國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

藉由迴圈相依性與限制式牴觸分析進行大範圍安全性質檢查

Scalable Security Property Checking by Loop Dependence and Constraint Contradiction Analysis

研 究 生：翁文健

指導教授：黃世昆　教授

中 華 民 國 九 十 八 年 六 月

藉由迴圈相依性與限制式牴觸分析

進行大範圍安全性質檢查

Scalable Security Property Checking by Loop Dependence and Constraint Contradiction Analysis

研　究　生：翁文健　　　　　Student：Wen-Chien Weng

指導教授：黃世昆　　　　　Advisor：Shih-Kun Huang

國　立　交　通　大　學

資　訊　科　學　與　工　程　研　究　所

碩　士　論　文

A Thesis

Submitted to Department of Computer and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

August 2008

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

藉由迴圈相依性與限制式牴觸分析

進行大範圍安全性質檢查

學生：翁文健　　　　　　　　　　　　　　指導教授：黃世昆　老師

國立交通大學資訊科學與工程學系（研究所）碩士班

摘要

安全性質檢查用來確認程式中是否存在違反安全性質且可被利用進行惡意攻擊之程式碼。對於邊界值測資以及動態程式分析很難進行此類檢查。故此 Execution Generate Executions 即 EXE 提出了 universal check 的概念，可以輕易地於動態程式分析中進行安全性質檢查，涵蓋範圍遍及所有可執行路徑。然而進行大範圍 universal check 是有困難的，因為執行路徑中 if-statement 的數量會呈指數成長。可是大部分的安全性質檢查是多餘的，原因如下：(1) 在迴圈之中或之後的安全性質檢查可能與迴圈執行結果無關。(2) 有其他的條件式保護造成多餘的安全性質檢查。本篇論文提出偵測此類多餘以及不必要的安全性質檢查。我們使用(1)迴圈相依性測試檢測迴圈與安全性質檢查的相關性。(2)條件式牴觸分析判斷安全性質檢查是否已被其他條件式限制保護。

Scalable Security Property Checking by Loop Dependence and Constraint Contradiction Analysis

Student：Wen-Chien Weng Advisor：Shih-Kun Huang

Department of Computer Science and Engineering

National Chiao Tung University

Abstract

Security property check is to verify if a program violates security rules and can be exploited to execute arbitrary code. This type of check is hard to be performed by testing with corner cases and dynamic program analysis. Thus Execution Generate Executions, or EXE for short, proposes the idea of universal check which is easy for dynamic program analysis and could cover all execution paths. Unfortunately, universal check is not scalable. The number of the if-statement of the execution path can be exponentially exploded, and most of the property checks are redundant, due to two reasons: (1) Property checks within or after a loop statement and the checks may not dependent on the loop. (2) The check is already protected by other constraints. This paper proposes methods to detect those redundant or unnecessary checks. We use (1) loop dependence analysis to check the dependency relationship between loop and property check, and the necessity of this check, and (2) constraint contradiction analysis to evaluate if the property check is already bounded by other constraints.

致謝

感謝指導教授黃世昆老師的栽培，謝謝老師讓我們可以很自由的做研究並且給予我們很好的建議。感謝博士班學長昌憲的指導，謝謝你不厭其煩的和我討論並且教了我很多東西。感謝立文、友祥、彥廷、泳毅、琨翰、士瑜、慧蘭不論是在技術、知識、生活上的幫助。最後更要感謝我的父母的栽培以及女友的支持。

要感謝的人太多了，那就謝天吧！

Content

List of Figures

List of Tables

1 Introduction

1.1 Problem Descriptions

Test generation is a mainstream of program analysis. It can be partitioned into two groups: static and dynamic test generation. Static test generation consists of analyzing the test program without executing the program and generating test input with the information acquired by static analysis. It is easier and faster, but imprecision. Dynamic test generation executes the test program and gathers symbolic constraints then computes input by solving these constraints with a constraint solver. It has better input coverage, more efficiency and precision than static test generation, since dynamic test generation can acquire additional runtime information. Runtime property checking is one of the applications of dynamic test generation. It checks the test program against a set of specific properties defined by users. It detects violations by generating an input to the test program that violates specific properties. However, the scalability is the main concern of dynamic test generation. There are many researches in order to address this problem. SMART[1], proposed by Godefroid, reuses the preconditions and post-conditions of higher level functions when those functions are used. EXE[2], proposed by Christian Cadar, uses cache mechanism to store constraint results. These methods are useful, however they can not work efficiently when there is a loop bounded by a symbolic variable in the test program. When a loop is bounded by a symbolic variable, the number of times of the symbolic execution is dependent on this symbolic variable. The symbolic execution has to try all possible value of this symbolic bound. In this paper we proposed a method called loop dependence and constraint contradiction analysis to address this problem and apply the method on security property checking. By this method we can scale the property check and enhance its performance.

1.2 Motivation

Most dynamic test generation addresses scalability by reducing the number of times of constraint query. EXE and CUTE[3] try to minimize the number of call to constraint solver, and minimize the formulas. SMART uses function summaries to lower the number of times of symbolic execution of a function. It is worth to note that most redundant symbolic execution happen due to symbolic bounded loop and unnecessary. We propose two methods. (1) To eliminate the redundant property checking (a kind of symbolic execution) caused by symbolic bounded loops. (2) Ignoring unnecessary property check that results from contradiction constraint.

1.3 Objective

In this paper, we observe that most security property checks (a kind of symbolic execution) are redundant. (1) Security property checks that are within or after a loop statement which is bounded by a symbolic variable. The times of performing the symbolic execution on the loop are dependent on the range of the symbolic variable which bounded the loop, but the security property checks are not necessary to be done at each time. If the security property checking checks a property that is nothing to do with the loop statement, then it only needs to be executed once. (2) Unnecessary security property checking. It is often to see that a property is already protected by some constraints. Then it is unnecessary to perform security property checking on it.

We propose two methods to address the above problem.

1. loop dependence analysis:

   It can ignore the redundant property check by analyzing the relationship between loop and property check.

2. contradiction constraint detection

   It can detect contradiction constraints and eliminate the unnecessary property check.

2 Background

2.1 Program testing

Program testing has been studied since the 70's(e.g., [4-7]). It assures that a program P would execute correctly with test input cases. By program testing, programmers can make sure their program functions well and find errors in programs. The main goal of program testing is to generate input cases that could exercise all program behaviors, that is, generating input cases that have the best test coverage. With better test coverage, programmers can test programs more thoroughly. And the problem is called test generation problem, it can be partitioned into two groups: static and dynamic test generation.

2.2 Symbolic execution

Symbolic execution[8] is a method which is widely used in constrain-based execution nowadays. The main idea of symbolic execution is that symbols are used to represent arbitrary values and the program is executed with these symbols. One symbolic execution can be mapped into a large set of concrete execution, so it offers a way to abstract concrete execution.

```
arithmetic ( int i ){
        int l, m, n;
        l = i + 2;
        m = l * 3;
        n = m – 1;
        if ( n % 2 == 0 )
                printf(“even”);
        else
                printf(“odd”);
}
```

Figure 1: Example code of symbolic execution

In Figure.1, the program reads an input, does arithmetic, and determines the answer is either even or odd. In the add operation at line 3, the symbolic l will be represented as “i + 2”, symbolic m will be represented as “3 * i + 6”, and symbolic n will be represented as “3 * i + 5.”

During the symbolic execution, we obtain an execution tree which represents the control flow of this program. The symbolic execution tree is represented by the path constraints along the execution path. In the above example, the execution tree has two branches, one is to print “even”, and the other is to print “odd”. The path constraint can be used to characterize the input. The input in the “even” branch have result which is even, and their path constraint is n % 2 == 0. The cases in the “odd” branch have result which is odd, and their path constraint is n % 2 != 0. Note that the path constraints can be accumulated. If there is a nested IF statement, the constraints will be conjoined together.

In the following discussion, CUTE, DART[9], EXE, RWset, and SMART all use symbolic execution.

```
void foo(int a, int b)
{
        if(b == hash(a))
                printf("equal\n");
        else
                printf("not equal\n");
}
```

Figure 2: Example of static and dynamic test generation

2.3 Concolic execution

The term concolic stands for cooperative concrete and symbolic execution. This term is first proposed by K. Sen, D. Marinov and G. Agha, but the idea is first proposed by E. Larson and T. Austin[10]. It is widely used in modern dynamic test generation. Dynamic test generation tools like CUTE, DART, EXE and ALERT(our dynamic test generation system, this paper is based on this framework, it will be introduced in the following.) all uses this technique. Concolic execution performs concrete execution and symbolic execution at the same time and replaces with the concrete value expression that symbolic execution does not know how to generate constraint or the constraint solver can not handle. Thus concolic execution mitigates the imprecision in symbolic execution. It also maintains the advantages of symbolic execution: (1) avoids generating redundant test cases, (2) better test coverage.

2.4 Static test generation

Static test generation (e.g., [11-13]) consists of analyzing the test program without actual executing the test program and generating the test case to the test program. It is fast and easy but less precision. If there is a function F in the test program and F can not be analyzed, then the static test generation will lost its precision and code coverage. For example, in Figure 2, function hash is a library function and can not be analyzed, therefore the static test generation is impossible to generate a test case satisfying the constraint b == hash(a). This situation is often occurred in application program, thus static test generation is hard to become a practical tool.

2.5 Dynamic test generation

Dynamic test generation (e.g., [14-16]) consists of executing the test program P with a set of random input, and gathering the symbolic constraints along the execution path, then using constraint solver to reason another test case that will trigger another execution path. Dynamic test generation will repeat the above steps until error occurs or traverses all branches. Dynamic test generation is slower than static test generation, but has better code coverage. For example, in Figure 2, with executing the function foo, dynamic test generation can give a random concrete value to input a. If it triggers the else branch, then next time dynamic test generation can fix the input a and make b == hash(a). Thus both branches can be reached. The main advantage of dynamic test generation is having additional runtime information. Due to the reason dynamic test generation is more powerful and practical.

2.6 Security property checking

In software testing, property means a specific behavior of a program execution, such as the absence of buffer overflow or memory leaks. Runtime property checking dynamically checks whether a program satisfies a set of specific property. Tools like Valgrind[17], Purify[18] and SAGE [19]are widely used. With these tools software developers can products more efficiently. Security property checking is a kind of runtime property checking to check security properties such as integer overflow, and buffer overflow. It also is a kind of dynamic test generation. Security property checking will try to generate a test case that violates the specific properties by using a constraint solver. If it could generate a test case that violates the specific security properties, then it infers there is a security flaw in the test program.

2.7 ALERT

ALERT is abbreviated from “Automatic Logic Evaluation for Random Testing”. It is a dynamic test generation using concolic execution technique and is a mix framework of CUTE and EXE. It uses CIL (C Intermediate Language, an infrastructure for C program analysis and transformation.) to simplify and instrument the test source code, and uses CVC3 (an automatic theorem prover for Satisfiability Modulo Theories problem) as constraint solver. It can generate test cases to cover the program execution paths. It also supports security property checking; moreover, it can automatically generate corresponding post constraints of C standard library functions without instruments C standard library functions.

In Figure 3 is ALERT system architecture and following are ALERT execution steps: (1) ALERT first preprocess the tested source code then use CIL to simplify the source code.

CIL supply 3 simplifications: simple three-address code, simple memory operation, and one return. Besides, CIL transforms while, for and switch statement to if statement. (2) After that ALERT again uses CIL to instrument the tested source code, add some additional functions to gather the runtime information for dynamic analysis. (3) And then ALERT compiles the instrumented source code to executable program and executes the program. (4) Then ALERT executes concolic execution, the constraints along the execution path will be collected. The last constraint along the path will be negated and all the constraint will be fed into the constraint solver to generate a set of input to the next path. ALERT executes the step 4 until all the execution paths of this test program are traversed or encounter an error. During concolic execution ALERT also performs security property checks. It checks whether the security property is violated. If it is true then ALERT will print warning message. Besides, loop dependence analysis and constraint contradiction analysis are also based on ALERT framework. While ALERT performs concolic execution, it also collects variables that are assigned during loop executions and variables in security property check. The symbolic constraints along the execution path are also used in constraint contradiction analysis.

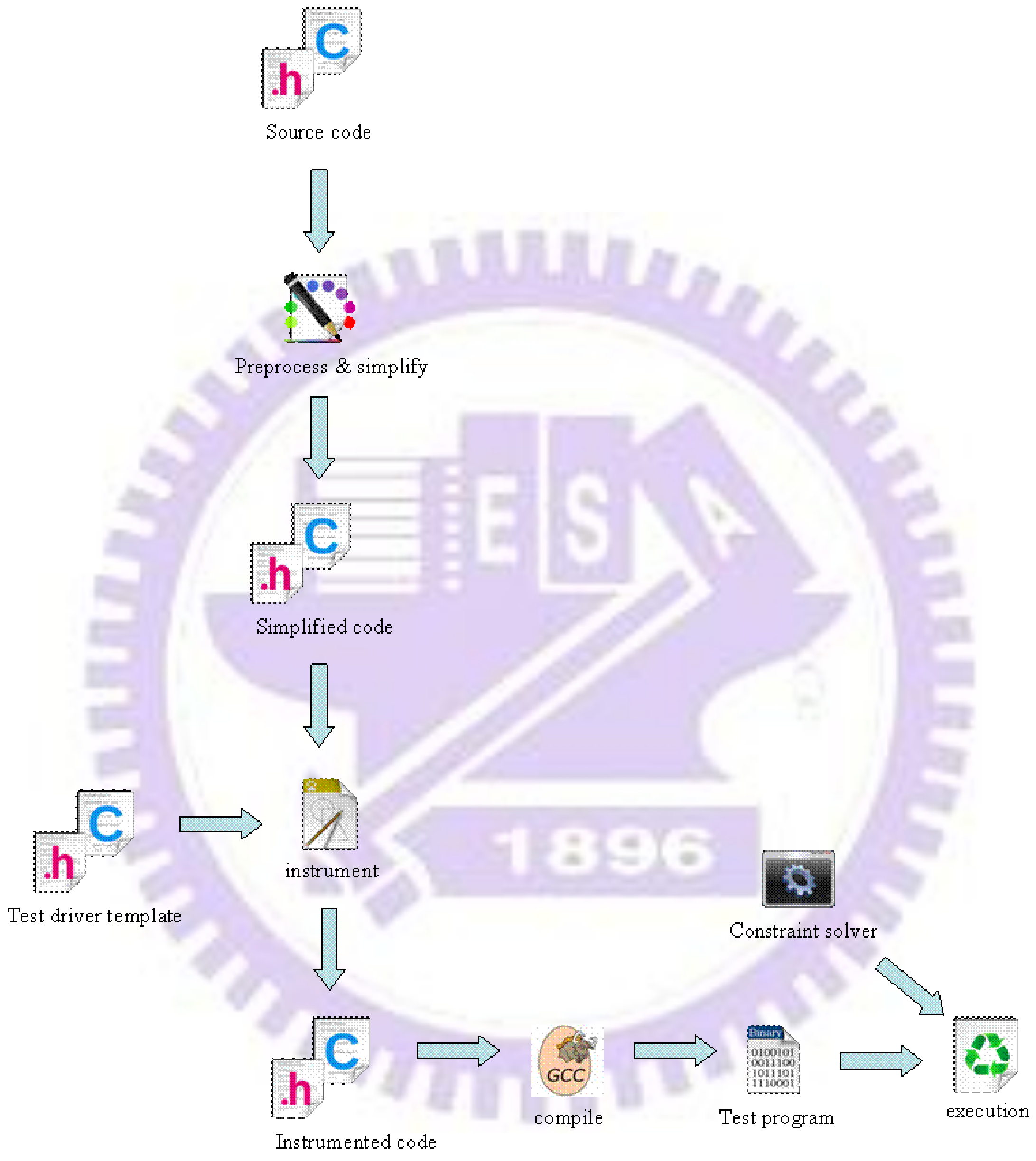


Figure 3: ALERT system architecture

3 Related work: Constraint-based Execution Optimization

Constraint-based execution systematically explores all execution paths in the program by collecting constraints along the path, then using a constraint solver to find next executable path. However, dynamic testing suffered from the problem of scalability. To enhance the performance we can either optimize constraint solving or eliminate redundant paths. We classified these optimizations according to their purpose in the following.

3.1 Constraint Solving Optimization

3.1.1 Minimizing Number of Calls to Constraint Solver

Constraint solving is a very CPU-expensive operation; therefore we want to minimize the number of calls to the constraint solver. We first present optimizations in EXE and CUTE, discuss their advantages and drawbacks.

Constraint Caching

In EXE, a persistent cache is used to minimize the number of calls to the solver. EXE prints the constraints as a string and computes the MD4 hash of the string. It then queries the cache to see whether it is hit or not. If it gets a hit then the cache will return the answer; if not, after invoking the solver EXE the (hash, result) pair is stored back to the cache.

If the cache size is not big enough, EXE has to replace a (hash, result) pair. Least recent used strategy may be a good one. It is possible that an error answer occurs due to hash collision. If so, it may get a false path, and cause the false positive or false negative. Thus, universal hash might be a good choice.

Fast Unsatisfiability Check

In CUTE, if the last constraint is syntactically the negation of any preceding one, then the solver do not need to solve it, since the last one contradicts one of the preceding constraints, and the whole path constraints is invalid.(In CUTE’s experimental results this method can reduce the number of semantic checks by 60-95%.)Intuitively, linear search can be used to check whether the last constraint is the negation of the preceding constraints. However, it is not practical in real world, since the number of constraints is usually huge. Hash function might be a good solution, it can reduce the search time from O(n) to constant.(n is the number of constraints.)

3.1.2 Minimizing Formulas

The longer formula we feed to the solver the more time it needs to solve constraints. Thus if we can minimize formula, we can speed up constraint solving. Following is three optimizations in EXE and CUTE respectively.

Common Sub-constraints Elimination

Before passing path constrains to the lp_solve(CUTE builds their own solver on top of lp_solve.), CUTE first identifies and eliminates common arithmetic sub-constraints. (In CUTE’s experimental results this method can reduce the number of sub-constraints checks by 64-90%.) It is possible to find common sub-constraints by linear search, but the number of constraints is usually huge. Thus linear search may not be a good choice.

An MD4 hash for each sub-constraint is computed and stored in a cache. Before adding a constraint into the path constraints, computing the hash of this constraint and then check whether it is in the cache or not. If so, the constraint will not add into the path constraint formula. Therefore the time complexity of searching reduces from O(n) to constant.(n is the number of constraints in the path constraint formula.)

Constraint Independence Optimization

This is one of EXE’s important optimization, which exploits that path constraints can be divided into multiple independent subsets. EXE defines independent if two constraints have disjoint sets of operands. This optimization method has two benefits. (1) It can ignore irrelevant constraints, thus decrease the cost of constraint solving. (2) It can increase the cache hit rate in EXE, since the subset may be used by previous paths and smaller subset reduce the number of checks. Thus it increases the cache hit rate.

EXE computes independent subset by constructing a graph G, whose nodes are variables existing in all the constraints. Adding an edge between nodes n_i and n_j if and only if there is a constraint that contains both of the nodes. Once graph G is constructed, EXE apply component connected algorithm on G to compute its connected component. Then for each connected component construct corresponding independent subset by adding all constraints in the same component.

Incremental Solving:

Incremental solving identifies dependency between sub-constraints and exploits it to achieve following goal: (1) solving constraints faster. (2) keeping the input similar.

CUTE defines two constraints p and p’ are dependent if variables in p and p’ have conjunction or there is a constraint p” such that p and p” are dependent and p’ and p” are dependent. It is worth to note that in CUTE, path constraints in two consecutive concolic executions C and C’ only differ in the last constraint. Assume their corresponding solutions are I and I”, and the last constraint in C is K. CUTE first collects constraints in C that are dependent with the negation of constraint K. Let the set of those constraints be M, and the solution of M is C”. Solution I” is the same as I except for those constraints in M every solution of which is replaced by C”(l).(l is the constraint in set M.), that is, the solution of constraints that are not dependent with K still remains while the solution of constraints that are dependent with K is changed.

The above optimizations have the same object: reducing the path constraints size by reusing previous answers. However, they achieve the goal by different ways. EXE divides constraints into multiple subsets, then checks the cache whether there is an answer, and just solves constraints that are not in the cache. CUTE first finds out the constraints in C that is dependent with the negation of the last constraint N, and then solves these constraints to obtain solution C”. The next solution C’ is the same as the previous one C but for those constraints dependent with N replace their solution with C”(l).(l is the constraint that is dependent with N.).

Because EXE will fork process, it is better to use cache, and every child process can query the cache. CUTE will negate the last predicate then compute the next input, thus the replace strategy is better for it, since every execution path is relevant to the previous one. In fact constraint independence optimization and incremental solving both use the “cache concept”. They do not solve redundant constraints and only solve new constraints. Both of them have significant improvement on execution optimization.

3.2 Path Optimization

3.2.1 Constraint Caching and reusing

Concolic execution will systematically execute every feasible path. In fact, the program paths often increase exponentially in real program. Therefore executing all feasible paths is not scalable. However, many paths are traversed repeatedly in test programs. One of solutions to alleviate the problem is constraint caching and reusing, that is to reuse constraints that already exist. SMART, which is an extension of DART, exercises functions in the program and records the summaries (pre-conditions and post-conditions) of these functions by their inputs and outputs. SMART re-use these summaries when program calls these visited functions. According to SMART’s experiments, it can reduce the number of execution path from exponential to linear. ALERT with SP module (Resolving Constraints for COTS/Binary Components for Concolic Random Testing proposed by Yang-Chieh Fan [20]) also uses the similar idea of constraint caching and reusing. However, it only implements on the C Standard Library currently. When the test program calls a function from the C Standard Library, instead of traversing this function, ALERT will automatically add the corresponding post-condition that is pre-constructed by ALERT developer. Not only alleviate the path explosion problem, it also makes the concolic execution more precise and explores more program paths.

SMART algorithm, compared with DART, is only different in function call statement. When SMART executes a function call f, it first checks whether a summary of f is available by checking whether current symbolic calling context implies the disjunction precondition currently record in the summary of f. If so, SMART adds the summary of f into the path constraints and turns the backtracking flag off, which is resumed when function f returned. If no summary of f for current input is available, SMART will put the current input of f into a stack and start backtracking f for current input. When it finishes backtracking, it will execute add_to_summary function to compute the summary of f. After computing the summary of f, SMART will determine where it should backtrack next by executing solve_path_constr function. When backtracking is over, SMART will restore the input saved in the stack.

Unlike SMART, ALERT with SP module provides an easy way to add post-conditions of different functions and also support a post-condition library of the C Standard Library. When the test program calls a function in the C Standard Library, ALERT instruments a stub function after this function to add the corresponding pre-conditions into the current constraint system. By this way it is not necessary to add source codes of functions in the C Standard Library into the test program, and can add post-condition without traversing the C Standard Library function.

The researchers of SMART run both SMART and DART on the subset of oSIP parser code, which is an open-source C library implementing SIP protocol. According to the paper of SMART, the runs DART needed is exponentially increasing with the packet size, however, SMART just increase linearly with the packet size. In addition, the running time is linear to the number of the runs. The paper of Resolving Constraints for COTS/Binary Components for Concolic Random Testing shows that the frequency of invocations of external functions increases with the scale of the test program. With larger test programs the missing constraints of external function increases. However, ALERT with SP module can efficiently mitigate this problem by adding additional post-constraints. It also implies ALERT with SP module can alleviate the path explosion problem, because it does not have to traverse those external functions to obtain constraints.

DART has path explosion problem, but SMART solves it without loss path coverage and makes it more scalable to large programs, because SMART can use the information about functions it computed and re-use those summaries when it executes these functions again. Thus, paths in these summarized functions do not need to be traversed repeatedly. Moreover, this idea is not only just on functions, but also on loops, program blocks or object methods.

The purpose of ALERT with SP module is to mitigate the gap between external functions and test programs due to missing constraints resulted from concretization. With its help, ALERT with SP module makes concolic execution more precise. In addition we also can view this method as a kind of constraint caching and reusing since it can generate required constraints without traversing external functions. Compare to SMART, ALERT with SP module is a simple kind of SMART which can not generate required constraints during execution and need to construct by users beforehand.

3.2.2 Path Reduction

To solve path explosion problem, SMART uses summary of functions to avoid repeated traversal. On the other way, this problem can be alleviated by avoiding repeated traversal paths that causes the same side-effects since a lot of paths are redundant.

RWset[21], which derives from EXE, discards current path that have same side-effects as previously exploded one. RWset proposes that it is unnecessary to traverse path that has same side-effects with previously one. Once it recognizes this situation, it will stop the traverse and generate an input for this path. And the experiments show that RWset got a significant improvement on efficiency.

The main idea of RWset is many execution paths have the same effects, and it depends on program point and program state. A program point is a MD4 hash of the program counter and callstack, and a program state consists of the current path constraint and the values of all concrete memory location. If there is an execution path that has the same program state with previously executed one at some program point, then RWset considers this path will produce the same effects and truncates this traverse. Besides, this idea can be enhanced by exploiting that two program states are identical if they only differ in variables that are not read after subsequent execution.

RWset records all program states along the execution path so far by writeset which is a set of values of all concrete memory locations and path constraints at every program point; moreover, it also stores readset at each program point. Given a program state, the readset collects all locations read after a program point. When program reaches a program point, RWset determines whether it has seen the program state before. RWset first intersects the writeset and path constraints with the corresponding readset, and then compare the intersection result with the current program state. If the result and the current program state are equivalent, then RWset will prune this execution and generate a test case for this path; otherwise RWset continues.

Researchers of RWset ran five medium-sized open-source benchmark compared with EXE. The results show that RWset has a significant improvement. It substantially reduces the number of test cases to achieve the same coverage as in EXE. In addition RWset does not cause much overhead, according to the evaluation result of RWset, for all the benchmarks, the average runtime overhead of RWset is at most 4.38%. They measure it by running an EXE version with all RWset require computations but without pruning any paths. The most important problem that RWset solved is the scalability problem. As we already know, constraint-based execution is not scalable when it checks large scale programs. RWset proposes a new method to detect and prune the redundant paths.

4 Method

4.1 Loop dependence analysis

The origin idea is proposed by RWSet. In this paper, we use this idea to analyze the relationship between loop and security check. We use a loop set to represent the set of variables that increase their value while executing loop iteration, and use a security property set to represents the set of variables that security property checks. This analysis dynamically executes along with symbolic execution. During the first loop iteration, variables which have assignments in the loop statement will be collected into the loop set. If there is any security property check, then the variables being checked is collected into the security property set. After finishing the iteration, we intersect the two sets. If the intersection is null then the property check is independent of the loop. The security check is unnecessary in the successive iterations. Thus the successive iterations could be ignored. If the intersection is nonempty, then the value that security property check checks is dependent on the loop. The security checks need to be done in the successive iterations. By this analysis the redundant security property check will be eliminated and the loop execution also be reduced.

In Figure 4, the elements in loop set are i and k, the element in the security property set is s, and the intersection of these two set is empty. We can infer that the property check is independent of this loop. Thus this loop will not be executed in the successive symbolic executions since it can be ignored. In Figure 5, the intersection of these two set is non-empty, and the element in the intersection set is s. Thus, this loop will be executed normally in the successive symbolic executions since the iterations of this loop may affect the security property checks.

```
testme(int bound, int s, int k){
        unsigned int i, a;
        for(i = 0; i < bound; i++)
                k++;
        UC(s);          //the security property check will perform on this line to
                        //check if s can be overflowed
        malloc(s);
}
```

Figure 4: Intersection result is null

```
testme(int bound, int s){
        unsigned int i, a;
        a = s;
        for(i = 0; i < bound; i++)
                a++;
        UC(a);          //the security property check will perform on this line to
                        //check if s can be overflowed
        malloc(a);
}
```

Figure 5: Intersection result is non-null

Loop dependence analysis can be viewed as dynamic reaching definitions analysis. If some variable in loops can reach security property checks, then the number of iteration of those loops will affect security property checks. Otherwise, those loops can not affect security property checks. Thus, the executions of those loops can be ignored. Loop dependence analysis takes the advantages of dynamic analysis. It lowers the false positive and false negatives while it does not cause much overhead.

4.2 Constraint contradiction detection

Programmers write codes more carefully than before since more and more attacks are discovered, such as buffer overflow, integer overflow, format string attack. Thus programmers usually add some secure checking code to prevent from being attack. Therefore, the security property checks will be unnecessary. In this paper we proposed a method called constraint contradiction detection to find those unnecessary secure property checks and eliminate them.

Instead of direct executing security property check, we check whether the security property is protected (bounded) by other constraint by using reverse incremental check (details in next section). If this property is already protected (bounded) then it means that the property check is unnecessary and can be ignored. Otherwise it means this security should be performed

In Figure 6, security property check should be performed before calling malloc() function call, however the constraint contradiction detection will detect the malloc() function call is bounded by the constraint 0 < i && i < 10, thus the property check becomes unnecessary and can be eliminated. For efficiency we also use a cache mechanism to store the constraint contradiction information. Before executing reverse incremental check, we first check the cache whether this property check already be executed and be protected. If it is true then the property check could be ignored, otherwise the reverse incremental check will be executed and the constraint contradiction information will be store in the cache.

```
testme(int i){
        if(0 < i && i < 10 )
                UC(i);          // the security property check will perform on this line
                                // to check if s can be overflowed
                malloc(i);
}
```

Figure 6: Example of constraint contradiction detection

4.3 Reverse incremental check algorithm

The kill subroutine takes current_path and cerrent_UC as arguments, in Figure 7 line 2, we first check each cache data whether this constraint is already detected. If true then the variables of the constraint will not be collected into the security property set, otherwise the incremental reverse check will be executed in line 6.

The reverse incremental check starts from the latest predecessor constraint of the current_UC, then using constraint solver to solve the predecessor constraint and current_UC, if valid write the current_path and current_UC into cache, else collect the predecessor constraint of current predecessor constraint into solver and solve again. Repeat line 9 to line 15 until there is no more constraints.

The main idea of reverse incremental check is based on a heuristic that programmer often fix bugs by put some patch code just right before where bugs happened. For an example, in Figure 5 the program might cause buffer overflow without the if statement checking the value of variable i in line 2, thus programmers will put the if statement right before malloc() function call in line 3. Therefore security property checks are often bounded by some constraints that assure the verification of the program. With reverse constraint contradiction detection, we can fast detect and ignore unnecessary security property check, moreover we cache this information to enhance the efficiency.

```
kill ( current_path, current_UC ){
    foreach contradicted constraint (old_path, old_UC)
        if (old_path == prefix(current_path) && old_UC == current_UC)
            return;  //UC do not add in Securityset
    //reverse incremental check
    i = the last constraint;
    add current_UC to solver;
    while (i > 0)
        add constraint into solver;
        solve ;
        if (valid)
            write current_path and current_UC in cache;
            return;
        else
            i--;
}
```

Figure 7: Reverse incremental check algorithm

5 Implementation

5.1 Data structure

we define three kinds of structure and use four vector arrays in the loop dependence analysis.

5.1.1 Structure

element: structure element consists of name and type. It is used to record the information of elements in the wset. The naming mechanism satisfies static single assign form (SSA).The details of the naming format will be described later.

loopbranch: structure loopbranch is used to record the information of branches which are condition expression in loop statements. It has five fields. The branchID is the identifier of branches. The Boolean variable begin distinguish the first iteration from loop branches. The loopID records its corresponding loop statement. The secondIterBranchID records the branchID of the second iteration.

loopInfo: structure loopInfo records information about the corresponding loop statement. Vector west in the structure loopInfo is used to collect all left hand side variables in the loop statement. Field intersect stores the intersect result with rset. We eliminate loop constraints depend on the intersect result. If intersect equals 0, it implies the wset of this loop statement does not intersect with rset. And this loop statement can be eliminated. Field seconIter is the same as the field secondIterBranchID in the structure loopbranch. Field lineNum stores the line number of the loop statement and field current stores the order of loop statements. We can identify loop statements by using lineNum and current.

5.1.2 Vector

rset: it is a simple vector which stores all the variables referenced by universal check and the naming format is the same as wset.

wsetStack: it records the state of nested loop, the loop id (the field lineNum in the structure loopInfo) will be popped into this stack and be popped out while leaving this statement.

loopbranchStack: this vector stores structure loopbranch which are condition expression in the loop statements in the execution order. It helps us distinguish loop branches form branches.

loop: this vector stores structure loopInfo in the execution order.

5.2 Naming

Loop dependence analysis collects variable name into loop set and security property set. We use the format "varName_cn_varRepos" to represent variables. "varName" is variable name. Use "cn" as procedure stack counter, increasing "cn" while enter a new procedure stack, decreasing it while leaving. "varPepos" increases when variable varName is assigned, single static assignment (SSA) can be achieved by this manner. For example in Figure 8, variable a in the function foo and function foo is called two times in main function. The name of variable a is a_1_1 for first call and a_1_2 for second call.

```
void foo ()
{
        int a;
        printf("%d\n");
}

int main(int argc, char *argv[])
{
        foo();
        foo();
        return 0 ;
}
```

Figure 8: Naming

5.3 Taint analysis

Loop dependence analysis also can be viewed as taint analysis. It traces variables assigned in loop statements and monitors whether variables in security property checks are tainted. It is easy doing tainted analysis in a single function however it is complicated while doing tainted analysis among multiple functions. We classify 4 taint approaches that variables in one function could affect variables out of the function and discuss in following.

5.3.1 Tainted by global variable

If a global variable is tainted, then it can affect any functions using it. In Figure 9, the security property check is affected by global variable a. Using our naming mechanism can handle this situation. Variable a will be formatted as a_k, and k depends on the bound of for loop statement, note that global variables do not have cn.

```
int a = 3;
testme(int i, int j){
        int b;
        for(b = 0; b < i; b++)
            a++;
        UC(a);          //security property check on variable a
        malloc(a);
}
```

Figure 9: Tainted by global value

5.3.2 Tainted by function arguments

Tainted variable can affect other functions through function arguments, thus while function arguments are passed, loop dependence analysis will check whether arguments are tainted. In symbolic execution arguments will be put in a stack before calling the callee function. While entering the callee the parameters will be pop out and be symbolized. In this step dependence analysis will check arguments are tainted or not before symbolize the arguments. If true the symbolic variable of the tainted parameter will be tainted either. In

Figure10, variable j in function testme will affect malloc() function in function foo. The dependence analysis will detect it and make symbolic variable of variable a in function a tainted.

```
foo(int a){
        UC(a);    //security property check on malloc(a)
        malloc(a);
}

testme(int i, int j){
        int a;
        for(a = 0; a < i; a++)
                j++;
        foo(j);
}
```

Figure 10: Tainted by function arguments

5.3.3 Tainted by return value

Tainted return value results from a tainted function argument. Therefore return values should be checked while being used by other functions. Before finishing callee function the return value will be put in a stack. Then it will be pop out and assigned to other variable. The loop dependence analysis checks whether the return value is tainted and set the variable assigned by this return value tainted if it is tainted. In Figure 11 the return value n of function foo taint the variable k of function testme. This will be detected by loop dependence analysis while pop the return value out and assign it to variable k. Loop dependence analysis checks the loop set, if return value is in the set then set variable k tainted by collecting into loop set.

```
int foo(int m, int n){
        int a;
        for(a = 0; a < m; a++)
                n++;
        return n;
}

testme(int i, int j){
        int k;
        k = foo(i, j);
        UC(k);    //security property check on malloc(k)
        malloc(k);
}
```

Figure 11: Tainted by return value

5.3.4 Tainted by pass by address

It is different to handle the pointer case when the function argument or return value is a pointer to some type. The main idea is to obtain what the pointer points to and then check whether it is tainted. Therefore it is necessary to maintain the information of all variables including memory address, variable name and type. In our security property checking system, we use a table called obj_map to maintain variable information. In Figure 12, before collecting pointer j in function foo into loop set, we have to look up the obj_map by variable address then check if the “real” variable is tainted. In this example obj_map will return a is pointed by j, and then check whether variable a is tainted.

```
void foo(int i, int *j){
        int k;
        if(i < 3)
                for(k = 0; k < i; k++)
                        (*j)++ ;
}
```

```
void testme(int i, int j){
        int a;
        if(i > 0){
                a = i;
                foo(j, &a);
//security property check on malloc(a)
                UC(a);
                malloc(a);
        }
}
```

Figure 12: Tainted by pass by address

5.4 Nested loop

When the test program enters a loop statement we will use line number as an identifier and put this identifier into the vetor wsetStack. When the test program leaves a loop statement we will pop its corresponding id out. Using this method we can trace the state of loops in the test program, if the size of this stack is greater than one it will implies the test program enters a nested loop statement. With the help of wsetStack we do not push redundant element into the vector wset and avoid confused.

5.5 Loop constraints elimination

Loop constraints elimination performs when the following conditions hold:

1. the last constraint along the current path is a loop constraint, and

2. the wset of the corresponding loop statement does not intersect with the rset.

Loop constraints elimination will pop the constraints out until the branch of the second iteration of this loop statement.

```
// sym and j are symbolic variables
test(){
        while(i < sym){
                if(j == 1){
                        //statements
                 }else{
                        //statements
                 }
        }
}
```

Figure 13: Loop constraints elimination example function test()

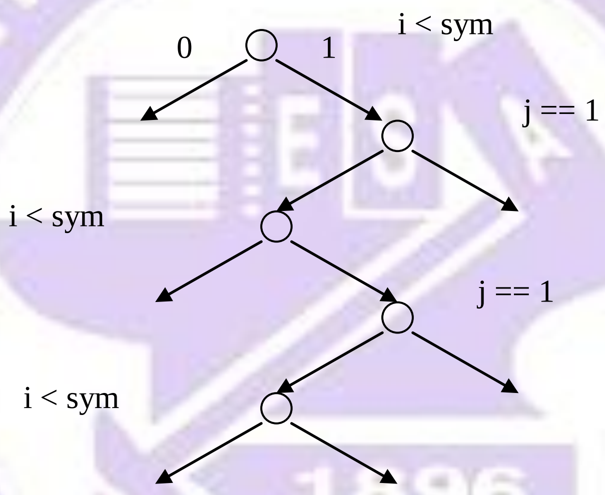


Figure 14: Execution tree of function test()

Symbolic execution collects constraints along the execution path into a set S and systematically traverses all the execution paths in the test program by negate the last constraints in the set S, that is, symbolic execution traverses by depth first search. A loop constraint is a constraint that decides whether the loop statement should be executed. For example in Figure 13, the while loop statement is dependent on the constraint i < sym, and the constraint i < sym is a loop constraint. That is, it is a condition expression in loop statements. Figure 14 is an execution tree of the function test(). The nodes represent the condition expressions of while loop statement and if statement. The left edge of nodes means the condition does not satisfy and represents by 0. The right edge of nodes means the condition satisfies and represents by 1. For example, if the while loop statement executes two iterations and the if statement j == 1 does not satisfy, then the execution path will be 1,0,1,0,0. and we say that there are three loop branch sets. The first set includes two nodes, one for the condition expression of the while loop statement, and one for the if statement. The second set is as same as the first set. And the third set only includes one node which is the condition expression of the while loop statement. When the last constraint in the set S is a loop constraint and the intersection result of this execution is null, then we will pop the constraints out until the first loop branch set of this loop statement. For example in Figure14 the three nodes of the bottom will be popped out. And our system will try to negate the branch of the if statement in order to traverse all branches within or after loop statements. The loop branches control the number of times of the loop iterations. If the intersection result of the corresponding loop statement in current execution is null, then we assume the loop iterations are independent with universal checks and we pops all branches until the first branch set of the loop iteration. By this method we reduce the number of times of iterations and universal checks, and we still can traverse all branches of if statements within or after the loop statement. Thus it can prevent from false negative due to lost collection of wset. If the west of the current loop statement intersects with rset then it implies that the number of times of this loop statement will affect the universal checks and the elimination will not perform.

Notice that the loop statements described in this paper are symbolic bounded, that is, the variable sym in the while statement in Figure13.is a symbolic value, and the number of times that the while statement should be executed in the traditional symbolic execution is bounded by the range of variable sym.

```
 1 // j and sym are symbolic variable
 2 while(i < sym){
 3         if(j == 3){
 4                 //security property check on malloc(a)
 5                 UC(a);
 6                 malloc(a);
 7         }
 8         i++;
 9         j++;
10 }
```

Figure 15: Special case of loop constraints elimination

Figure 15 is a special case, the if statement in line 3 is not satisfy in every iteration and it only satisfies in specific iteration. If we only pops out one loop branch set, there will be two situations: (1) the true branch in Figure 15 will not be traversed until the specific iteration. (2) the worst case, the false negative may occurs due to lost collection of west.

One of the advantages of our loop constraint elimination strategy is the special case can be found and traversed quickly. By our strategy, if the loop can be reduced then the constraints will be reduced until the first loop branch set. And ALERT will try to negate the constraint of the if statement in line 3. Both branches of the if statement in line 3 will be traversed.

```
1 loop_constraint_elimination(current_branch)
2 {
3         loopInfo loop;
4         loop = loopbranchfind (loopbranchStack, current_branch);
5         while(loop != NULL){
6                 if(loop.wset != empty && loop.intersect == 0 && loop.begin == false){
7                         pop until the first loop branch set;
8                         is_loop_branch = loopbranchfind (loopbranchStack, current_branch);
9                 }else
10                break;
11        }
12 }
```

Figure 16: Loop constraint elimination algorithm

Figure 16 is the loop constraint elimination algorithm. In line 4 we first check if the current_branch is a loop branch. If false the loop constraint elimination will finish. If true we then check whether the elimination conditions are all satisfied in line 6. (1) the loop statement should be traversed. (2) the wset of this loop statement does not intersect with rset in the current execution. (3) it is not the first loop branch set because it is unnecessary to eliminate the first loop branch set. If the above conditions are all satisfied then the loop branch sets will be popped until one loop branch set left. And the steps will perform on all loop statements.

6 Evaluations

6.1 Sort algorithms compared with CREST

CREST is an open-source dynamic test generation tool for C. It is maintained by Jacob Burnim from University of California, Berkeley. CREST uses CIL to insert instrumentation code into a test program, performs concolic execution and then uses Yices, a constraint solver, to generate new input to another unexplored execution path.

In our paper we use insertion sort as a test case. We use four sort algorithms insertion sort, selection sort, bubble sort and shell sort as test cases of CREST and ALERT with loop dependence analysis and then compare the result. Due to different driver template, the test codes for CREST and ALERT with loop dependence analysis are a little different but same work. In Figure 17, the codes from line 23 to line 25 are set char array ary and integer variable len symbolic, and line 26 limits the variable len under some value due to the variable len can not greater than the length of array ary. For ALERT with loop dependence analysis, char array ary and integer variable len are set symbolic in the instrumented code, line 8 and line 9 in Figure 19 are set them symbolic. Line 22 in Figure 19 is the same purpose as in line 26 in Figure 17.

We run CREST by the command "../bin/run_crest ./testInsertionSort 9000 –dfs". The reason of using depth-first-search is ALERT uses depth-first-search too. The result is in Table 1. The number of iterations increases with the length of the array. As we know there are many redundant iterations that traverse the same execution paths because the for loop statement in line 7 Figure17 and the while statement in line 10 Figure17 are bounded by symbolic values. However, we run the test case in ALERT with loop dependence analysis, because the universal checks on malloc() and memcpy() in the source code in Figure 18 do not intersect with rset, then it implies that the traverses of the for statement and the while statement in Figure 18 can be reduced. In Table1 we can see ALERT with loop dependence analysis really reduces much redundant execution paths. ALERT with loop dependence analysis has the same result with different array length because the universal checks is irrelevant to the array length which bounds the for loop statement. Figure 20 is the source code of selection sort and Table 3 is the result of selection sort. Both of insertion sort and selection sort have a nested for loop statement in the source codes. Thus the result of selection sort is similar to insertion sort. As the array length increases, the total iterations for CREST grow exponentially since there is a double nested loop statement bounded by a symbolic value. However in ALERT with loop dependence analysis, we can detect that the universal checks on malloc() function and memcpy() function are not dependent with the for loop statement. Thus the total iterations do not increase with array length. Table 4 is the result of shell sort and Table 5 is the result of bubble sort. Similar situations happen on Shell sort and Bubble sort too. Note that the result of Shell sort using ALERT with loop dependence analysis can not compare with CREST because CREST does not support division operation.

```
#include <crest.h>
#include <stdlib.h>
int insertion_sort(char *ary, int *len)
{
        int i, j;
        char value;
        for(i = 1; i < *len; i++){
                value = ary[i];
                j = i - 1;
                while(j >= 0 && ary[j] > value){
                ary[j + 1] = ary[j];
                        j--;
                }
                ary[j + 1] = value;
        }
        return 0 ;
}
int main(void)
{
        char ary[50];
        char *dest;
        int len;
        for(int i = 0; i < 50; i++)
                CREST_char(ary[i]);
        CREST_int(len);
        if(len < 51 && len > 0){
                insertion_sort(ary, &len);
                dest = malloc(len);
                memcpy(dest, ary, len);
        }
        return 0;
}
```

Figure 17: Insertion sort for CREST

```
#include "test.h"
#include <stdlib.h>
int insertion_sort(char *ary, int *len)
{
        int i, j;
        char value;
        for(i = 1; i < *len; i++){
                value = ary[i];
                j = i - 1;
                while(j >= 0 && ary[j] > value){
                ary[j + 1] = ary[j];
                        j--;
                }
                ary[j + 1] = value;
        }
        return 0 ;
}
void testme(char *ary, int len)
{
        char *dest;
        int k;
        if(len < 51 && len > 0){
                k = len;
                insertion_sort(ary, &k);
                dest = malloc(k);
                memcpy(dest, ary, k);
}
```

Figure 18: Insertion sort for ALERT + loop dependence analysis

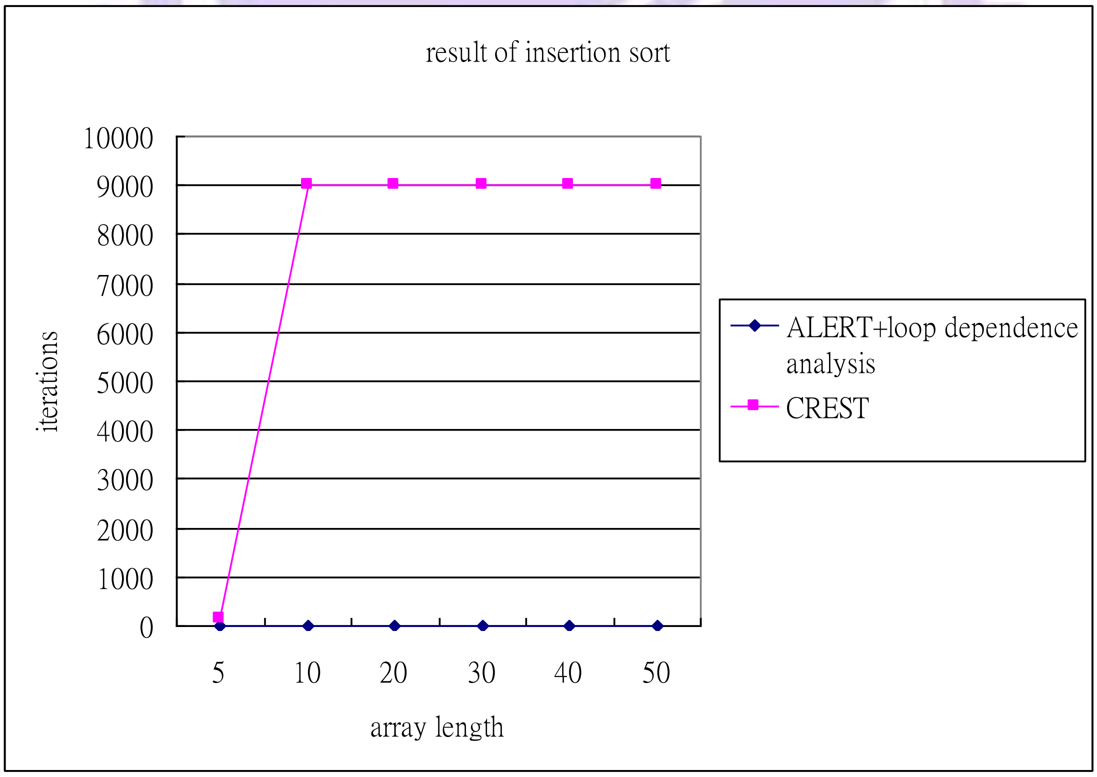
```
void f(void)
{ char ary[50] ;
        int len ;
        {
                _sqAdd((unsigned int )ary, sizeof(ary), "ary");
                _sqInputBytes(ary, 50);
                _sqInput(& len, T_INT);
                _sqMakeSymbolicBytes(ary, 50);
                _sqMakeSymbolic("len", T_INT);
                _sqPush("ary", (unsigned int )ary, T_UINT);
                _sqPush("len", len, T_INT);
                testme(ary, len);
        }
}
int main(int argc, char* argv[])
{
        _sqGetopt( argc, argv );
        _sqInit();
        f();
        int j = _sqSolve();
        _sqHistRecord(j+1, branch_hist);
        if (!isComplete) {
                printf("****************Generating next test path****************\n");
                return 0;
        } else {
                printf("********************Mission Complete!********************\n");
                return 1;
        }
}
```

Figure 19: Part of instrumented code for ALERT + loop dependence analysis

Table 1: Result of insertion sort

Array length	ALERT+UCOpt (iterations)	CREST (iterations)
5	5	155
10	5	>9000
20	5	>9000
30	5	>9000
40	5	>9000
50	5	>9000

Table 2: Result of insertion sort



```
#include "test.h"
void selectionSort(char *ary, int *len)
{
        int i, j, indx;
        char min, tmp;
        for(int i = 0; i < *len - 1; i++){
                indx = i;
                min = ary[i];
                for(j = i + 1; j < *len; j++){
                        if(min > ary[j]){
                                indx = j;
                                min = ary[j];
                        }
                }
                tmp = ary[i] ;
                ary[i] = ary[indx];
                ary[indx] = tmp;
        }
}
void testme(char *ary, int len)
{
        char *dest;
        int k;
        if(len < 41 && len > 0){
                k = len;
                selectionSort(ary, &k);
                dest = malloc(k);
                memcpy(dest, ary,k);
        }
}
```

Figure 20: Source code of selection sort

Table 3: Result of selection sort

Array length	ALERT+UCOpt (iterations)	CREST (iterations)
5	5	239
10	5	>9000
20	5	>9000
30	5	>9000
40	5	>9000
50	5	>9000

Table 4: Result of shell sort

Array length	ALERT+UCOpt (iterations)	CREST (iterations)
5	5	239
10	5	>9000
20	5	>9000
30	5	>9000
40	5	>9000
50	5	>9000

Table 5: Result of bubble sort

Array length	ALERT+UCOpt (iterations)
5	5
10	5
20	5
30	5
40	5
50	5

6.2 An n*n matrix allocation example

```
#include "test.h"
#include <limits.h>
void testme(int n)
{
        int i, **matrix, tmp;
        if(UINT_MAX / sizeof(int) > n && n > 0)
                matrix=(int **)malloc(n * sizeof(int*));
        if (!matrix)
                printf("allocation failure 1 in matrix()\n");
        for(i=0;i<n;i++) {
                if(UINT_MAX / sizeof(int) > n && n > 0)
                        matrix[i]=(int *)malloc(n * sizeof(int));
                if (!matrix[i])
                        printf("allocation failure 2 in matrix()\n");
        }
}
```

Figure 21: Source code of matrix allocation

Table 6: Result of matrix allocation using contradiction constraint detection

n	Eliminated universal checks (by contradiction constraint detection)
10	9
20	19
30	29
40	39
50	49

Table 7: Result compared between using constraint contradiction analysis and not using constraint contradiction analysis

n	Total iteration (ALERT+ loop dependence analysis + constraint contradiction analysis)	Total iteration (ALERT + loop dependence analysis)
10	4	13
20	4	23
30	4	33
40	4	43
50	4	53

Figure 21 is a program which allocates an n*n matrix. In line 7 it first allocate an array of integer pointer and line 10 to line 15 allocates n arrays of integer. There are vulnerabilities in line 7 and line 12 since the type of argument to malloc() function is unsigned integer and n*sizeof(int*) and n*sizeof(int) can be overflowed and allocates memory with wrong size to the program. Thus the input validation is required in line 6 and line 11. It protects n*sizeof(int*) and n*sizeof(int) against integer overflow. Table 6 is the result of ALERT using contradiction constraint detection. Variable n is symbolic. When program executes the for loop statement in line 10, universal check performs only at the first iteration since contradiction constraint detection will detects the universal checks are redundant in the rest iterations. Thus the number of eliminated universal checks increases with variable n. Another advantage of constraint contradiction analysis is it can kill unnecessary rest element. In Figure 21 line 7 and 12, the variable n will be collected into the rset, and loop dependence analysis will detect the universal checks are dependent with the loop statement in line 10 and the loop iterations can not be eliminated. However, if we use constraint contradiction analysis, it will detect that the variable n is already protected by the if statement in line 11 and the universal check on malloc() function is unnecessary. Thus variable n will not be collected into the rset, and the rset will not intersect with the wset and the loop iterations can be eliminated except the first iteration. Table 7 is the result of matrix allocation. We bound the range of variable n due to testing convenience. The first column is the range of variable n, the second column is ALERT with loop dependence analysis and constraint contradiction analysis, and the third column is ALERT with loop dependence analysis. As we can see we can eliminate the redundant universal check and loop iterations by constraint contradiction analysis. If we do not use the constraint contradiction analysis, the loop dependence analysis will not eliminate those redundant iterations since it determine the universal check is relative to the loop statement.

7 Conclusion

Scalability is an important issue in dynamic test generation. Loop bounded by symbolic variable is a main reason to bother the scalability. To address this problem we proposed loop dependence analysis to analyze the relationship between loop statement and property check and ignore the redundant property check and loop iterations. Besides we also observe that many property checks are unnecessary because they are already bounded by some mitigate. These mitigate will contradict the property check. We proposed a method called constraint contradiction to detect those contradictions and eliminate the unnecessary constraints. Finally we apply our method on security property checking, with the method's help security property checking is more scalable and lower the execution times while maintaining the same coverage.

References

[1] Patrice Godefroid, "Compositional dynamic test generation," in *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47-54, 2007.

[2] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill and Dawson R. Engler, "EXE: Automatically generating inputs of death," In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322-335, 2006.

[3] Koushik Sen, Darko Marinov and Gul Agha, "CUTE: A concolic unit testing engine for C," In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263-272, 2005.

[4] Bernard Elspas, Karl N. Levitt, Richard J. Waldinger and Abraham Waksman, "An assessment of echniques for proving program correctness," *ACM Computing Surveys (CSUR)*, vol. 4, pp. 97-147, 1972.

[5] Chittor V. Ramamoorthy and Siu-Bun Ho, "Testing large software with automated software evaluation systems," In *Proceedings of the International Conference on Reliable Software Table of Contents*, pp. 382-394, 1975.

[6] K. Krause, M. Goodwin and R. Smith, "*Optimal Software Test Planning through Automated Network Analysis," IEEE Symposium on Computer Software Reliability*, pp. 18-22, April 1973.

[7] William E. Howden, "Methodology for the generation of program test data," *IEEE Transactions on Computers*, vol. 100, pp. 554-560, 1975.

[8] James C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385-394, 1976.

[9] Patrice Godefroid, Nils Klarlund and Koushik Sen, "DART: Directed automated random testing," In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213-223, 2005.

[10] Eric Larson and Todd Austin, "High coverage detection of input-related security faults," In *Proceedings of the 12th USENIX Security Symposium (Security '03*), August 2003.

[11] Christoph Csallner and Yannis Smaragdakis, "Check'n'crash: Combining static checking and testing," In *Proceedings of the 27th International Conference on Software Engineering*, pp. 422-431*, 2005.*

[12] Willem Visser, Corina S. Păsăreanu and Sarfraz Khurshid, "Test input generation with java PathFinder," In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 97-107, 2004.

[13] Tao Xie, Darko Marinov, Wolfram Schulte and David Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," In *the Tools and Algorithms for the Construction and Analysis of Systems*, pp. 365-381, 2005.

[14] Neelam Gupta, Aditya P. Mathur and Mary Lou Soffa, "Generating test data for branch coverage," in *Proceedings of the International Conference on Automated Software Engineering*, pp. 219, 2000.

[15] Bogdan Korel, "A dynamic approach of test data generation," *IEEE Conference on Software Maintenance*, pp. 311--317, November 1990.

[16] Bogdan Korel, "Automated software test data generation," *IEEE Transactions of Software Engineering*, vol. 16, pp. 870-879, 1990.

[17] Nicholas Nethercote and Julian Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," In *Proceedings of the 2007 PLDI Conference*, pp. 89-100, 2007.

[18] Reed Hastings and Bob Joyce, "Purify: Fast detection of memory leaks and access errors," In *Proceedings of the Winter USENIX Conference*, http://www.rational.com/supprot/techpapers/fast_detection/, 1992.

[19] Patrice Godefroid, Michael Y. Levin and David A. Molnar, "Automated whitebox fuzz testing," In *Proceedings of the Network and Distributed System Security Symposium*, http://research.microsoft. com/users/pg/public_psfiles/ndss2008.pdf, 2008.

[20] Yang-Chieh Fan, "Resolving Constraints from COTS/Binary Components for Concolic Random Testing," 2007.

[21] Peter Boonstoppel, Cristian Cadar and Dawson Engler, "RWset: Attacking path explosion in constraint-based test generation," *Lecture Notes in Computer Science*, vol. 4963, pp. 351, 2008.