A Synergy between Static and Dynamic Analysis for the Detection of Software Security Vulnerabilities*

Aiman Hanna, Hai Zhou Ling, XiaoChun Yang, and Mourad Debbabi

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
ahanna@encs.concordia.ca, ha_ling@encs.concordia.ca,
xc_yang@encs.concordia.ca, debbabi@encs.concordia.ca

Abstract. The main contribution of this paper is a framework for security testing. The key components of this framework are twofold: First, a static analyzer that automatically identifies suspicious sites of security vulnerabilities in a control flow graph. Second, a test-data generator. The intent is to attempt proving/disproving whether, or not, the suspicious sites are actual vulnerabilities. The paper introduces the static-dynamic hybrid vulnerability detection system, a system that targets the automation of security vulnerability detection in software. The system combines the detection powers of both static and dynamic analysis. Various components compose this model, namely Static Vulnerability Revealer, Goal-Path-oriented System, and Dynamic Vulnerability Detector.

Keywords: Security Automata, Security Testing, Static Analysis, Dynamic Analysis, Test-Data Generation.

1 Introduction

Deployed software often carry various security vulnerabilities, some of which can be very severe if exploited. To mitigate the problem, a serious effort should be placed on software testing. However, testing is a nontrivial process, that usually results in a great deal of cost and time overhead, especially if conducted manually. Consequently, the automation of such process is becoming a necessity, which resulted in an increased effort by both academia and industry to address the issue. In spite of this effort, further work is yet needed to achieve efficient software security testing.

In this paper, we present the Hybrid Vulnerability Detection System (HVDS), which provides a solution towards the automation of software security testing.

* This research is the result of a fruitful collaboration between CSL (Computer Security Laboratory) of Concordia University, DRDC (Defence Research and Development Canada) Valcartier and Bell Canada under the NSERC DND Research Partnership Program.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 815–832, 2009.

In section 2, we introduce our static vulnerability revealer component, which is used to detect potentially vulnerable program paths. Section 3 provides a brief background on code instrumentation and highlights our code instrumentation model. Section 4 provides a brief background on the subject of test-data generation. Section 5 provides an introduction to our goal-path-oriented system, which traverses the potentially suspicious paths and attempts to generate test-data to force the execution of such paths. Section 6 provides a case study conducted over our framework. Finally, Section 7 provides a conclusion on the work presented in this paper.

2 Static Vulnerability Revealer

In this section we introduce the Static Vulnerability Revealer (SVR)[11] component. Given a source code of a program, and a formally specified security property, SVR main concern is to find out all (the largest set possible) program paths that have the *potential* of violating the security property in concern. We refer to these paths as *suspicious paths*. SVR is based on two major techniques, static analysis and model-checking.

Static analysis takes advantage of control flow, data flow, and type information generated by the compiler to predict undesirable behavior of programs. While static analysis is efficient in catching property violations that involve syntactically matching of program constructs, it is less suitable for analyzing system-specific properties. Model-checking technique excels static analysis in this aspect. However, a major drawback of model-checking techniques is the significant effort required to construct checkable program models, especially for large software systems. We tackle this problem by using static analysis technique to automate the model construction process, hence bringing static analysis and model-checking into a synergy, leveraging the advantages and overcome the shortcomings of both techniques.

SVR targets detecting security vulnerabilities in software source code. In order to provide multi-language support, we based our system on the GCC [16] compiler. Starting from version 4, GCC mainline includes the Tree-SSA [15] framework that facilitates static analysis with a universal intermediate representation (GIMPLE) common to all supported languages. SVR works on the GIMPLE representation and abstracts required information to construct program models.

Another component utilized in SVR is a conventional pushdown system model-checker called Moped [2], [12], which comes with a procedural language Remopla for model specification. Moped performs reachability analysis of a specific statement in the Remopla code. SVR exploits the reachability analysis capability for the purpose of verifying security properties. In the system, a security property is modeled as a team edit automaton, specifying the erroneous behavior using sequences of program actions. An `error` state is introduced to represent the risk state for each automaton. The automaton is then translated to Remopla representation and later synchronized with the pushdown system of the program in

concern. During verification, program actions would trigger the state changing of the automaton. If the `error` state is reached, a sequence of program actions violating the given property has been detected. In other words, SVR converts the security detection problem to a reachability problem.

Figure 1 gives an overall view of the SVR system, which integrates the aforementioned components. As shown in Figure 1. Different phases compose SVR system. The first phase addresses the property specification. A formal language is needed to allow a security analysts to specify security properties that they wish to test the software against. For this, we have utilized security automata. The concept of security automata was introduced in [17]. Security automata was presented as the basis of an enforcement mechanism within an execution monitoring system. An execution monitoring system is a system that would monitor program execution, possibly by running in parallel with the target program. Once a security-related action is to be executed, the monitor determines if the action conforms to the security property of concern. If so, the action is allowed; otherwise the monitor alters the execution so that the property is obeyed. A program monitor can be formally modeled by a security automaton, which is a deterministic finite or countably infinite state machine [14].

To ease the work done by the security analyst, we provide a graphical capability with which the security property can be stated. Each property is specified in security automata, and the specification also supports syntactical pattern matching of variables. Given the graphical representation, our tool automatically translates the specified properties into a Remopla specification, which then be used as part of the input to the Moped model-checker. The output of this phase is a Rempola model representing the given property.

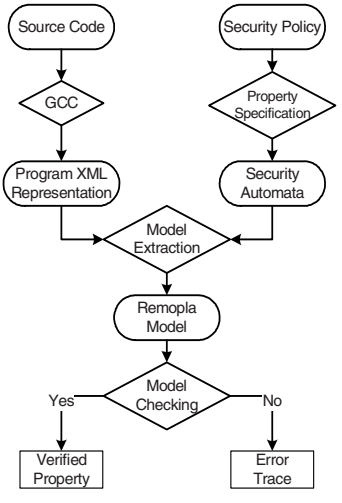


Fig. 1. Overall View of SVR System Architecture

The second phase addresses program model extraction. The input to this phase is the source code of the program under test, while the output is a Remopla model that synchronizes the Remopla representation of the program with the Rempola representation of the security property. Specifically, a pass is added to the GCC compiler where the GIMPLE representation is dumped into XML files, from which the program model is extracted and represented using Remopla. The generated program model is then combined with the Rempola representation of the property. The combination is done to serve the purpose of: (1) achieving synchronization between the program pushdown system and the security automata, (2) binding the pattern variables of security automata with actual values taken from the source code, and (3) reducing the size of the program's model by only considering program actions that are relevant to the specified security properties.

In the third phase, the resulting Remopla model from previous phases is provided as input to the Moped model-checker for reachability analysis. An error is reported when a security automaton specified in the model reaches an error state. The output of this phase is a set of suspicious paths that would possibly violate the security property in concern. It should be clearly noted that, due to the static analysis nature of SVR, some of the reported vulnerabilities are indeed positive reports of violation, while others could be false positives, an issue that we will revisit shortly.

2.1 Constructing Remopla Models

This subsection describes the construction of Remopla formal model in details, including the generation of the Remopla representation for both security properties and program under analysis.

Modeling Security Properties. The security property is specified using security automata, and we focus on temporal properties. A `start` node and an `error` node are introduced, respectively, to represent the initial and final risky states.

From Security Automata to Remopla. Given a property automaton, we serialize it into Remopla representation, which we also refer to as Remopla *automaton*. A Remopla automaton is represented using a Remopla module. As an example, Figure 3 shows the Remopla module of the security automaton in Figure 2. The nodes and the transition labels of a security automaton are mapped to Remopla constructs, as defined hereafter:

- Integers are used to identify the automaton nodes, each of which corresponds to an enumerator of a Remopla enumerated type (i.e. the enumeration variable `states` in Figure 3). For tracking the state of the automaton, an integer variable (i.e. `current_state` in Figure 3) is introduced and initialized to the automaton's initial state (i.e. `start` in Figure 3) using the Remopla keyword `INITIALIZATION`, following which the Remopla instructions are evaluated by the model checker at the beginning of the verification.

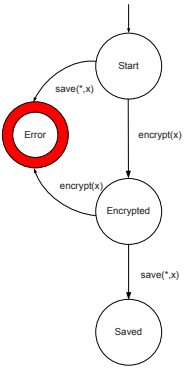


Fig. 2. Security property that specifies a file must be encrypted before being saved

– Transition labels are used to identify security relevant program actions. Table 1 shows the program actions we capture, together with the corresponding Remopla representation. The Remopla constructs prefixed with `ACTION_` are defined as elements of a Remopla enumeration type, which includes the relevant program actions. A transition is triggered if its label matches the input program action. A transition with the label in the first entry, for instance, is activated when the program action matches `ACTION_PROGRAM_START` (i.e. the `main` entry of a program). The second row represents the termination of program execution, and the next two entries denote respectively the entry point of function and its return. The mapping for a function call is defined in the fifth entry. Each function argument is an element of a global Remopla array `ARG[]` which is inquired during the model-checking process when function parameters are involved in the property verification. Notice that the mapping in the last entry, representing an assignment, is focused more on the program action but not on the data value being passed.

The translation from a security automaton to its Remopla representation follows the mapping defined in Table 1, and each security automaton is represented as a Remopla module. Figure 3, for example, shows the Remopla module for the security property in Figure 2. The module takes the current program action as input, checks it against the defined transitions, and changes the automaton state accordingly. For example, lines 8 to 14 represent the `start` node of the property in Figure 2. If the `action` matches `ACTION_FUNCTION_CALL_encrypt` with parameter `ARG_X`, the state is changed to `encrypted`. A violation of the given property is detected as long as the `error` state is reached.

Program Model Extraction. The model extraction is the process that translates program source code to Remopla representation. The translation consists of two phases: (1) The GIMPLE representation of parsed source code is converted and dumped into XML files, and (2) from which the Remopla model representing the source code is extracted. To reduce the size of the extracted program model, a preprocessing phase is incorporated before the model extraction. Since the

```
enum states{start, encrypted, saved, error};
int current_state;
INITIALIZATION: current_state = start;

move_state (int action)
{
  if
  :: current_state == start ->
      if
      :: action == ACTION_FUNCTION_CALL_encrypt
          && ARG[0] == ARG_X -> current_state = encrypted;
      :: action == ACTION_FUNCTION_CALL_encrypt ->
          current_state = error;
      :: else -> break;
      fi;
  :: current_state == encrypted ->
      if
      :: action == ACTION_FUNCTION_CALL_save
          && ARG[1] == ARG_X -> current_state = saved;
      :: else -> break;
      fi;
  :: current_state == saved -> break;
  :: current_state == error -> break;
  :: else -> break;
  fi;
}
```

Fig. 3. Generic Remopla representation of the automaton in Figure 2

Table 1. Remopla representation of program actions

Program Action	Remopla Representation
entry of program	ACTION_PROGRAM_START
exit of program	ACTION_PROGRAM_END
entry of f	ACTION_FUNCTION_CALL_f
return of f	ACTION_FUNCTION_RETURN_f
f(v_0,...,v_n);	ARG[0]=ARG_v_0;...;ARG[n]=ARG_v_n; f( );
var = v;	ACTION_VAR_MODIFICATION_var;

set of properties to verify has been specified, we have the knowledge of a set of security-related functions. By analyzing the call-graph of the program in concern, we are able to identify functions that are relevant to the verification, directly or indirectly, and hence the extracted model can preserve only security-relevant behavior and have a small size.

The first entry of Table 2 shows the Remopla construct for the control flow structure in source code. Note that each condition in the source code is represented using the Remopla keyword `true`. With such a condition, the model checker would choose either branch non-deterministically during verification, considering both branches are feasible. At this stage, we take into account all paths in the source code without pruning infeasible paths, which naturally leads to false positives. These false positives will later be eliminated by our model as we explain in section 5.

The translation from program constructs to Remopla representation follows the mapping defined in Table 2.

```
#include <stdio.h>
#include <stdlib.h>
...
void encrypt(char * buffer, char *key);
void save(FILE *outfile, char *buffer);

int main (int argc, char *argv[])
{
   ...
   printf("Please enter the key for encryption.\n");
   int keyval;
   scanf("%d", &keyval);
   if(keyval > 9999){
      if(keyval > 999999){
         printf("Invalid key.\n")
      }else{
         char *key = (char *)malloc(MAX_LEN);
         memset(key, '\0', MAX_LEN);
         snprintf(key, MAX_LEN, "%d", keyval);
         encrypt(buffer, key);
      }
   }else{
      printf("Invalid key.\n");
      exit(1);
   }
   save(outfile, buffer);
   printf("File %s has been encrypted & saved.\n", argv[1]);
   ...
   return 0;
}
```

Fig. 4. Sample source code to illustrate Remopla model generation

```
//Automata actions declaration:
enum actions {ACTION_FUNCTION_CALL_copy_to_user,
              FUNCTION_CALL_copy_from_user,
              ACTION_FUNCTION_CALL_access_ok,
              ACTION_PROGRAM_END};
//Function call possible arguments:
enum args {ARG_4,ARG_user_ptr2,
           ARG_0,ARG_user_ptr1,
           ARG_kernel_ptr};
//ARG array declarartion:
int ARG[10];
int current_state;

module void copy_from_user (){
  move_state(ACTION_FUNCTION_CALL_copy_from_user);
  return;
}
module void copy_to_user (){
  move_state(ACTION_FUNCTION_CALL_copy_to_user);
  return;
}
module void access_ok (){
  move_state(ACTION_FUNCTION_CALL_access_ok);
  return;
}
module void main (){
  ARG[0]=ARG_0;ARG[1]=ARG_user_ptr1;ARG[2]=ARG_4;
  access_ok();
  if
  :: true ->
     ARG[0]=ARG_kernel_ptr;ARG[1]=ARG_user_ptr1;ARG[2]=ARG_4;
     copy_from_user();
  :: else -> break;
  fi;
  ARG[0]=ARG_user_ptr2;ARG[1]=ARG_kernel_ptr;ARG[2]=ARG_4;
  copy_to_user();
  return;
}
```

Fig. 5. Remopla model of the source code in Figure 4

Table 2. Remopla representation of program constructs

Program Construct	Remopla Representation

`if(cond){`
`...`
`}else{`
`...`
`}`
	`if`
	`:: true -> ...;`
	`:: else -> ...;`
	`fi;`

`f(){`
`...`
`}`
	`module void f(){`
	`move_state(ACTION_FUNCTION_CALL_f);`
	`...`
	`move_state(ACTION_FUNCTION_RETURN_f);`
	`}`

`f();`	`f();`
`f(`v_0`,...,`v_n`)`	`ARG[0]=ARG_`v_0`;...;ARG[n]=ARG_`v_n`;f();`

As shown in the second entry, a function is represented as a Remopla module of type `void` and without parameters. Two important program actions are associated with each function: the entry of it and the exit of it. We express these actions explicitly and embed them in the program model. These actions are passed as parameters to the `move_state` module, which in turn checks them against the defined transitions and changes the automaton state accordingly. By sending the action to the property module, we establish the synchronization between the program state and the automaton state.

The last two entries defines respectively the mapping for function calls with and without parameters. Before a function is called, its arguments are stored in the global Remopla array `ARG[]`, indexed by the parameter's position in the function's signature. When the property module is checking a given program action, this global array is inquired for the passed parameters.

For example, Figure 5 shows the Remopla program for the sample code in Figure 4, which is a faulty program that violates the encrypt-save property; where a file must be encrypted before being saved. For clarity, Figure 5 shows only statements relevant to our discussion. Note that the security-irrelevant operations have been eliminated by the preprocessing phase mentioned above, and the generated model contains only security-relevant operations. The variables `actions` and `args` contain all the program actions and passed parameters, respectively. The initial state of the program corresponds to the initial state of the considered security automaton. The program model can be in one of the states defined in the given property automaton (i.e. the enumeration variable `states` in Figure 3), and the state of the model (i.e. `current_state` in Figure 5) is synchronized with the property automaton (i.e. the one in Figure 3) by invoking the `move_state()` module with the current program action as a parameter. In this example, line 32 in Figure 5 is reported by our tool as a violation of the property in Figure 3.

Suspicious Paths Detection and Reporting. SVR process concludes by detecting program paths that could potentially violate the security property in concern.

While SVR is one of the components composing our model, it should be noted that SVR can be executed as an independent security violation detection tool. However, due to the nature of static analysis, the detected vulnerabilities reported by SVR may very well include false positives. While this may be acceptable in some cases, in other cases it may be massive and overwhelmingly unacceptable. Consequently, the decision of whether, or not, to run SVR independently is left to the security analyst. Since our goal is to eliminate any reporting of false positives, we do refer to the detected paths as suspicious paths. We then inject those paths to the other components of our model, where dynamic analysis is conducted to verify the real existence of such reported vulnerabilities.

3 Code Instrumentation

For the purpose of test-data generation, and security policy enforcement, code instrumentation is necessary. We use the compiler-assisted approach for code instrumentation. The main rationale of this choice is the fact that compilers generally have the needed syntactic (e.g. abstract syntax tree) and semantic (e.g. typing and flow analysis) information of the program. This allows us to precisely select the exact program points where instrumentation should be performed.

The decision to go with the compiler-assisted approach, naturally raises another concern: which compiler? The answer to this question was GCC. Our decision was driven by multiple facts. GCC is a multi-platform compiler. The compiler also supports many programming languages including Java, C, C++, Objective C, Fortran, and Ada. In fact, GIMPLE is a GCC intermediate representation for many of these languages. Our code instrumentation is performed on the GIMPLE representation of the program.

In order to allow GCC to perform the needed instrumentations that we require, we extended the compiler. Our extension was applied over the core distribution of GCC for C programming language, version 4.2.0 [16]. The extension allows GCC to perform all original functionalities as well as the functionalities needed by our code instrumenter, which are detailed in section 5.1.

4 Test-Data Generation

Prior to proceeding with the other components of our framework, we provide a background on the subject of test-data generation. Various approaches have been proposed including random test data generation [3], directed random test data generation [10], path-oriented test data generation [4], [5], [8], genetic and evolutionary algorithms [6], [7], goal-oriented [13], and the chaining approach [9]. These approaches vary in nature and target different goals.

For our purpose of security vulnerability detection, none of these techniques is suitable. The reason behind that lies in the nature of security vulnerabilities. If these vulnerabilities are present in a software, then they are present at specific program points, which we refer to as *security targets*. Test-data generated by random testing may never lead to these points. Full-path coverage may result

into a massive effort being wasted in exploring paths in the software that are not at all related to the vulnerability targets in queston. Goal-oriented approach may succeed in generating data to reach the goal, but through an irrelevant path; in other words, a path that is not vulnerable. Consequently, to achieve our security detection goals, we had to design another test-data generation model, which we do refer to as the Goal-Path-oriented System (GPS).

5 Goal-Path-Oriented System

This component is concerned with the generation of test data, with which specific program points can be reached through specific paths. A simple definition of the problem can be stated as follows: *Given a target point t in a program, and a path p to reach that target, find program inputs* $x_1, x_2, \ldots, x_n$*, with which t can be reached through the execution of p.* Our view of the problem is driven by the fact that a specific security vulnerability, that violates a security property, presents itself at a specific program point. Yet, merely reaching this point through any execution path may not result in the violation of the security property.

The goal-path-oriented system accepts as input a set of suspicious paths from SVR. The system then attempts to generate test-data, with which these paths can be executed. Different phases compose the GPS process. The overall system architecture of GPS is shown in Figure 6.

The following subsections illustrate the various components and phases of the system. Besides the input suspicious paths from SVR, GPS process starts by

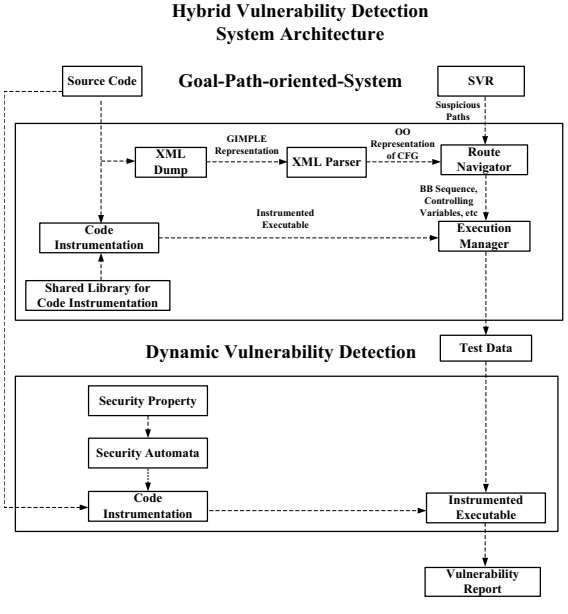


Fig. 6. GPS System Architecture

generating an XML-dump of the program. This XML-dump includes the GIMPLE representation of the program's CFG. This representation is then processed by an XML parser to produce and object-oriented (OO) representation of the CFG. This OO representation is finally input to the Path Navigator component for further processing, which also accepts the suspicious paths produced by SVR. In parallel to the processing that we just described, a different processing path over the source code representation is also performed by GPS. During that path, a code instrumentation process is applied over the code. The result of this processing is provided as an input to the execution manager, which performs the data generation itself.

5.1 Path Navigator

We tend to view the different paths of a software as routes of a city. Applying this concept of view, the problem of finding test-data that allow a specific route to be taken can be looked at as finding all relevant conjunctions along a route and determining which route to be taken at each of these conjunctions. Based on the same view, we also understand that there may be controlling traffic signals or gates along the route, where the values of these controlling elements determine the flow to the destination. We actually emphasize the same view in regard to the possible existence of a massive number of traffic lights and gates within the city. However, only a bounded number of them could be relevant to the specific route to be taken. Given a suspicious path, the goals of the path navigator component of our model can be described as follows:

- Determine the exact set of variables to be generated for a given path, i.e input variables; we refer to those variables as *relevant input variables*,
- Determine the exact set of controlling variables along conjunctions (path divisions). Those variables are the ones controlling conditional statements along the path. We refer to those as *controlling variables*,
- Determine, for every conditional statement, the exact set of input variables that is influential to the controlling variables. We refer to those variables as *master controlling variables*. The generation of data actually takes place for those variables,
- Determine the direction to be taken at each related conjunction to the path, we refer to these as *minimization directions*.

Consequently, further static processing is needed for the purpose of achieving these goals. These various operations are performed by the path navigator. Finally, the path navigator examines all the conjunctions along the given path. For each of these conjunctions, the navigator finds out the true/false evaluation value, with which the conjunction would lead to the continuation of the path traversal.

There are many advantages of the functions performed by the navigator in relation to the data-generation process. Firstly, since only relevant input variables along a path is considered, an optimization is obtained, especially in real-life

software that may include a large number of input variables. Secondly, the navigator detects all the variables that *indeed* control the execution of the path. In reality, those variables could be different than the ones directly appearing in the conditional statements themselves. Finally, the detection of the truth value of a conditional statement along the path is critical to our data-generation minimization process, as will be explained in subsection 5.2.

Section 3, provided a brief introduction of code instrumentation techniques. The code instrumenter component of GPS performs various instrumentation over the source code for various intended reasons. In general, the following instrumentations are performed by the component:

- Input Abstractor
- Execution Monitor

Below we briefly highlight the functionality of these different components.

Input Abstractor: To achieve the maximum automation possible, we need to abstract program statements that would require user interaction (i.e. `scanf()`). The function of the input abstractor is to instrument the original code, so that calls for data entries are abstracted by other calls that would perform the same functionality without any user interaction. Currently, our implementation provides an abstraction to only a limited set of all possible input methodologies, but extensions can be added.

Execution Monitor: Should the execution of the program leads to a critical path, then this execution is to be halted. In such case, information about the problem node must be collected to help the next attempt. In addition, we need to calculate a minimization value, at dynamic time, for each conditional statement in a given suspicious path. This is a value that is calculated in reference to zero. For instance, considering a program node with the following condition: `if (x > 12);`, and where the true branch is part of the suspicious path. Now, suppose at dynamic time the initial generated data for `x` is `0`. The minimization value is calculated as `(12 - x) = 12`. Our target is to make this minimization value less than `0`, so that the proper branch is executed. The following attempts would then try to make this minimization value less than `0`. The aforementioned functionality is provided by the execution monitor, which is instrumented into the code to achieve such functionality.

5.2 Execution Manager

This component represents the core of our dynamic analysis. Our model starts by converting all the constraints along a suspicious path to constraint functions. The path navigator detects, at static time, the truth value at each conditional statement along the path. The model then attempts to generate data that would allow the path to be followed. With such knowledge, the constraint function

can be constructed. Should a data generation results in the execution being driven away from the intended path, that execution is halted and information is collected for further attempts. In order to allow execution to passes through the failure node, the model needs to satisfy the set of constraint functions along the path so that the intended execution may succeed. To this end, the test-data generation can now be viewed as a constrained optimization problem.

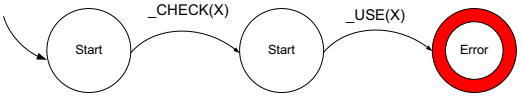
6 Case Study

The experiments aimed at applying our framework to various test suites of programs. All experiments were conducted over a core-2 PC with a clock frequency of 2.0 GHZ and 2 MB of level II cache under the following environments: Operating system: Ubuntu Linux release 6.10, Linux kernel: version 2.6.17-11-generic, C/C++: GCC 4.2, and Java runtime environment: Java(TM) SE Runtime Environment (build 1.6.0 02b05).

6.1 Security Properties to Be Validated

During the experiments, we defined a set of security properties based on which the programs under test can be verified. Those properties were as follows:

1. $RACE-CONDITION$: Time-of-check-to-time-of-use (TOCTOU) vulnerabilities are due to the elapsed time between check and use, which allows either an attacker, or an interleaved process/thread, to change the state of the targeted resource and yield undesired results[1]. To prevent TOCTOU race conditions that might be exploited by an attacker to substitute a file between the check (e.g. "stat" or "access" call) and the use ("open" call), a program should not pass the same file name to two system calls on any path. Figure 7 shows the automaton which represents the $RACE-CONDITION$ security property.



_CHECK(X) = access, stat, statfs, statvfs, lstat, readlink, tempnam, tmpnam, tmpnam_r

_USE(X) = acct, au_to_path, basename, catopen, chdir, chmod, chown, chrrot, copylist, creat, db_initalize, dbm_open, dbminit, dirname, dlopen, execl, execle, execlp, execv, execve, execvp, fattach, fdetach, fopen, freopen, ftok, ftw, getattr, krb_recvauth, krb_set_tkt_string, kvm_open, lchown, link, mkdir, mkdirp, mknod, mount, nftw, nis_getservlist, nis_mkdir, nis_ping, nis_rmdi, nlist, open, opendir, patchconf, pathfind, realpath, remove, rename, rmdir, rmdirp, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname

Fig. 7. $RACE-CONDITION$ Automata

2. $CHROOT - JAIL$: the *chroot*() function establishes a virtual root directory for the owning process. The main purpose of *chroot*() is to confine a user process to a portion of the file system so to prevent unauthorized access to system resources. Calls to *chroot*() requires root (super-user) access. If a programmer continues to run as root after the *chroot*() call, he/she opens up a potential vulnerability window for an attacker to use elevated privilege. Another problem with *chroot*() is that it changes root directory, but not the current directory. Therefore, program can escape from the changed root if calling *chdir*("/") is forgotten. Figure 8 shows the automaton which represents the $CHROOT - JAIL$ security property.

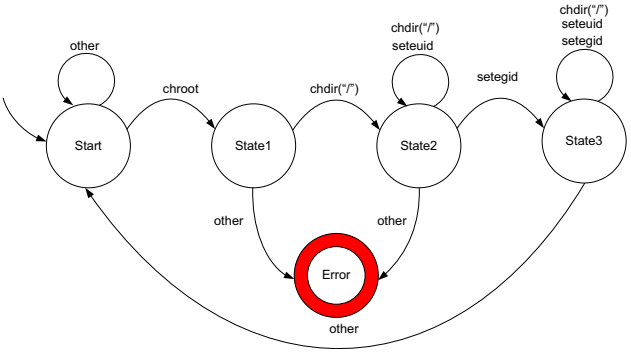


Fig. 8. $CHROOT - JAIL$ Automata

3. $MEMORY - MANAGEMENT$: for this security property, we targeted memory leaks, the use of an unallocated pointers and double deletion of pointers. Figure 9 shows the automaton which represents the $MEMORY - MANAGEMENT$ security property.
4. $STRCPY$: $strcpy(dest, src)$ is a classic call that is vulnerable to buffer overflow attacks. The destination buffer must be big enough to hold the source string plus the null terminating character. Figure 10 shows the automaton which represents the $STRCPY$ security property.
5. $TEMPNAM - TMPFILE$: this security property is drawn based on the coding rule $TEMPNAM - TMPFILE$ from CERT[1]. software applications often use temporary files for information sharing, temporary data storing, and computation speeding up. However many applications terminate execution without cleaning these files, which gives attackers a chance to hijack private and sensitive data. Additionally, as a temporary file is usually created in a shared folder, the appropriate permissions should be set to these files so to ensure the protection against attackers. As such, a call to $umask(077)$ must be done before a call to *mkstemp* to make sure that only the owner can access these files. This security property is illustrated in Figure 11.

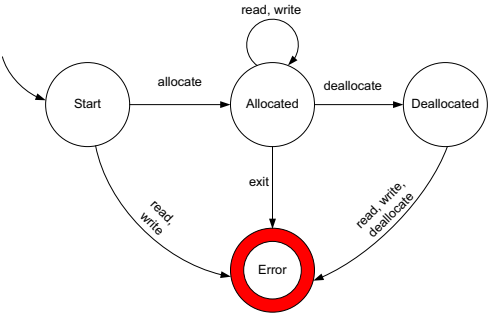


Fig. 9. *MEMORY – MANAGEMENT* Automata

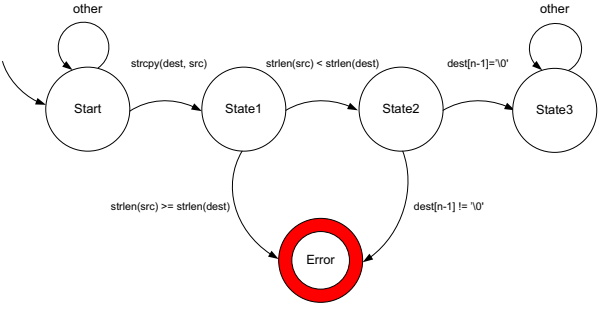


Fig. 10. *STRCPY* Automata

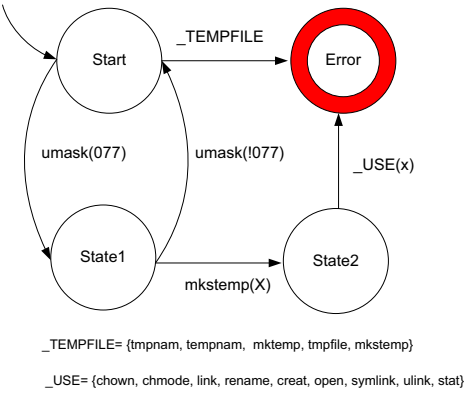


Fig. 11. *TEMPNAM – TMPFILE* Automata

6.2 Programs under Test

We conduct the experiments on a suit of C programs, which contain violations of the above-mentioned security properties. This suit contains five programs as follows:

1. Program 1: this program has a $RACE-CONDITION$ vulnerability, which is triggered when the program calls a *check* function: *access* followed by a *use* function: *open*;
2. Program 2: this program contains a violation of $CHROOT-JAIL$ security property. The program has a vulnerable path, where the *chroot* is not used in a correct way;
3. Program 3: this program has a $MEMORY-MANAGMENT$ vulnerability. Memory leak occurs when execution terminates without releasing allocated memory ;
4. Program 4: this program is vulnerable to buffer-overflow attacks. It violates the security property $STRCPY$;
5. Program 5: this program manages temporarily files using vulnerable functions. It has a violation of $TEMPNAM-TMPFILE$ security property.

In order to prove or disprove the existence of the vulnerabilities, reachability testings are performed. The results were compared to random testing, which is often used as a benchmark. The experimental results are shown in tables 3 and 4. Two measurements were used to evaluate the performances of our approaches: the total time used for the whole test data generation process, and the number of iterations, which it is worth noting that this is in fact how many times the program under test is executed.

As shown in Table 3, random testing approach was able to generate test data. However, the time consumed for each case varies dramatically. The result

Table 3. Experimental Result using Random Testing

Program name	**Security property to be verified**	**Time (millisecond)**	Iterations
Program 1	$RACE-CONDITION$	329325	151
Program 2	$CHROOT-JAIL$	5234254	2057
Program 3	$MEMORY-MANAGMENT$	95520	175
Program 4	$STRCPY$	27249942	9763
Program 5	$TEMPNAM-TMPFILE$	17239761	6560

Table 4. Experimental Result using Hybrid approach

Program name	**Security property to be verified**	**Time (millisecond)**	**Iterations**
Program 1	$RACE-CONDITION$	1413	13
Program 2	$CHROOT-JAIL$	1493	13
Program 3	$MEMORY-MANAGMENT$	5757	12
Program 4	$STRCPY$	140800	37
Program 5	$TEMPNAM-TMPFILE$	502532	121

shows that the performance of random approach is unstable and unpredictable. Additionally, whenever there is an "equal to" condition along the suspicious path, the performance of random testing decreased a lot. Theoretically, random approach is able to generate test data for any feasible suspicious path if time is not of concern; however, in reality, time matters.

The experimental result table 4 shows that our hybrid approach is stable in generating data, and that is capable of generating test data in a reasonable time. Furthermore, the experiments clearly indicate that the system is capable of performing security testing in a systematic way: from specification to detection and verification. The number of iterations and the time used depend on factors like number of constraints of a suspicious path, the execution time of the program, the number of codes instrumented, and the relationship (linear or non-linear) between input variables and control variables.

7 Conclusion

We presented in this paper a framework that utilizes both static and dynamic analysis for the purpose of detecting software security vulnerabilities. The framework targets detection and automation for programs where the software is available; i.e. Free and Open-Source Software (FOSS). Various components compose our model, with which some do heavily depend on static analysis, while the others are fully based on dynamic analysis. Additionally, other components were introduced for the purpose of security property specification and code instrumentation. The synergy of such approaches and methodologies have resulted in a promising model that moves towards the automation of security vulnerability detection of software.

References

1. Build Security (access on April 29, 2009), `https://buildsecurityin.us-cert.gov/daisy/bsi/home.html/`
2. Securityfocus (access on February 26, 2009), `http://www.securityfocus.com/bid/27796`
3. Bird, D., Munoz, C.: Automatic generation of random self-checking test cases. IBM Systems J. 22(3), 229–245 (1982)
4. Boyer, R., Elspas, B., Levitt, K.: Select - a formal system for testing and debugging programs by symbolic execution. SIGPLAN Notices 10(6), 234–245 (1975)
5. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself (March 2005)
6. Chakraborty, M., Chakraborty, U.: An analysis of linear ranking and binary tournament selection in genetic algorithms. In: International Conference on Information, Communications and Signal Processing. ICICS (September 1997)
7. Cigital and National Science Foundation. Genetic algorithms for software test data generation
8. Clarke, L.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering 2(3), 215–222 (1976)

9. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transaction on Software Engineering and Methodology 5, 63–86 (1996)
10. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing (June 2005)
11. Hadjidj, R., Yang, X., Tlili, S., Debbabi, M.: Model-checking for software vulnerabilities detection with multi-language support (October 2008)
12. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped - a model-checker for pushdown systems, http://www.fmi.uni-stuttgart.de/szs/tools/moped/
13. Korel, B.: Automated software test data generation. IEEE Transactions on Software Enfineering 16(8) (August 1990)
14. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors (January 2005)
15. Novillo, D.: Tree ssa: A new optimization infrastructure for gcc. In: Proceedings of the GCC Developers Summit3, pp. 181–193 (2003)
16. GNU Project. GCC, the GNU Compiler Collection, http://gcc.gnu.org/
17. Schneider, F.B.: Enforceable security policies. ACM Transaction of Information System Security (2000)