory size of buffer-related variables and instrument the code with corresponding constraint assertions before the potential vulnerable points by constraint based analysis. Then the problem of detecting vulnerabilities is converted into the problem of detecting vulnerabilities to verifying the reach ability of these assertions by model checking. In order to reduce the cost of model checking, program slicing is introduced to reduce the code size. CodeAuditor is a prototype implementation of our approach. With CodeAuditor, several yet unreported vulnerabilities are discovered in several open source software, and the performance is shown to be improved significantly with the help of program slicing.

1. Introduction

The rapid increase of our dependence on computer systems means increasing interest in attacking those systems. Buffer overflow is a programming error that can cause vulnerability, and it occurs when data written to a buffer, due to insufficient bounds checking, corrupts data values in memory addresses adjacent to the allocated buffer. An exploitable buffer overflow is capable of rendering a computer system totally vulnerable to the attacker. It is thus a major concern of the computing community to provide a practical and efficient solution to discovering and removing these vulnerabilities.

Software vulnerability detection can be implemented either statically [1] [2] [3] or dynamically [4][5]. Static techniques can be divided into general static method and formal verification method based on sound theory. The former is based on program analysis, while the latter is based on formal logic and theory of automata, which can be used to prove that a program correctly satisfies a given property. However, the number of admissible states with the dimensionality of state space is often extremely large and increases exponentially. A practical method for software model checking based on abstract-verify-refine paradigm, which can abstract code not relevant to the property and limit the scope and thus simplify the problem. This typically not only increases the number of vulnerabilities to be detected but also decreases the number of false alarms.

However, single model checking can not be used to detect vulnerabilities conveniently. For example Blast [6] detects the NULL pointer dereferences bugs with the help of CCured [7], and it can not check the security of complex operations of buffers. In this paper we trace the length of buffers via constraint based analysis, instrument corresponding constraint assertions before the potential vulnerable points, and thus convert detecting vulnerabilities to verifying the reachability of these assertions by model checking. But the main problem of model checking is the combinatorial explosion of system states, so we use program slicing on the instrumented code in order to reduce the size of program. CodeAuditor is a prototype tool of our approach, which includes a front-end preprocessor and a back-end Checking module. The preprocessor also instruments the abstract syntax tree (AST) of GCC and directed by XML Configure-File, which specifies the pattern of instrumentation for buffer overflow, format string and code injection. In the back-end, we use a model checking tool Blast to verify the reachability of assertions. Using CodeAuditor, some yet unreported vulnerabilities are discovered in several open source software, and the performance is improved enormously when program slicing is used.

The remainder of this paper is organized as follows. Section 2 introduces the technologies and schemes including program analysis and program slicing. We present the detail of constraint based analysis in Section 3. Model checking is described in Section 4.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.24

IEEE computer society

Section 5 gives a brief description of CodeAuditor. We analyze a sample test case to illuminate our scheme in Section 6 and our experiments are presented in Section 7. Section 8 introduces related work and we conclude in section 9.

2. Static analysis and instrumentation

In this section we describe those technologies used in our tool, including constraint based analysis, instrumentation, alias analysis and program slicing.

2.1 Constraint based analysis

We introduce the syntax-directed method to analyze the AST of GCC and create an extension constraint based analysis for buffer overflow, which can be easily applied for other bugs, such as format string and code injection. We define a formalization framework for inserting assertions as follows:

(1) Buffer facts are modeled as pairs of integer ranges including the *max length* and *used length* of each buffer, and we call these two integers the attributes of buffers.

(2) Model the statement and function related to buffers as integer transfer functions and constraints.

Based on these two principles we establish our framework of constraint based analysis [8][9] for code-instrumentation, which will be described in section 3. With this method we add the instrumentation code to source code, and do not change the correctness of syntax and semantic. In the end we convert the buffer overflow problem to a reachability problem which can be checked directly through model checking.

2.2 Instrumentation based on AST directed by XML Configure-File

The important phase in our approach is adding instrumentation to the source code. The instrumentation is directed by the particular model described by XML Configure-File [10]. Different models can be used to check for corresponding correctness properties. Considering that users are likely to build their own models, this phrase is designed in a compatible way to support both simple models and general ones. This paper deals with three types of vulnerabilities including buffer overflow, code injection and format string, and we describes this model using XML Configure-File, which is used to guide the instrumentation.

The phase of instrumentation was based on AST of GCC, and the elements of AST that include the function AST node and global list. The former can be divided into local variable list and statement list. During the instrumentation phase, we traverse the AST, parse the XML Configure-File, match the right pattern in special modules and execute instrumentation operation. The instrument is accomplished in one pass of the AST of the source code, and the output is the instrumented code.

2.3 Alias analysis

Two pointers are said to be aliased if they point to the same location in memory. Pointer alias analysis is difficult because there are many potentially aliased pointers in a program, and it is also why the alias analysis is important in our instrumentation. In order to improve the precision of detection, we introduce the alias analysis during the instrumentation, and the attributes of alias variables should be updated simultaneously. Based on flow-sensitive SSA-based pointers-to analysis on GCC AST, for every concerned variable we compute their must-alias at every program location, and update their attributes at the same time. In section 6, we give an example including alias, whose attributes are updated simultaneously.

2.4 Program Slicing

In this paper we use program slicing [11][12] to extract the statements which are relevant to the vulnerabilities in the program. Given a source program *P*, slicing starts with a criterion *SC(L) = (L,V),* Where L is a vulnerable location in the program and *V* is a set of attributes of buffers in the program. We say *S* is a slice of program *P* for criterion *C* if *S* is derived from *P* by eliminating irrelevant statements from *P* such that S is syntactically correct.

After the instrumentation phase, we implement the algorithm of program slicing [13] based on the AST of GCC to eliminate the unnecessary statements. The user can active this algorithm by using a command-line switch. The essence of the slicing algorithm is as following: starting with the statement specified in the slicing criterion, it includes each predecessor that assigns a value to any variable in the slicing criterion and generates a new slicing criterion for the predecessor by deleting the assigned variables from the original slicing criterion and adding any variables referenced by the predecessor. The following definitions [14] are helpful in understanding how program slices are constructed:

Defs(n): The set of variables defined assigned to at statement *n*.

Refs(n) :The set of variables referenced at statement *n*.

Req (n): A set of nodes that is required to also be included in a slice along with node *n*.

Our target is updating the slicing criterion and confirms if the current statement should be deleted from the statement list. Suppose the statement *n* is a predecessor of statement *m*, and the corresponding slicing criterion is *SC(m)* and *SC(n)*, then we have:

(1) If $\forall v \in V, v \notin defs(n)$ then the statement n should be deleted, and we need not update *SC(m)* and the new criterion *SC(n) = SC (m)*;

(2) If $\exists v \in V, v \in defs(n)$, we should update the slicing criterion:

$$SC(n) = F_i(n, SC(m)) \qquad i \in [1,6]$$

Where F_i represents the function for difference statements. Here we can consider the following language features: expression statements, compound control statements, structure variables, indirect assignment by pointer, indirect reference by pointer, dynamic structures, references to structure members by pointer, assignment to structure members by pointer, procedure call.

For space reasons, only assignment and compound control statements are discussed in this paper, and other features left unspecified here.

For expression statement n, a predecessor of statement m if $\exists v \in V, v \in defs(n)$,we should update the slicing criterion as follows:

$$SC(n) = (SC(m) - defs(n)) \cup (\cup S_{<n,x>} \forall x \in refs(n)).$$

This means that we should delete the variable $defs(n)$ in $SC(m)$ and combine the new criterion with $\cup S_{<n,x>} \forall x \in refs(n)$.

For Compound control statements (if else, while, switch), we should update the slicing criterion as follows:

$$SC(n) = (SC(m) - defs(n)) \cup \left(\bigcup_{x \in refs(n)} S_{<n.x>} \right) \cup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in req(n)} S_{<k,y>} \right)$$

The difference between compound control statements and expression statement is that the former includes the set of required statements for statement n, *req(n)*, whenever statement n is included in a slice.

This algorithm only continues to iterate all the statements one time. In the worst case, the algorithm takes *O(s)* time where s is the number of statements. The result of this phase is a list of definitions (and statements) in the slice.

3. Constraint Based Analysis

We model buffers as a pair of integer ranges, including the number of bytes allocated for the buffer (*s.max*), and the number of bytes currently in use (*s.used*), and model each string operation in terms of its effect on these two attributes. So the safe property to be verified is: *s.max >= s. used* for all string variables *s*. We use the lattice of framework from [8].

Let $\mathbb{Z}$ denote the set of integers and write $\mathbb{Z}^\infty = \mathbb{Z} \cup (-\infty, +\infty)$ for the extended integer, and the range is a set $R \in \mathbb{Z}^\infty$ which has the form of $[m,n] = \{i \in \mathbb{Z}^\infty : m \le i \le n\}$. We define the complete lattice on $\mathbb{Z}^\infty$: L=$(\mathbb{Z}^\infty, \perp, T, \subseteq, +, -, \times, max, min,)$, where $\perp$,T is the elements of $\mathbb{Z}^\infty$,and $\subseteq$ as the partial order, *min* and *max* is the operator on $\mathbb{Z}^\infty$, $+,-,\times$ is the binary operators on $\mathbb{Z}^\infty$ which are the extension for the usual arithmetic operators.

For every string, we define constraint assertions on string S: *assert(S.max>=S.used)*, an assignment $\alpha : v \to \alpha(v) \in \mathbb{Z}^\infty$ satisfies a constraint system. In other words a potential vulnerability is safe if all the constraint assertions are true when the variable are replaced by the corresponding values $\alpha(v)$. The least solution to the constraint system is the smallest assignment α that satisfies the system. In this paper, the solution of this constraint system was left to the model checking.

Our analysis is based on the AST in GCC compiler. And it traverses the AST of C source code first, and generates an integer constraints system. We only focus on the string and integer variables, including pointer and array. For the relevant statement in the input program we generate integer range constraint for buffer operation by matching the node of the XML Configure-File, and determine what kind of constraint should be created. Some constraints are listed in the Table 1. The left column shows the C code we are interested, and the right column shows the generated constraints and assertions. For example, when we parse a statement s*trncpy(dst, src, n)* that matches a pattern in XML Configure-File, a constrains

(*min(src.used, n)* $\subseteq$ *dst.used*)

and an assertion

(*dst.max>= min(src.used, n))*)

Table 1. Some string operations and it's constraints and assertions

C Code	constraints and assertions
char *p	0 $\subseteq$ p.max ; 0 $\subseteq$ p.used
char a[n]	n $\subseteq$ a.max; 0 $\subseteq$ a.used
p = malloc(n)	n $\subseteq$p.max; 0 $\subseteq$ p.used
p = strdup(s)	s.max $\subseteq$ p.max; s.used $\subseteq$ p.used
p[n] = x	*assert*(p.max >= n);min(p.len, n+1) $\subseteq$ p.used
strcpy(dst, src)	*assert*(dst.max >= src.used); src.used $\subseteq$ dst.used
strncpy(dst, src, n)	*assert*(dst.max >= min(src.used, n)); min(src.used, n) $\subseteq$ dst.used
strcat(s,t)	*assert*(s.max >= s.used + t.used); t.used + s.used $\subseteq$ s.used
strncat(s,t, n)	*assert*(s.max >= s.used + n); s.used + n $\subseteq$ s.used
scanf("%ns",str)	*assert*(str.max >= n); n $\subseteq$ str.used
sprintf(dst, "%s", str)	*assert*(dst.max >= str.used);str.used $\subseteq$ dst.used
sprintf(dst, "%d", n)	*assert*(dst.max >= 20); 20 $\subseteq$ dst.used

are generated. The constraint means that the *dst.used* should be replaced by *min (src.used, n)*, and the assertion means that if *dst.max>= src.used* is *true* then this is a bug.

In this paper, we also model user-define function calls which guarantee that the analysis is context-sensitive. For example, supposing that char **foo* (char *s) is a prototype of user defined function, we need to add variables *foo_ret_max* and *foo_ret_used* as the return values of *foo*, and we also add *foo_s_max* and *foo_s_used* as attributes of the parameter *s*. All of these variables are added as the global variables. The attributes of the parameters should be assigned when the function is called, and the attributes of the return value will be assigned when the function returns. In section 6 we give an example to describe this process.

Unlike the reference in Java, pointers in C can point to the middle of a buffer, so pointers are the trouble spots for program analysis. And any code fragments that manipulate buffers using pointer operations are very difficult to analyze. Our tool has not implemented deep pointer analysis, and we cannot generate constraints for doubly-indirected pointers.

For example:

```
1    int * a = (int *)malloc(sizeof(int)*10);
2    int ** b = &a;
3    *b = (int *)malloc(sizeof(int)*5);
4     a[6]  = 20; //buffer overrun
```

Updating the attributes of **b* in the 3rd line should update the attributes of array *a* at the same time, as *a* is an alias of **b* and they point to the same location of memory, here *b* is a doubly-indirected pointers, and we do not deal with it.

Function pointers, arrays of pointers and array of user-define structure are currently ignored, but we can handle the single variable of user defined structure. These simplifications are all unsound in general, but still useful for a large number of real programs.

4. Model Checking

In this paper our reachable problem is verified by model checking. Model checking is an algorithmic technique to verify a system description against a specification [15][16]. Given a system description and a logical specification, the model checking algorithm either proves that the system description satisfies the specification, or reports a counterexample that violates the specification. The input to the software model checker is the program source (system description) and a temporal safety property (specification). The output of the model checker is ideally either a proof of program correctness that can be separately validated [17], or a counterexample in the form of a specific execution path of the program. In the case of buffer overflows, P is a property indicating that every access to a specified buffer is safe. Using standard compiler analyses, all pointer dereferences and array accesses are instrumented with bounds checks, so the system goes to a line labeled ERROR if a bound is violated. The buffer overflow problem is thus converted to check the reachability of the line labeled ERROR.

The main problem of model checking is the combinatorial explosion of system states. Abstraction methods attempt to reduce the size of the state space. Counter example-guided abstraction refinement

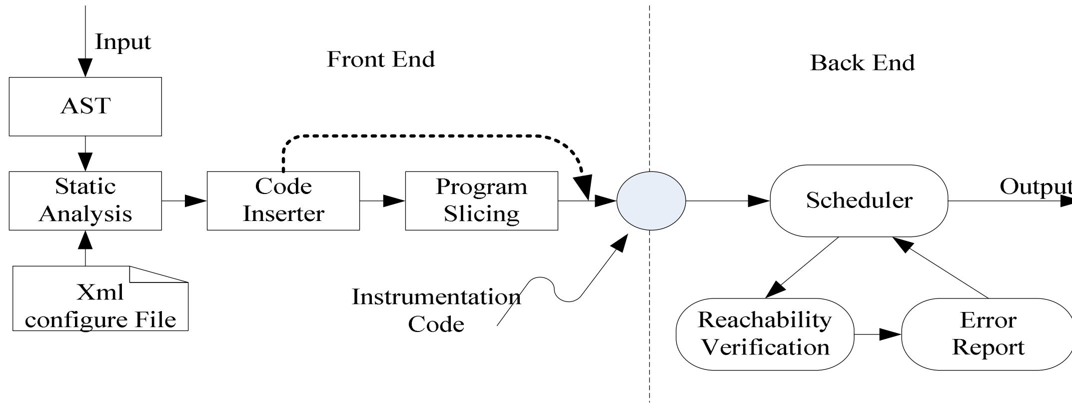

Figure. 1 Framework of CodeAuditor

(CEGAR) [18] is an automatic abstraction method where, starting with a relatively small skeletal representation of the system to be verified, increasingly precise abstract representations of the system are computed. The key step is to extract information from false negatives ("spurious counterexamples") due to over approximation. There are other techniques to reduce the system states, and we use program slicing to eliminate unnecessary statements in this paper.

Blast is an automatic verification tool for checking temporal safety properties of C programs. Combining with Lazy Abstraction, Blast is a representative tool of Model checking. In this paper we use blast as the back-end checker. The lazy abstraction concept [19], which is proposed and implemented in the BLAST, is aimed at optimizing the native abstract-check-refine loop by integrating the three steps. It means that the three steps (abstraction, checking, and refinement) are performed in an interleaving manner. The lazy abstraction is based on the following two principles: (1). On-the-fly abstraction: The native approach generates the entire abstract model at the "Abstract" stage. The lazy abstraction concept abstracts a region only when it is needed in the next step of checking. (2). On-demand refinement: In the native approach, the entire abstract model has to be rebuilt after refinement. The lazy abstraction concept suggests that we can reuse the partial answer that is obtained in previous iterations.

5. CodeAuditor Framework

The CodeAuditor framework in Fig. 1 consists of two components: the front end and the back end. The front end is the static analysis module, which takes a C program as input and an XML Configure-File that includes the rules of code instrumentation and the potential vulnerable functions of buffer overflow such as *strcpy*, and the programming slicing algorithm is implemented in this phase. The output of front end is a C program with instrumentation code and one or more assertions. The back end verifies the reachability of assertions and presents a final report of the security vulnerability detection.

5.1 Front End

The front-end generates constraint conditions, i.e. security properties needed for detecting buffer overflow, for related grammatical elements of C programs, and these conditions will be inserted as C instrumentation code into the program. The front end includes three sub-modules: static analysis module, code inserter module and programming slicing module.

Static Analysis. This module analyzes the grammatical structures of the original codes and finds out all the ones whose attribute is relevant to the code instrumentation. The attributes are divided into three types: length, operation and restraint. For example, the length attribute of a pointer variable p should be record, so an integer variable *p_length_max* and *p_length_used* is inserted into the original codes. The operation attribute of a assignment statement *p=array* needs to insert：

p_length_max = *array_length_max* and,
p_length_used = *array_length_used*.

For dereference assignment **p='a'* and function call *strcpy(dst, src)*, the assertions *p_length_max>0* and *dst_length_max* ⩾ *src_length_used* should be generated. At the same time we use the alias analysis of GCC, and we call the interface *get_alias_set()* for every AST node we concerned to obtain the alias variables, whose attributes are updated simultaneously.

Code Inserter. This module is called to insert related code when static analysis module finds a grammatical structure listed in the XML Configure-File.

Program Slicing. This module is used to eliminate statements which are irrelevant to the assertions. The user can activate this module by command-line, and then get pre-slicing results and pos-slicing results.

5.2 Back End

The back end is composed by three modules: scheduler, reachability verification module and error report module.

Scheduler. The execution at the backend is controlled and coordinated by the scheduler. In order to verify the reachability, this module needs to activate these assertions in the program in advance. In addition this module receives error information from the error report module and generates the final report.

Reachability Verification Module. In the C program to be audited, there exist more than one constraint paths. This module uses the model checking tool BLAST to verify its reachability for every constraint path. In our framework, other model checking tools, which take C programs as input, can be easily integrated.

Error Report Module. The output of BLAST is an error report which tells the trace path of the security vulnerability. However, the line number information in the trace path is based on the instrumentation code. This module implements the reorientation of the line numbers, and then returns the final result to the scheduler.

Table 2: source file and instrumentation code

Source file (perfect.c)	Instrumentation code
1 char *foo(const char * str1, const char *str2)	*1 int foo_str1_length_max = 0, foo_str1_length_used = 0;*
2 {	*2 int foo_str2_length_max = 0, foo_str2_length_used = 0;*
3 char *str;	*3 int foo_ret_length_max = 0, foo_ret_length_used = 0;*
4 char *p;	4 char * foo (const char * str1, const char * str2)
5 str = (char *) malloc(10);	5 { char * str;
6 p = str;	*7 int str_length_max = 0;*
7 strcpy(p, str1);	*8 int str_length_used = 0;*
8 p = str + 6 ;	9 char * p;
9 strcpy(p, str2);	*10 int p_length_max = 0;*
10 return str;	*11 int p_length_used = 0;*
11 }	13 str = (char *)malloc (10);
12 int main()	*14 str_length_max = 10;*
13 {	*15 str_length_used = 0;*
14 foo("hello ", "world");	16 p = str;
15 return 0;	*17 p_length_max = str_length_max;*
16 }	*18 p_length_used = str_length_used;*
	19 assert (p_length_max >= foo_str1_length_used);
	20 strcpy (p, str1);
	21 p_length_used = foo_str1_length_used;
	22 str_length_used = foo_str1_length_used; //alias
	23 p = str + 6;
	24 p_length_max = str_length_max - 6;
	25 p_length_used = str_length_used - 6;
	26 assert (p_length_max >= foo_str2_length_used);
	27 strcpy (p, str2);
	28 p_length_used = foo_str2_length_used;
	29 str_length_used = foo_str2_length_used + 6; //alias
	30 foo_ret_length_max = str_length_max;
	31 foo_ret_length_used = str_length_used;
	32 return str;
	33 }
	35 int main ()
	36 {
	37 foo_str1_length_max = 7; foo_str1_length_used = 7;
	38 foo_str2_length_max = 6; foo_str2_length_used = 6;
	39 foo (&"hello "[0], &"world"[0]);
	40 return 0; }

6. An Example

Our tool works in two phases: The first is the static analysis phase, where we traverse the AST, parse the XML Configure-File, execute special instrument and reconstruct the instrumented code; the second phase checks if the assertion is reachable. If the assertion is reachable, Blast can give an error trace path from the entrance to the vulnerability, but the line number of the trace is not the original source file, so we should use the result analysis module to redirect the line number of original source file.

The first column of Table 2 lists the C code to be checked. After being analyzed and processed by our tool, the instrumentation code is shown in the second column of Table 2. It is clear that function call, string operators, library functions and alias information have been involved. The instrumented code is given to the verification module, and the error path listed in Table 3.

7. Experiments

CodeAuditor was run on several C programs ranging in size from 400 and 6000 lines of code. First we want to measure the effectiveness of our tool, and see if it can detect the bugs that have been reported before and if it can find several unreported new ones; second we want to test the performance of our tool when program slicing is introduced.

7.1 Vulnerabilities Detection

Excepted for 159 simple test cases, we also apply CodeAuditor on several application software including minicom, corehttp and monkey. The experiment results are shown in Table 4. The second column is the size of software before and after instrumentation, the third column is the total number of alarms reported by CodeAuditor, the fourth column is the number of alarms that are true, the fifth column is the number of alarms that are false, and the last one is the number of new bugs.

Table 3: Error Trace for the bug of above code

Error Path reported by our verify-module:		
FileName	**lineNr**	**SourceCode**
perfect.c	14	foo("hello ", "world");
perfect.c	5	str = (char *) malloc(10);
perfect.c	6	p = str;
perfect.c	7	strcpy(p, str1);
perfect.c	8	p = str + 6 ;
perfect.c	9	strcpy(p, str2);

Table 4. Experiment Results of CodeAuditor

Software	LOC		Total Alarms	True Alarms	False Alarms	New Bugs
	Before	After				
minicom-1.80	6000	18080	3	2	1	1
corehttp-0.5.3 alpha	5008	13020	9	8	1	7
monkey0.1.1	443	1200	5	2	3	2

Table 5 Performance before and after program slicing was used

#	Predication number	Trace length	Time (ms)	result
Assert_1	4126	165	time out	No result.
Assert_1_slice	43	33	2530	safe
Assert_2	4140	305	time out	No result.
Assert_2_slice	33	36	2148	safe
Assert_3	507	47	3409	unsafe
Assert_3_slice	36	11	2743	unsafe
Assert_4	915	126	2315	safe
Assert_4_slice	15	6	1950	safe
Assert_5	715	76	12765	unsafe
Assert_5_slice	15	23	8550	unsafe

7.2 Performance Results

minicom has 6000 lines (including whitespace and comments). When the instrumentation phase is done, there are 29 assertions has been generated, four of which can be verified in time, and others are time out. Here we set the time limitation to be thirty minutes.

We introduce program slicing algorithm to our tool, and we select five assertions to test the performance of slicing algorithm. Also we add some code into blast in order to record the length of trace path and the verify number. Table 5 shows the experiment results of five assertions in minicom. The measurements are presented as the verify number, the trace length, the executive time, and the result, where the verify number means the number of calling Simplify in Blast, the trace length means the max-length of the path in work stack, the executive time shows the time the Blast spend to check the assertions, and the result shows the result of verification. We can see that the performance is improved enormously when program slicing is used. As the program slicing does not change the value of variables in slicing criterion, the safety of assertion will not be changed when the program is sliced.

8. Related Work

ATOM [20] and Pin [21] are Code instrument tools, which are primarily used for performance analysis and gathering statistics about programs but could be used to detect lower-level software bugs such as invalid memory accesses. Cascade [10] is another tool with auxiliary instrumentation to find bugs, but it can only process few hundred lines of code. Tikir and Hollingsworth [22] describe an instrumentation technique for obtaining coverage for testing. CRED is a detector tool based on compiler for finding bugs, but it is a dynamic buffer overflow detector and it's not accurate in some cases. CCured [7] uses a static verifier to prove as many dangerous operations safe as possible using a type system, and instrumentation is added to catch any bugs for operations that cannot be proved safe. One key difference between CodeAuditor and CCured is that CodeAuditor is designed to be a tool in terms of static analysis, instrumentation engine, and verified using model checking. There are some other groups who focus on techniques to incorporate static analysis in test case generation. In [23][24] static technologies are used to analyze source code and generate test cases.

Another popular method of finding bugs is dynamic bug detection, and such systems include GNU's checker, Cred [25]. In Cred, memory object is created to record the buffers. The bounds checker proposed by Jones and Kelly [26] is particularly attractive in that no pointer representation modifications are necessary.

Another area of related research is automatic program verification, which requires a balance between precision and efficiency. The more precise the method is, the fewer false positives it will produce. And the more expensive it is, the fewer program it will be applicable to. Historically, this trade-off was reflected in two major approaches to static verification: program analysis and model checking. Blast [27] presents a perfect tool which combines program analysis and model checking.

9. Conclusion and Future Work

Our goal is to audit precisely the buffer overflow vulnerabilities in the applications. We have presented our tool which is CodeAuditor based on program analysis and the model checking tool BLAST, and showed the experiment results of CodeAuditor. The result indicates that the tool has the low false alarm rate. In order to improve the performance of our tool, we implement program slicing to decrease the state number of system. In the future we are planning to apply our approach to other security vulnerabilities.

References

[1] J. Viega, J.T.Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code", In ACSAC, December.2000, pp. 257-267.

[2] D. Pozza, R. Sisto, "Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software", In COMSWARE, January.2006, pp.1-7.

[3] D. Evans, J. Guttag, J. Homing, and Y.M. Tan, "LCLint: A Tool for Using Specification to Check Code", In SIGSOFT FSE, December.1994, pp 87-96.

[4] D. Avots, M. Dalton, V.B. Livshits, and M.S. Lam, "Improving Software Security with a C Pointer Analysis", In ICSE, May.2005, pp.332-341.

[5] Newsome J., Song D., *Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software*, In NDSS, Feb. 2005.

[6] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering" , In STTT, 2007, pp.505-525.

[7] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W.Weimer, "CCured: type-safe retrofitting of legacy software", In TOPLAS, May.2005, pp.477-526.

[8] D. Wagner, J. Foster, E. Brewer, and A. Aiken. "A first step towards automated detection of buffer overrun vulnerabilities", In NDSS, 2000, pp. 3-17.

[9] P. Cousot, P. Cousot, “Formal language, grammar and set-constraint-based program analysis by abstract interpretation”, In FPCA, Oct.1995, pp. 170-181.

[10] N. Sethi, C. Barret, “Cascade: C assertion checker and deductive engine”, In CAV, 2006, pp.166-169.

[11] B.W Xu, J. Qian, X.F. Zhang, Z. Qiang Wu, and L. Chen. “A brief survey of program slicing”, In ACM SIGSOFT Software Engineering Notes. 30(2), Mar. 2005, pp.1-30.

[12] M. Weiser, “Programmers use slices when debugging”, In Commun. ACM, 1982, pp.446-452.

[13] M. Weiser, “Program slicing”, In IEEE Transaction on Software Engineering, 1984, SE-10(4), pp. 352-357.

[14] J.R. Lyle, D.R. Wallace, “Using the unravel program slicing tool to evaluate high integrity software”, In Proceedings of Software Quality Week, May.1997.

[15] Clarke E.M., Grumberg O., and Peled D., *Model Checking*. MIT ,1999.

[16]Queille, J.P., Sifakis, J.: *Specification and verification of concurrent systems in CESAR*. In: Proc. Symposium on Programming, LNCS 137. Springer ,1982 , pp. 337-351.

[17]. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., and Weimer, W., *Temporal-safety poofs for systems code*. In: Proc. CAV, LNCS 2404, Springer, 2002, pp. 526-538.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking”, In J.ACM, Sep.2003, pp.752-794.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and G.Sutre. “Lazy abstraction”, In POPL, 2002, pp. 58-70.

[20] A. Srivastava, A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools”, In PLDI, June.1994, pp.528-539.

[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace,V. Reddi, and K. Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”, In PLDI, June.2005, pp.190-200.

[22] M. Tikir and J. Hollingsworth, “Efficient Instrumentation for Code Coverage Testing”. In ACM SIGSOFT Software Engineering Notes, 2002, pp.89-96.

[23] K. Ku, T.E. Hart, and M. Chechik, “A Buffer Overflow Benchmark for Software Model. Checkers”, In ASE, 2007, pp.389-392.

[24] D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar, “Generating Tests from Counterexamples”. In ICSE, 2004, pp. 326-335.

[25] O. Ruwase, M. Lam, “A Practical Dynamic Buffer Overflow Detector”, In NDSS, 2004, pp.159-169.

[26] R. Jones, P. Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs”. In Proceedings of the International Workshop on Automatic Debugging, May.1997, pp.13–26.

[27]D. Beyer, T. A. Henzinger, and G. Théoduloz, “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”, In CAV, 2007, pp. 504-518.