Towards Automation of Testing High-Level Security Properties

Aiman Hanna, Hai Zhou Ling, Jason Furlong, and Mourad Debbabi *

Computer Security Laboratory, CIISE,
Concordia University, Montreal (QC), Canada
{ahanna@encs.concordia.ca,ha_ling@encs.concordia.ca,
furlong.jc@forces.gc.ca,debbabi@ciise.concordia.ca}

Abstract. Many security problems only become apparent after software is deployed, and in many cases a failure has occurred prior to the awareness of the problem. Although many would argue that the simpler solution to the problem would be to test the software before deploying it. Although we support this argument, we understand that it is not necessarily applicable in a modern development environment. Software testing is labor intensive and is very expensive from a time and cost perspective. While much research has been undertake to automate software testing, very little has been directed at security testing. Additionally, the majority of these efforts have targeted low-level security (safety) instead of high-level security. In this paper, we present elements of a solution towards automation of testing security properties and for the generation of test data suites for detecting security vulnerabilities in software.

Key words: Security Testing, Dynamic Analysis, Data Dependency, Test Data Generation, Control Flow Analysis.

1 Introduction

When Von Neumann published his famous architecture in June of 1945, as part of the first draft of EDVAC, he might have anticipated, the potential power of that architecture. The evolution from a strictly single-function piece of hardware to a system that can behave in dramatically different ways through the use of software is is a significant technological advancement that has been exploited by virtually every major industry. However, it is unfortunate that the exponential growth of software in the past few decades has not met with an equivalent, or even relative, growth of concern with respect to software security. Although the problem is much more visible today than few years ago, security problems are still present even in most trusted software, such as operating systems. Many security problems only become apparent after software is deployed, and in many cases a

* 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.

failure has occurred prior to the awareness of the problem. Software testing is the most prominent way to eliminate many of these problems. While considerable research energies have been expended to automate software testing, very little has been directed at security testing. In order to find a security vulnerability in a program, four questions need to be answered:

1. What security property need to be tested?
2. How can security analyst state the property in concern?
3. How vulnerabilities can be located?
4. How test data can be generated to prove that a vulnerability not only exists, but will indeed take effect?

This list emphasizes that the range and nature of possible security vulnerabilities in software is very broad and detecting one of these vulnerabilities or another may require totally different approaches. Many vulnerabilities can be detected through static analysis of the source code. For instance, the password aging vulnerability, which occurs when a system does not enforce the policy that passwords need to be changed over time, has the potential to diminish password integrity. Yet, static analysis is sufficient for detecting vulnerabilities such as these. This can be achieved by checking for the existence of routines that validate the timestamp on passwords and then ensuring that the system utilizes these routines. While static analysis can be very useful in detecting many types of vulnerabilities, others will remain hidden. In such cases, dynamic analysis is needed. The focus of our research is on the detection of security vulnerabilities where dynamic-analysis needs to be conducted.

The answer to the first question falls within the domain of the security analyst. In answering the second and third questions, we have previously introduced extensions to GCC for code instrumentation, as well as Team Edit Automata (TEA) [14]. Used together, these promise to be a powerful tool for the analyst to state security properties, both formally and efficiently for the detection of a wide range of safety and security vulnerabilities.

In this paper, we provide elements of an answer to the fourth question, which concerns the generation of test data for testing security vulnerabilities. Previously published research has focussed on the following approaches: random test data generation [1], directed random test data generation [9], genetic and evolutionary algorithms [5, 6], path-oriented test data generation [2, 4, 7], goal-oriented [10], and the chaining approach [8]. These approaches use different types of information to achieve their goals: Some of them rely on the control flow of the program, while others rely on data dependency to guide their search process. While we highly regard each of these approaches, we need to point that the classification of these approaches as whether or not they are viable depends on the desired outcome. Path-oriented approaches, which rely on control flow analysis, can be viewed as very useful if full path coverage is needed. If the desired goal is to achieve a specific program target, then a path-oriented approach may be very inefficient since a lot of search effort may be wasted exploring parts of the program that have no relation to the target. Goal-oriented approaches, which attempt to lead program execution towards a specific target may also fail for

the same reason because they too depend on control flow analysis. Frequently, finding approaches that are both efficient and exacting, require executing parts of the program that are seemingly, from control flow graph perspective, unrelated to the solution [11]. The chaining approach uses both control flow and data dependency analyses that generate test data to reach designated targets in the code.

While none of these approaches targets security testing specifically, we view the chaining approach as as the most likely to facilitate efficient security testing. The nature of many security vulnerabilities is such that they tend to occur at identifiable locations in the code which we will designate *security targets*.

In the next section, we briefly present the chaining approach and its limitations in terms of testing for security properties. Section 3 provides a brief description of some of the most significant low-level and high-level security vulnerabilities. In Section 4, we present the security chaining approach. Section 5 provides an overview of our system and finally, Section 6 provides a conclusion of the work presented in this paper.

2 The Chaining Approach

The main goal of the chaining approach [8] is to find a data set with which a program execution can reach a specific node, referred to as the target node. The target is a node in the Control Flow Graph (CFG), which represents the objective of the test analysis. A simple definition of the approach is as follows: Given node Y in a program, the goal is to find a program input x on which node Y will be executed. The approach is an extension to the goal-oriented approach [12]. The goal-oriented approach classifies the different branches of a program as: *critical*, *semi-critical*, *non-essential*, and *required*. A branch is critical if and only if the execution of this branch would permanently drive the execution away from the target node. A semi-critical branch would also drive the execution away from the target node, but not permanently; i.e. through the execution of the back branch of a loop, the program execution may return back to a previous node where alternative branches leading to the target can be taken. A branch is a non-essential if the execution, or non-execution, of this branch does not affect reaching the target. A required branch is a branch that must be executed for the program to reach the target. To illustrate the approach, consider the C++ code fragment given in Figure 1, and its corresponding control flow graph shown in Figure 2.

Since goal-oriented approach relies merely on control flow analysis, both branches to nodes 6 and 8 are considered to be non-essential. That is the case since the execution of either branch will eventually lead back to node 4, and assuming that the loop at that node is not infinite, the execution will either way move towards the target, node 12. The goal-oriented approach will fail since the execution of node 9, which is treated as a part of a non-essential branch, is actually vital to reaching the target.

```
void GoOrientedApproach() {
1    int i1 = 0, i2 = 0, i3 = 0, i4 = 0, i5 = 0, i6 = 0;
2    cout << "Enter 4 Integers: ";
3    cin >> i1 >> i2 >> i3 >> i4 >> i5;
4    while (i1 < 10) {
5         if (i5==12) {
6              cout << "Point1" << endl;
7              i6++;
          }
          else {
8              cout << "Point2" << endl;
9              i1 *= 10;
          }
10        i1++;
     }
11   i2++;
12   if (i2 < 10) {
13        i3 = i2 - 1;
14        if (i3 % 2 == 0) {
15             cout << "Point3" << endl;
16             i2++;
17             if(i3 > 5) {
18                  if (i6 > 0) {
19                       cout << "Target Point" << endl;
                    }
                    else {
20                       i3 -=5;
21                       cout << "Point 4" << endl;
                    }
22                  i3--;
               }
          }
     }
     else {
23        i4 = i3--;
24        i5 = i2 + 4;
25        if (i4 == i5) {
26             cout << "Point9" << endl;
          }
          else {
27             cout << "Point10" << endl;
          }
28        cout << "Point11" << endl;
     }
}
```

Fig. 1. Sample Source Code

The chaining approach overcomes this shortcomings by extending the goal-oriented approach to consider data analysis as well. The approach views the problem as the set of one goal and multiple subgoals. Assume that data generation is required for some variable in order to reach the target. The approach starts in an identical fashion to the goal-oriented; it randomly selects a input value x_0 and executes the program with this value, then monitors the execution to detect if a critical path is reached. If the execution leads to the target then the goal has been reached and x_0 is the solution to the test data generation problem. However, if a violation occurs along the execution path; that is, a critical path is executed, the approach terminates the execution and considers the node where execution led to a critical branch to be a *problem node*. The focus of the approach at that point shifts from the goal to the subgoal, which is passing through the problem node towards the target. To solve the subgoal, the chaining approach

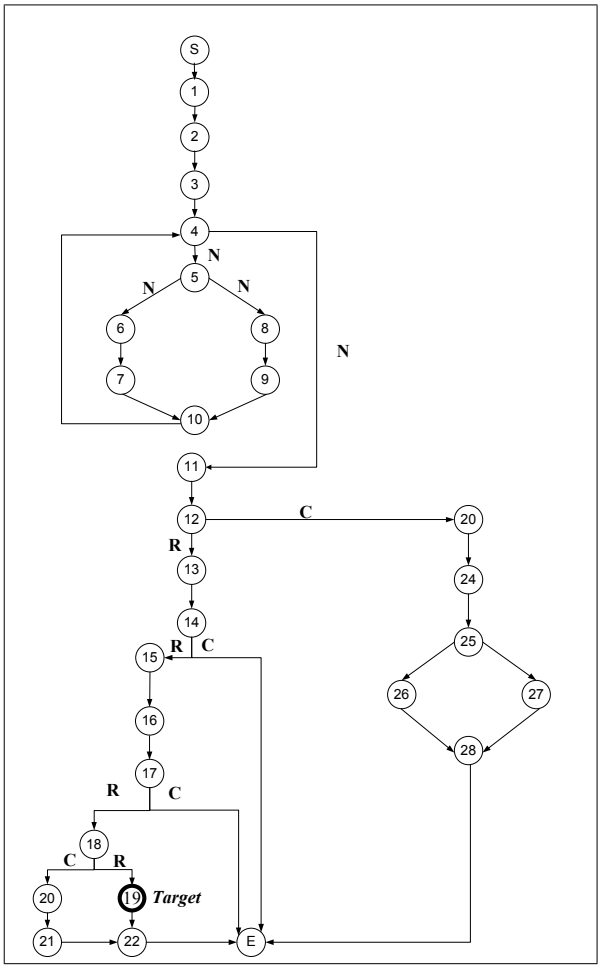

Fig. 2. Corresponding Control Flow of Code in Figure 1

uses a function minimization technique to find an alternative value that will execute the program at the problem node. If a value is found then the execution continues, with the possibility of hitting another problem node whereupon the process recursively repeats itself. If, however, function minimization fails to find an alternative value, then the approach switches to a data dependency analysis. To illustrate the idea, we must recall some of the basic concepts considered by the approach as given in [8].

– A flow graph of program Q is a directed graph $C = (N, A, s, e)$ where N is a set of nodes, A is a binary relation on N (a subset of $N \times N$), referred to as a set of edges, and s and e are, respectively, unique entry and unique exit nodes, $s, e \in N$.

- A *node* in N corresponds to the smallest single-entry, single-exit executable part of a statement in Q that cannot be further decomposed.
- An *edge* $(n_i, n_j) \in A$ corresponds to a possible transfer of control from instruction n_i to instruction n_j.
- An *edge* (n_i, n_j) is called a *branch* if (n_i) is a test instruction. Each branch in the control flow graph can be labeled by a predicate, referred to as a branch predicate, describing the conditions under which the branch will be traversed.
- A *use* of variable v is a statement (or predicate) that uses (references) this variable, such as `y=v+1; print(v); if (v!=0){..},` etc.
- A *definition* of variable v is a statement that assigns a value to this variable, such as `v=15; input (v);` etc.
- Let $U(n)$ be a set of variables whose value are used at node n, and let $D(n)$ be a set of variables whose values are defined at n. There exists a data flow (data dependence) between statement S_1 and S_2 if: (1) S_1 is a definition of variable v, (2) S_2 is a use of variable v, and (3) there exists a path in the program from S_1 to S_2 along which v is not modified.
- A *definition-clear* path from n_{k_1} to n_{k_q} with respect to variable v is a path in the control flow graph, such that: (1) v is defined at n_{k_1}, (2) used at n_{k_q}, and (3) it was not modified along the path between n_{k_1} and n_{k_q}; more formally $1 < i < q, v \notin D(n_{k_i})$.
- *Last definition*: Let p be a node and v be a variable used in p. Last definition of v at node p is defined as follows: A node n, which satisfies the following conditions: (1) v belongs to $D(n)$, (2) v belongs to $U(p)$, and (3) there exists a definition-clear path of v from n to p. Consequently, a *set of last definitions* $LD(p)$ is defined as the set of all last definitions of all variables used in p.

Now, let us revisit Figure 1. The chaining approach starts executing the program with an initial random value x_0. If this input value leads to the target then a solution is found. Assume however, that the execution successfully reaches node 18 but then the critical branch is taken at that node. The approach then attempts to solve that first subgoal, which is to find a value that will still preserve the execution to go all the way to node 18 (this is a constraint), but then changes the execution at that problem node p. Consequently, this is a minimization problem with constraints. If the attempt is successful, then a solution is found; otherwise, the approach attempts to alter the execution at node p by identifying the nodes that have to be executed prior to reaching this node. Effectively, the approach finds a set $LD(p)$ of last definitions of all variables used at problem node p then requires that these nodes be executed prior to the execution of p. By enforcing such a requirement, the chances of altering the execution flow at a problem node may be increased, and hence the desired branch is taken. Such a sequence of nodes to be executed is referred to as an *event sequence* (or *chain*).

An event sequence E is a sequence $\langle e_1, e_2, .., e_k \rangle$ of events, where each event is a tuple $e_i = (n_i, S_i)$ where n_i is a node and S_i a set of variables referred to as a *constraint set*. For every two adjacent events, $e_i = (n_i, S_i)$ and $e_{i+1} = (n_{i+1}, S_{i+1})$ there exists a definition-clear path with respect to S_i from n_i to n_{i+1}.

Generally, event sequences are generated as follows. Initially, for a given target node g, the following event sequence is created: $E_0 = \langle (s, \phi), (g, \phi) \rangle$ If during program execution, a problem node p is encountered, then: First, find all last definitions at p, $LD(p) = (d_1, d_2, ..., d_N)$, where d_i is a node where last definition of variables at p occurred. Second, Use that set to generate N event sequences. Each newly generated event sequence contains:

- An event associated with problem node p, and
- An event associated with last definition d_i

Consequently, the following event sequences are generated:

$$\begin{gathered} E_1 = \langle (s, \phi), (d_1, D(d_1)), (p, \phi), (g, \phi) \rangle \\ E_2 = \langle (s, \phi), (d_2, D(d_2)), (p, \phi), (g, \phi) \rangle \\ \vdots \\ E_N = \langle (s, \phi), (d_N, D(d_N)), (p, \phi), (g, \phi) \rangle \end{gathered}$$

The approach then selects one of the chains and attempts to find a solution. If another problem node occurred in that chain, a similar list is made as above with this new problem node and its previous LD node included in the chain. For instance, assume E_1 is selected, and that another problem node p_1 is encountered in the execution of E_1. Assume $LD(p_1) = (f_1, f_2, \ldots, f_M)$, then the following event sequences are created:

$$\begin{gathered} E_{1_1} = \langle (s, \phi), (f_1, D(f_1)), (p_1, \phi), (d_1, D(d_1)), (p, \phi), (g, \phi) \rangle \\ E_{1_2} = \langle (s, \phi), (f_2, D(f_2)), (p_1, \phi), (d_1, D(d_1)), (p, \phi), (g, \phi) \rangle \\ \vdots \\ E_{1_M} = \langle (s, \phi), (f_M, D(f_M)), (p_1, \phi), (d_1, D(d_1)), (p, \phi), (g, \phi) \rangle \end{gathered}$$

The process repeats which effectively results in a search tree being created, where E_0 is the root and any other generated event sequence is a child. The chaining approach traverses that search tree in a depth-first fashion, attempting to find an event sequence E for which a program input that executes that selected event sequence is found. A general search tree generated by the approach is partially shown in Figure 3.

3 Low-Level and High-Level Security

The range and nature of security vulnerabilities in software is quite broad. However, at higher abstractions, security vulnerabilities can be classified as either low-level (safety), or high-level (security). For the sake of brevity, we will only indicate some of the most significant.

3.1 Low-Level Vulnerabilities - Safety

Examples of low-level security vulnerabilities include: Buffer Overflow,Heap-based Exploitation, Stack-based Exploitation, Integer Overflow, File Management, and Memory Management.

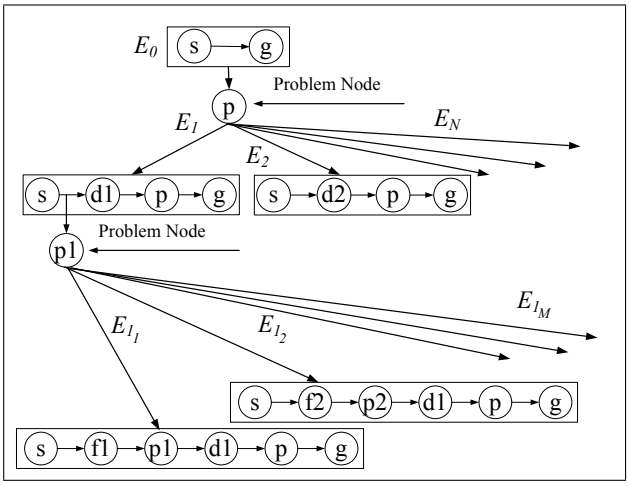

Fig. 3. A Partial Search Tree Generated by the Chaining Approach

3.2 High-Level Vulnerabilities - Security

Examples of high-level security vulnerabilities include: Authentication, Privilege Escalation, Inappropriate Authorization, Access Control, Integrity, Confidentiality, Non-Repudiation, Availability, and Cryptographic Vulnerabilities.

4 The Security Chaining Approach

Since the range of security vulnerabilities is quite varied, we must emphasis that a single solution capable of handling all types of vulnerabilities is not feasible. Different solutions for handling specific vulnerabilities, or a group of vulnerabilities, remains within the realm of possibility. In previous work [14], we presented *Team Edit Automata* (TEA) as a powerful model for stating and enforcing safety and security properties. TEA is partially based on *Security Automata* [13] which is proven capable of enforcing all safety properties as well as a limited set of security properties. Building on this research, we will now extend it to facilitate the automatic testing of security vulnerabilities through the security chaining approach.

While we regard the chaining approach well suited for test-data generation, the approach may fail, as it is not intended for security testing. There are many cases where the reachability of a target is insufficient for the detection of security vulnerabilities. To illustrate the idea, let us look at the simple example in Figure 4; the control flow graph corresponding to that code is shown in Figure 5. The program verifies user's role, solicits a PIN from the user, encrypts that PIN, and then sends the encrypted PIN over a network if certain conditions are met. The security analyst is interested in testing the software against a specific security property: All PINs sent over the network must be encrypted.

Clearly, node 35, where the encrypted PIN is sent, is "a" target here. It is also clear that there are multiple paths from start to this node. From the chaining approach point of view, there is only one goal, which is to generate test data to reach this node. However, from security testing point of view, there are multiple goals that must be achieved in parallel to detect any vulnerability. One goal is still to reach node 35. If this node is not reachable, then this code suffers from the availability security vulnerability. Another goal that must be considered is the path taken to reach this target. When the chaining approach attempts to generate test data, it may go through the usual process of hitting problem nodes, attempting to alter executions, generating search trees and traversing it in depth-first fashion. A successful analysis may generate the following: $x1 = 15, x2 = 1, x3 = 5, x4 = 20, x5 = 10, x6 = 75$, which allows the program to traverse to the target node through the following path: $S \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 18 \rightarrow 32 \rightarrow 34 \rightarrow 35$. However, such an execution does not suffer any security problems since the PIN would be encrypted at node 13 before being sent at node 35. One possible solution to the problem is to use the chaining approach to generate test data to reach the target, while concurrently executing a finite state machine (FSM) to monitor the status of the encrypted PIN, $ePass$, so that it can only be sent if it is in an encrypted state (that is being assigned a returned value from the *encrypt* function); otherwise the FSM enters an error state. However, this solution will fail for the same reason. If the data generated by the chaining approach is as above then the FSM will not detect any vulnerabilities. Another approach to test the security property in concern, is to use static analysis. While this may work, there is the potential of reporting false negatives. Since static analysis does not require program execution, it has no knowledge of the reachability of a specific point. Consequently, static analysis would report all potential problems, including false positives, which is not scalable to a real-life application. The security chaining approach eliminates these problems altogether.

In addition to the primary goal, which is the reachability of the target, the security chaining approach considers other goals. Specifically, this approach considers another type of event sequence (chains), referred to as the *security chain*, which is directly related to the security property itself. The following basic concepts are introduced by our approach:

- *Security Target*: A security target t is a node that must be: (1) reached, and (2) directly affecting/controlling the security property under test.
- *Last Security Definition*: Let s be a statement related to the security property under scrutiny and v be a variable used in s. Last security definition of v at statement s is defined as follows: A statement n, which satisfies the following conditions:(1) v belongs to $D(n)$, (2) v belongs to $U(s)$, and (3) there exists a definition-clear path of v from n to s. Consequently, a *set of last security definitions LSD*(s) is defined as the set of all last definitions of all security related variables used in s.
- *Undesired Last Security Definition*: Let s be a security target statement - a statement related to the security property under scrutiny. Undesired Last

Security Definition is a statment/node n such that if execution goes through that node, no security vulnerability would occur at the security target, s. Consequently, a *set of undesired last security definitions* $ULSD(s)$ is defined as the set of all undesired last definitions of all security related variables used in s.

- A path in a CFG is classified as either *critical* or *required*. A path is critical if and only if: (1) the execution of such a path would permeably drive execution away from the target node, "OR" (2) the path includes a node n that belongs to $ULSD(s)$, where s is the target node. A required path is a branch that: (ι) must be taken in order to reach a target, and ($\iota\iota$) is a part of a path to the target that does not include critical branches.
- *Predomination*: A node n predominates a node k if and only if: (1) There is a path from n to k, and (2) There is no possible way for node k to be reached unless node n is reached.

To illustrate the idea, let us revisit Figure 4 and its CFG shown in Figure 5. The first goal of the approach is to reach node 35, where the security target, the variable *ePIN*, is present. However, reaching that target must be forced through a very specific path for the approach to report a security vulnerability at the given code. The approach starts in an identical fashion to the chaining approach, flagging all critical paths that would permanently lead execution away from the target. The approach then, through code instrumentation and static analysis, detects all the $LSD(\_)$ of the target node (that is $LSD(statement\_at\_node\_35)$ in our example). The approach then flags all undesired last definitions. In our example, there are five $LSD(\_)$ located at nodes 9, 13, 16, 24 and 28. However, nodes 9, 13, 24 and 28 are all flagged as undesired last definitions, since the execution of these nodes would immediately lead to the security property being preserved. At that moment, the approach has one final goal, which is to generate test data to reach node 35 through node 16.

The approach definition of critical paths is significantly different than the one defined by the chaining approach. After the security chaining approach finds an undesired last definition, the approach finds the immediate test node that predominates this node and flags the branch from this test node towards the undesired last definition node as critical. Effectively, all undesired execution paths are eliminated. The approach then attempts to generate test data, in a similar fashion to the one used by the chaining approach. If the generation is successful, then a solution is found; which means a security vulnerability is detected. If all attempts fail to generate the test data, then the path is treated as impossible and no vulnerability is detected by the approach. It should be noted that false negatives are eliminated here since either the approach would report the problem with a set of test data that proves its existence, or nothing is reported if a path is thought of to be impossible after the approach has exhausted all attempts. It should also be noted that the order of flagging paths as critical is important, since it significantly reduces the search overhead. For instance, node 21 in Figure 4 is a chaining approach last definition of variable $x6$ at node 32. However, attempts to alter execution at node 32, should it become a problem

```
void EncryptAndTransmit() {
1   long int pin, epin;
2   int x1, x2, x3, x4, x5, x6;
3   cin >> x1 >> x2 >> x3 >> x4 >> x5 >> x6;
4   if (x1 > 10) {
5       x4++;
6       if(x2 > x3) {
7           cout << "User Detected as Admin. Enter Admin PIN:";
8           cin >> pin;
9           epin = encrypt(pin);
        }
        else {
10          if(x4 > 12) {
11              cout << "User Detected as Controller. Enter Controller PIN:";
12              cin >> pin;
13              epin = encrypt(pin);
            }else {
14              cout << "User Detected as Supervisor. Enter Supervisor PIN:";
15              cin >> pin;
16              epin = pin;
            }
17          x6 += 25;
        }
18      x6 += 10;
    }
    else {
19      if(x4 > 20) {
20          if(x5 > 15) {
21              x6 += 65;
22              cout << "User Detected as Personal. Enter Personal PIN:";
23              cin >> pin;
24              epin = encrypt(pin);
            }
            else {
25              x6 += 15;
26              cout << "User Detected as Tester. Enter Tester PIN:";
27              cin >> pin;
28              epin = encrypt(pin);
            }
29          x6 -= 4;
        }
        else {
30          cout << "User with Insufficient Permission -Program Will Terminate!.";
31          exit(1);
        }
    }
32  if (x6 > 50) {
33      cout << "User Permission does not Allow Remote Connection.";
    }
    else {
34      cout << "Permissions OK. Encrypted PIN will be Sent Over the Network.";
35      Open_Net_Connection(epin);
    }
36  cout << "Thanks for Using Secure Software! ";
}
```

Fig. 4. Sample Code for Sending Encrypted Password Over a Network

node, through the execution of node 21 would never be considered, since this path is already flagged as critical because of the security chaining approach undesired last definition node(node 24).

Effectively, the security chaining approach would result in the generation of only those event sequences that are needed to be executed for security vulnerabilities to be detected. Each one of those event sequences would construct a search tree. In contrast to chaining approach which terminates the search upon one success, the security chaining approach would attempt each one of those trees, since each one represents a potential vulnerability. If the approach cannot

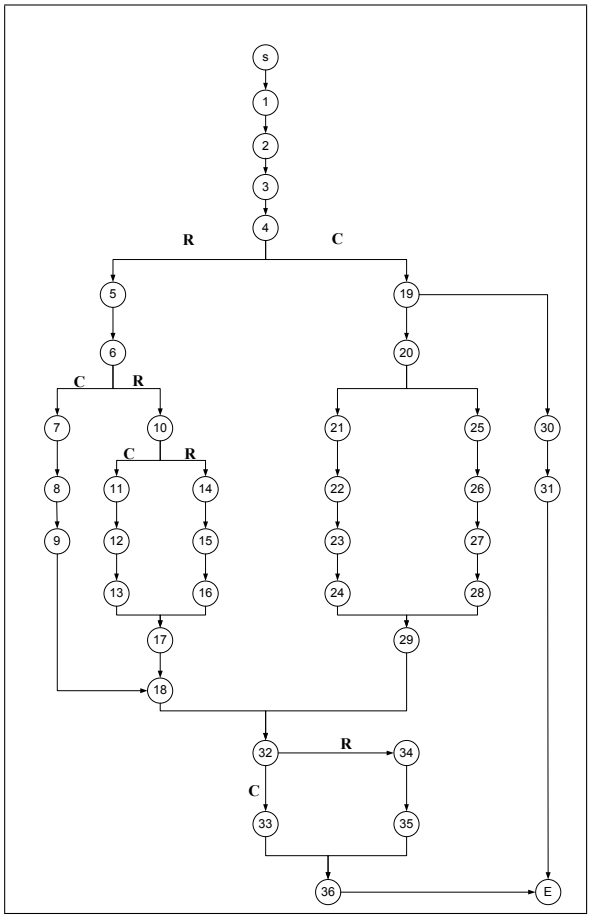

Fig. 5. Corresponding Control Flow Graph of Code in Figure 4

find a solution to traverse a tree, then this tree is considered impossible, and hence no vulnerability is detected or reported, eliminating all false positives.

For instance, considering the code in figure 4, the approach would initially create a single event sequence, shown in figure 6, which would evolve to a search tree should problem nodes are encountered.

5 Framework Architecture

A high-level view of our system architecture is shown in Figure 7. The System contains 7 main components:

1. GCC Extension for Code Instrumentation: This extension is able to instrument any code at a variety of program points in a source code. This tool

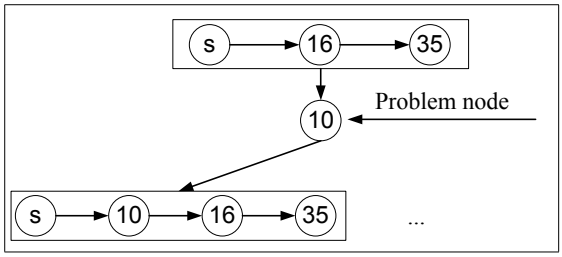

Fig. 6. Search Tree Generated by the Security Chaining Approach

injects the additional code which monitors the dynamic behavior of the program.

2. Team Edit Automata: This component describes the security property as selected by the security analyst. The Team Edit Automata model combines the powerful enforcing capabilities of Edit Automata into the component-interactive architectural model defined by Team Automata. The resulting model is a team composed of one or multiple components of edit automata. A team edit automaton connects its component automata through action signatures - definitions that designate the source and destination of actions.
3. GCC Extension for XML-Dump: GCC Extension for GIMPLE XML dumping [3]. The purpose of the tools is to dump the GIMPLE tree into XML format, which is used in the next stage.
4. XML Parser: Parses the XML representation of the GIMPLE tree and generates the CFG for control and data flow analysis.
5. Security Chaining Data Flow Analyzer: This component will perform the data flow analysis, and annotate the CFG by classifying the branch as Critical or Required.
6. Security Chaining Execution Manager: This is the engine of the security chaining approach. It will use the annotated CFG as the guide to run the instrumented executable and to monitor the execution to generate the test data.
7. Report Analyzer: Collects the data generated by the Security Chaining Execution Manager and generates final test reports indicating the presence of any vulnerabilities, their locations and conditions under which these vulnerabilities would be realized.

To sum up, our work is not directly performed on source code, rather on an intermediate representation of it; specifically, a language independent GCC GIMPLE tree. We also utilize some extensions of GCC. First, the GCC extension for XML-Dump is used to generate the XML representation of the GIMPLE tree, then feeds it to the XML Parser to generate the CFG. Given the CFG, the security chaining Data Flow Analyzer is used to perform data flow analysis and to classify the branches of the CFG as *critical*, or *required*. Then, *GCC Extension for Code Instrumentation* we inject the monitoring code and produce

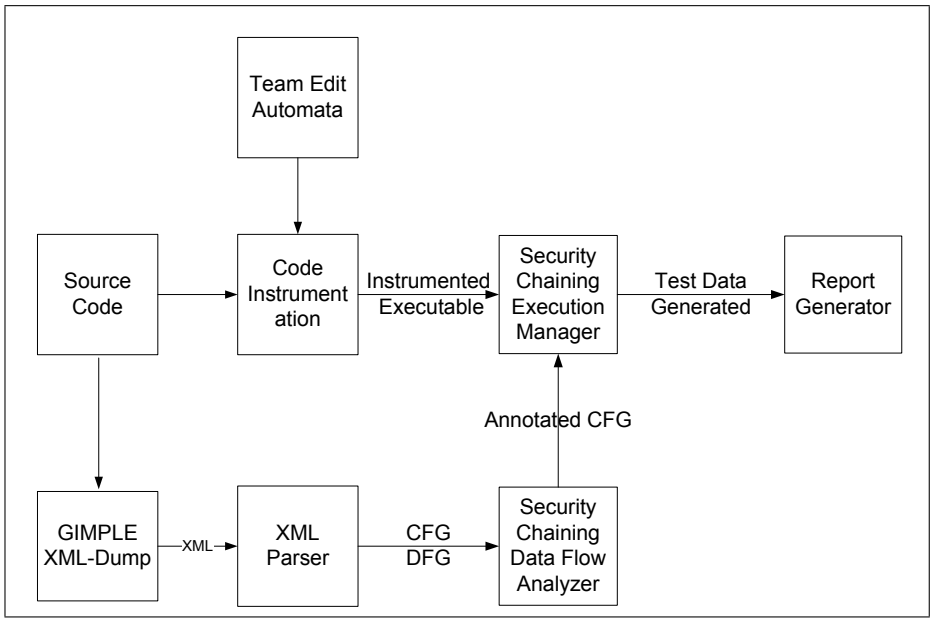

Fig. 7. A High-level View of the System Architecture

the exactable files. Utilizing the annotated GFC, the execution manager runs the produced executable(s) and monitors their execution to generate test data, which is sent to the Report Generator, which produces final reports detailing all the detected vulnerabilities.

6 Conclusion

In this paper, we presented elements of a solution towards automation of software security testing and the generation of test data for the purpose of detecting security vulnerabilities in software. The proposed solution enables the detection of a range of vulnerabilities, both high-level and low-level. The solution utilizes both control flow and data dependency to achieve the needed goals.

While we understand that constructing a single approach that is capable of detecting all possible software security vulnerabilities is not possible, we do believe that multiple components of a single tool may be able to move us forward toward this target. Previously, we introduced TEA [14], which is capable of handling a wide range of safety and security properties. We have incorporated the the Security Chaining Approach into the Security Testing component of our Trusted Free & Open-Source Software suite. This addition not only allows our tool to handle a larger set of security vulnerabilities but to also detect specific set of vulnerabilities that are not, and in most cases could not, be handled by even the best currently available commercial tools for software security testing.

References

1. D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems J., 22(3), pp. 229-245*, 1982.
2. R. Boyer, B. Elspas, and K. Levitt. Select - a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices, 10(6), pp. 234-245*, 1975.
3. F. Brandner, D. Ebner, and A. Krall. Compiler generation from structural architecture descriptions. *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, September 2007.
4. Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. March 2005.
5. M. Chakraborty and U. Chakraborty. An analysis of linear ranking and binary tournament selection in genetic algorithms. In *International Conference on Information, Communications and Signal Processing*. ICICS, September 1997.
6. Cigital and National Science Foundation. Genetic algorithms for software test data generation.
7. L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering, 2(3), pp. 215-222.*, 1976.
8. R. Ferguson and B. Korel. The chaining approach for software test data generation. In *ACM Transaction on Software Engineering and Methodology*, volume 5, pages 63–86. ACM, January 1996.
9. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. June 2005.
10. B. Korel. Automated software test data generation. *IEEE Transactions on Software Enfineering, Vol. 16 No 8.*, August 1990.
11. B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE05)*, 2005.
12. Bogdan Korel. A dynamic approach of test data generation. 1990.
13. Fred B. Schneider. Enforceable security policies. *ACM Transaction of Information System Security*, 2000.
14. Zhenrong Yang, Aiman Hanna, and Mourad Debbabi. Team edit automata for testing security property. *Third International Symposium on Information Assurance and Security*, 2007.