International Journal of Security and Its Applications
Vol.8, No.2 (2014), pp.125-138
http://dx.doi.org/10.14257/ijsia.2014.8.2.13

Domain Specific Language for Detecting Intrusion Signatures with Genetic Search

Kanin Chotvorrarak and Yachai Limpiyakorn,

Department of Computer Engineering, Chulalongkorn University,
Bangkok 10330, Thailand
Kanin.C@student.chula.ac.th, Yachai.L@chula.ac.th

Abstract

This paper presents a network intrusion detection system, which is categorized as a type of signature-based detection method. A domain specific language, called isDSL, is developed as a means of declaring intrusion signatures. The isDSL rule syntax is defined based on the structure of TCP/ IP stack, and the sign of attack is prescribed as a combination of properties and values that could span across the packets or TCP/IP layers. The prototype of intrusion detection system has been implemented. It consists of three major components: 1) isDSL parser, 2) Traffic monitor, and 3) Network intrusion detector. The isDSL parser supports the parsing of the intrusion conditions prescribed in a rule script into a set of rule structures used for matching with the network intrusion packets. Traffic monitor is the engine responsible for capturing the network packets and storing them in the buffer for further inspection. The Network intrusion detector applies the genetic algorithm for searching malicious states on network traffics. Preliminary experiments were conducted to study the performance of the presented approach. The findings reported that the application of genetic algorithm for searching the signs of security breaches against declarative rules would be efficient and promising.

Keywords: *domain specific language, intrusion detection system, genetic algorithm, network security*

1. Introduction

Intrusion detection is a type of security management system for computers and networks. In computer security, it is one of the important technologies for identifying possible security breaches, which include both intrusions (attacks from outside the organization) and misuse (attacks from within the organization). An intrusion detection system, IDS, is a device or software application that gathers and analyzes information from various areas within a computer or a network to assess the signs of intrusions defined as attempts to compromise the confidentiality, integrity and availability, or to bypass the security mechanisms of the network system. Example intrusion detection systems include Snort [1], STAT IDS [2], and RealSecure [3], *etc*.

Network intrusion detection has been an active field of research for a long time. In 1980, Anderson introduced the concept of intrusion detection, and defined a threat from the unauthorized access [4]. Denning published "An Intrusion Detection Model" in 1987, presenting intrusion detection methods which include profiles, anomalies and rules [5]. During 1990s, the Internet growth had led to the major concern of network security. To maintain the security of computer networks and the integrity of the user data, some aspects or conditions must be verified as an intrusion detection task. These conditions, specified in terms of properties of the network traffic, will form a specification of a desired or a malicious state of the network.

ISSN: 1738-9976 IJSIA

A Domain Specific Language, DSL, is a specification language dedicated to a particular problem domain. It is expressive and enabling an easy description of intrusion signatures that spread across several network packets. This paper thus presents a signature-based intrusion detection system using genetic algorithm, GA, for searching the network intrusions against the rules defined in a DSL script written with the domain specific language created in this work.

The remainder of this paper is organized as follows. Section 2 presents a brief of intrusion detection systems, domain specific languages, and genetic algorithm. Section 3 describes the design of domain specific language, isDSL, created for declaring intrusion signatures. Section 4 describes the implementation of the prototype system. Preliminary experiments were conducted to study the efficiency of the presented approach. The findings are summarized in section 5. Section 6 concludes the work in this research and future direction.

2. Background

2.1. Intrusion Detection System (IDS)

Intrusion detection systems can be characterized based on Location and Detection methods. The former can be further classified into two types: 1) Network intrusion detection, and 2) Host intrusion detection. Whereas the latter can be majorly classified into two types: 1) signature-based detection, and 2) anomaly-based detection [6]. With signature-based detection, intrusions are detected using their signatures, *i.e.*, particular properties of network packets used by the intrusion. On the other hand, in anomaly based detection, the system models the normal behavior of the network using statistical methods and/or data mining approaches. The anomaly detection will monitor network behaviors, and if it is considered anomalous according to the network model, then there is a great probability of an attack. The network intrusion detection system, NIDS, developed in this work is a type of signature-based method.

2.2. Domain Specific Language (DSL)

DSL is the high level language focusing only on must-have ability. It is used to define syntax and semantics for the abstraction of the problem domain, for the following purposes: efficiency, easy to understand, and reduce complexity of the language. The domain expert like security professional in network security domain can understand and maintain DSL scripts without much programming skill and some parts of software can be written by domain experts [7]. Moreover, there are a lot of libraries to support the development of DSL in several programming languages, such as .Net parser generator so called “Irony” in C# language, and “ANTLR” in Java language.

Some intrusion detection systems, like Snort [1] and Bro [8], provide custom languages to describe the intrusion signatures, but they are usually scripting languages, based mostly on pattern matching and regular expressions. Since these languages do not use a declarative approach, thus making them less expressive. In literature, the NeMODe system [9] is an example of network intrusion detection systems that provides a declarative and expressive domain specific language for describing intrusion signatures that could spread across network packets. Simply stating constraints over network packets in the script, the desired intrusions will be recognized by Constraint Programming paradigm used as the backend detection mechanism.

2.3. Genetic Algorithm (GA)

Genetic algorithm [10] is the heuristic search that mimics the process of natural selection. It is generally used to search for optimizing solution navigated by a fitness function. The set of initial population will evolve to the optimization value via the GA operators, crossover and mutation. In this work, GA is used for generating several chromosomes of packet combination based on an individual signature description. And with these populated chromosomes, the target intrusion would be detected, if exists.

3. Design of isDSL

isDSL (intrusion signature Domain Specific Language) is designed on the basis of TCP/ IP layer structure. The following subsections describe the influences of TCP/ IP stack that affects the design of isDSL, the syntax of isDSL, and the verification of the conditions prescribed in a rule.

3.1. Transmission Control Protocol/ Internet Protocol (TCP/IP)

The software that manages the packets of information in a computer network is called Transmission Control Protocol/Internet Protocol, TCP/ IP. This software has become the universal standard for information exchange on the Internet. TCP creates the packets and reassembles them into the original message. IP handles packet addressing and ensures that they are transmitted across multiple networks and computers to the correct destination. A lot of vulnerabilities of TCP/ IP protocol have been reported, and network intruders use these flaws to make various network attacks.

The TCP/IP stack is a complete set of networking protocols. Each layer contains a set of properties as illustrated in Figure 1. The syntax of isDSL is designed based on the layered structure of TCP/ IP stack.

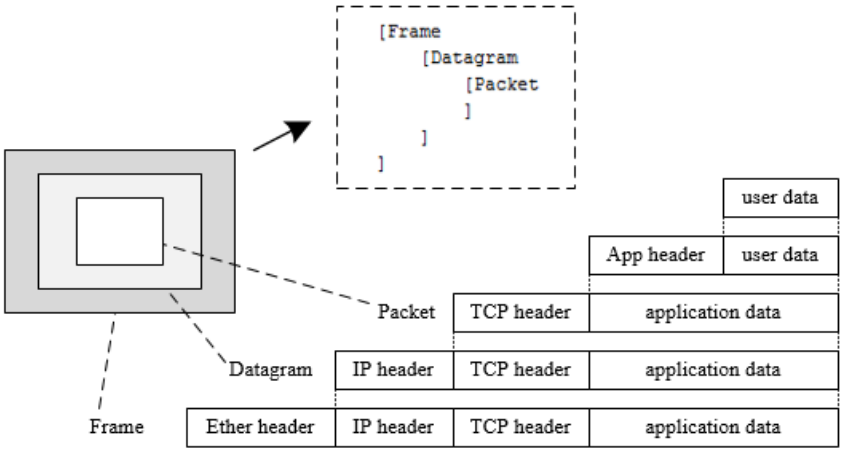

Figure 1. Design of isDSL based on Layered Structure of TCP/IP Stack

3.2. isDSL Rule Syntax

The tokens comprising the isDSL rules are derived from the properties of TCP layers, including Ethernet Frame header, Datagram, TCP, UDP, and DNS header. Figure 2 shows the properties within TCP/ IP packets used for defining the isDSL rule syntax in this work. Ex-

ample properties within each layer include: source and destination MAC address contained in Ethernet Frame header, source and destination IP address contained in IP datagram, flags of syn, ack, psh, fin, etc. contained in TCP packets. The excerpt of isDSL rule syntax is illustrated in Figure 3.

```
<Frame_prop> ::= "srcMac" | "desMac" | "protocolType" | "arpOp"
<Datagram_prop> ::= "srcIP" | "desIP" | "TTL" | "length" | "fragment" | "protocol" | "version" | "id"
<TCP_prop> ::= "srcPort" | "desPort" | <TCP_flags> |  "winSize" | "seqNum" | "ackNum" | "nextSeqNum"
<TCP_flag> :: = "res" | "crw" | "ece" | "urg" | "ack" | "psh" | "rst" | "syn" | "fin"
<UDP_prop> ::= "srcPort" | "desPort" | "lenght" | <DNS_Packet>
<DNS_prop> ::= "dnsID" | "dnsOp"
```

Figure 2. isDSL Tokens Aligned with Properties within TCP/IP Packets

The isDSL script contains the rules representing intrusion signatures. The isDSL rules describe intrusion signatures via the packet's conditions. The logical operators *equal* "=" and *not equal* "! =" are used for the comparisons in the rule conditions. Table 1 summarizes the isDSL condition types defined in this work. Each condition type is associated with a distinct procedure of packet's property investigation.

Table 1. isDSL Condition Types

Type	***Meaning***	***Example***
PIDoPID	Packet ID Operator Packet ID	a == b, a != b
PPoV	Packet property Operator Value	a.port == 80,
PPoPP	Packet property Operator Packet property	a.desPort == b.srcPort
FoV	Function Operator Value	Sequence(b,a,c)

The condition type of PIDoPID describes the comparison of a packet and another packet referenced by their identifiers. For example, the condition a == b is satisfied, if all properties of packet "a" are the same as all properties of packet "b". While.a != b is satisfied, if there exists at least one property of packet "a" differs that of packet "b".

The condition type of PPoV describes the comparison of a packet property and its value. For example, the condition a.port == 80 is satisfied, if the property "port" of packet "a" equals the value of 80.

The condition type of PPoPP describes the comparison of a packet's property and another packet's property. For example, the condition a.desPort == b.srcPort is satisfied, if the destination port of packet "a" equals the source port of packet "b".

The condition type of FoV describes the assertion of the control function, namely Sequence() and Count(). For example, the assertion of Sequence is true if the arrival of packet "c" follows packet "a", which in turn, follows packet "b".

Some intrusion signatures declared with isDSL are provided. The intrusion signature of ARP (Address Resolution Protocol) spoofing is shown in Figure 4. The ARP spoofing is the attack that the intruder sends a fake ARP message onto networks in order to deceive the sender to send the message to the intruder.


```
// Parameter
paramList.Rule = "(" + idList + ")";
idList.Rule = MakeStarRule(idList, comma, identifier);

// Frame Rule
frameConStmt.Rule = f_MacCon | f_ProtocolTypeCon | f_ArpOpCon;
f_Mac.Rule = srcMac | desMac;
f_MacCon.Rule = identifier + dot + f_Mac + biOp + identifier + dot + f_Mac |
          identifier + dot + f_Mac + biOp + value;
f_ProtocolTypeCon.Rule = identifier + dot + protocolType + biOp + identifier + dot + protocolType |
                identifier + dot + protocolType + biOp + value;
f_ArpOpCon.Rule = identifier + dot + arpOp + biOp + identifier + dot + arpOp |
             identifier + dot + arpOp + biOp + value;

// Datagram Rule
datagramConStmt.Rule = d_IpCon | d_TtlCon | d_FragmentCon | d_ProtocolCon | d_LenghtCon | d_VersionCon | d_IdCon;
d_Ip.Rule = srcIP | desIP;
ipValue.Rule = number + dot + number + dot + number + dot + number;
d_IpCon.Rule = identifier + dot + d_Ip + biOp + identifier + dot + d_Ip |
          identifier + dot + d_Ip + biOp + ipValue;
d_TtlCon.Rule = identifier + dot + ttl + biOp + identifier + dot + ttl |
          identifier + dot + ttl + biOp + value;
d_FragmentCon.Rule = identifier + dot + fragment + biOp + identifier + dot + fragment |
             identifier + dot + fragment + biOp + value;
d_ProtocolCon.Rule = identifier + dot + protocol + biOp + identifier + dot + protocol |
             identifier + dot + protocol + biOp + value;
d_LenghtCon.Rule = identifier + dot + lenght + biOp + identifier + dot + lenght |
             identifier + dot + lenght + biOp + value;
d_VersionCon.Rule = identifier + dot + version + biOp + identifier + dot + version |
             identifier + dot + version + biOp + value;
d_IdCon.Rule = identifier + dot + id + biOp + identifier + dot + id |
          identifier + dot + id + biOp + value;

// TCP Packet Rule
tcpPacketConStmt.Rule = t_PortCon | t_FlagCon | t_AckNumberCon | t_SeqNumberCon | t_NexSeqNumberCon | t_WinSizeCon;
t_Port.Rule = srcPort | desPort;
t_Flag.Rule = fin | syn | rst | psh | ack | ece | cwr | urg | res;
t_PortCon.Rule = identifier + dot + t_Port + biOp + identifier + dot + t_Port |
          identifier + dot + t_Port + biOp + number;
t_FlagCon.Rule = identifier + dot + t_Flag + biOp + identifier + dot + t_Flag |
          identifier + dot + t_Flag + biOp + number;
t_AckNumberCon.Rule = identifier + dot + ackNumber + biOp + identifier + dot + ackNumber |
          identifier + dot + ackNumber + biOp + number;
t_SeqNumberCon.Rule = identifier + dot + seqNumber + biOp + identifier + dot + seqNumber |
          identifier + dot + seqNumber + biOp + number;
t_NexSeqNumberCon.Rule = identifier + dot + nexSeqNumber + biOp + identifier + dot + nexSeqNumber |
          identifier + dot + nexSeqNumber + biOp + number;
t_WinSizeCon.Rule = identifier + dot + winSize + biOp + identifier + dot + winSize |
          identifier + dot + winSize + biOp + number;

// UDP Packet Rule
udpPacketConStmt.Rule = u_PortCon | u_LenghtCon;
u_Port.Rule = srcPort | desPort;
u_PortCon.Rule = identifier + dot + u_Port + biOp + identifier + dot + u_Port |
          identifier + dot + u_Port + biOp + number;
u_LenghtCon.Rule = identifier + dot + lenght + biOp + identifier + dot + lenght |
          identifier + dot + lenght + biOp + value;

// DNS Packet Rule
dnsPacketConStmt.Rule = d_DnsIdCon | d_DnsOpCon;
d_DnsIdCon.Rule = identifier + dot + dnsId + biOp + identifier + dot + dnsId |
          identifier + dot + dnsId + biOp + number;
d_DnsOpCon.Rule = identifier + dot + dnsOp + biOp + identifier + dot + dnsOp |
          identifier + dot + dnsOp + biOp + value;
```

Figure 3. isDSL Rule Syntax

```
[Rule "ARP spoofing" (a,b,c)
    [Frame
        a.desMac = "FF:FF:FF:FF:FF:FF",
        b.srcMac != c.SrcMac
        [Datagram
            a.desIP = b.srcIP, a.desIP = c.srcIP,
        ]]]
```

Figure 4. ARP Spoofing Rule

Figure 5 describes the rule representing the signature of DNS (Domain Name System) spoofing where a DNS server accepts and uses incorrect information from a host that has no authority giving that information. In particular, the forged data is placed in the cache of the named servers causing users to be directed to malicious Internet sites or e-mail being routed to non-authorized mail servers.

The use of control functions is allowed with isDSL programming. Currently, two control functions are provided, namely Sequence () and Count (). The function Sequence (packetA, packetB) is used for verifying that the rule is satisfied only when packetA arrives before packetB. The function Count (packet, number of occurrences) is used for verifying that the suspect packet satisfies the rule if its reoccurrences reach the number specified. Figure 6 demonstrates the use of function Count () residing the rule representing the signature of SYN (Synchronization) flooding attack. A SYN flood is a method that the user of a hostile client program can use to conduct a denial-of-service (DoS) attack on a computer server. The hostile client sends a succession of SYN requests, using fake IP addresses, to a target system in an attempt to consume enough server resources to make the system unresponsive to legitimate traffic.

The use of control functions would help tuning out false positives due to the expressiveness of describing more conditions that are required for inspection.

```
[Rule "DNS spoofing" (a,b,c)
   [Frame
       [Datagram
          c.desIP = a.srcIP,
          b.desIP = a.srcIP
          [UDP_Packet
             a.desPort = 53, b.srcPort = 53, c.srcPort = 53,
             c.desPort = a.srcPort, b.desPort = a.srcPort
             [DNS_Packet
                a.dnsID = b.dnsID, b.dnsID = c.dnsID
    ]]]]
    b != c
]
```

Figure 5. DNS Spoofing Rule

```
[Rule  "Syn flooding" (a)
    [TCP_Packet
         a.syn = 1
    ]
    Count(a,100)
]
```

Figure 6. SYN Flooding Rule

3.3. isDSL Rule Verification

Once a rule script has been parsed for syntactic analysis, the conditions prescribed in the rule are verified for conflict and duplication. Dictionary, which is a hash table, is used to support the inspection, and to store the validated condition list comprising a rule.

Each condition contains parameters, properties, and operator. The composition of these condition arguments is the input for generating a value of key returned from the hash function. Prior to adding a condition into the Dictionary hash table, its key value is checked for collision to prevent duplicated conditions prescribed in a rule.

In order to examine the condition conflict contained in a rule, the operator will be inversed when generating a key value returned from the hashing function. The key collision indicates the conflict of conditions prescribed within a rule. Figure 7 illustrates the Dictionary hash used for isDSL rule verification in this work.

```
// Key => packet properties, parameter, operator
// Value => Condition
Dictionary<string, Condition> ConditionList = new Dictionary<string, Condition>();
if (!ConditionList.ContainsKey(prop + param + op)){
    ConditionList.Add(prop + param + op, con);
}
```

Figure 7. Dictionary Hash to support isDSL Rule Verification

4. Implementation

The prototype of signature intrusion detection system consists of three main components as shown in Figure 8: 1) isDSL parser, 2) Traffic monitoring engine, and 3) Network intrusion detection engine, as described in the following subsections.

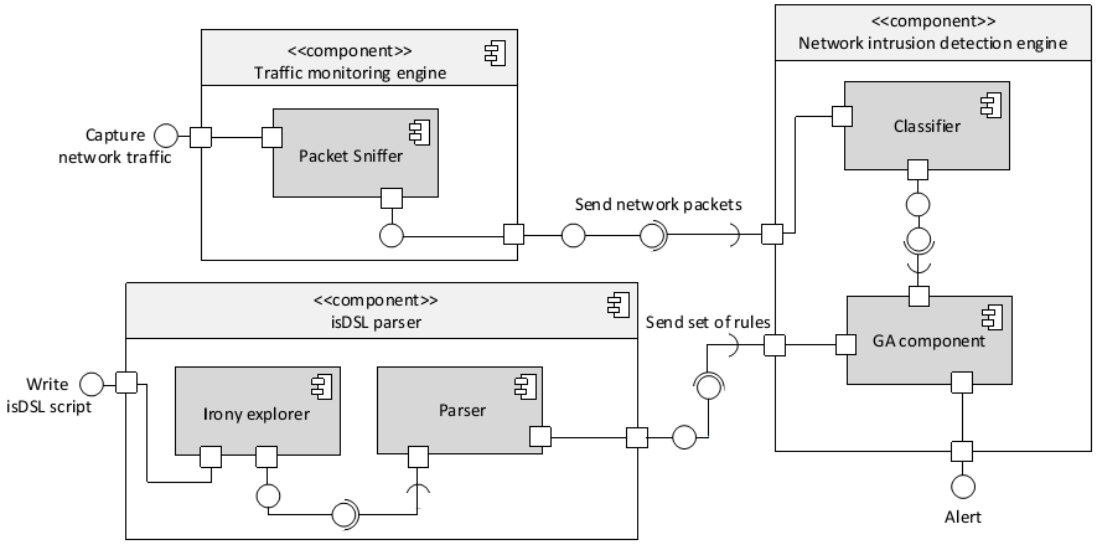

Figure 8. Components of Implemented Prototype

4.1. isDSL Parser

The Irony.Net [11] is used as the parser generator to construct the isDSL parser in this research. Irony is an open source software library with a complete set of libraries

and tool for language implementation. Example of isDSL parser interface for creating an isDSL script is shown in Figure 9.

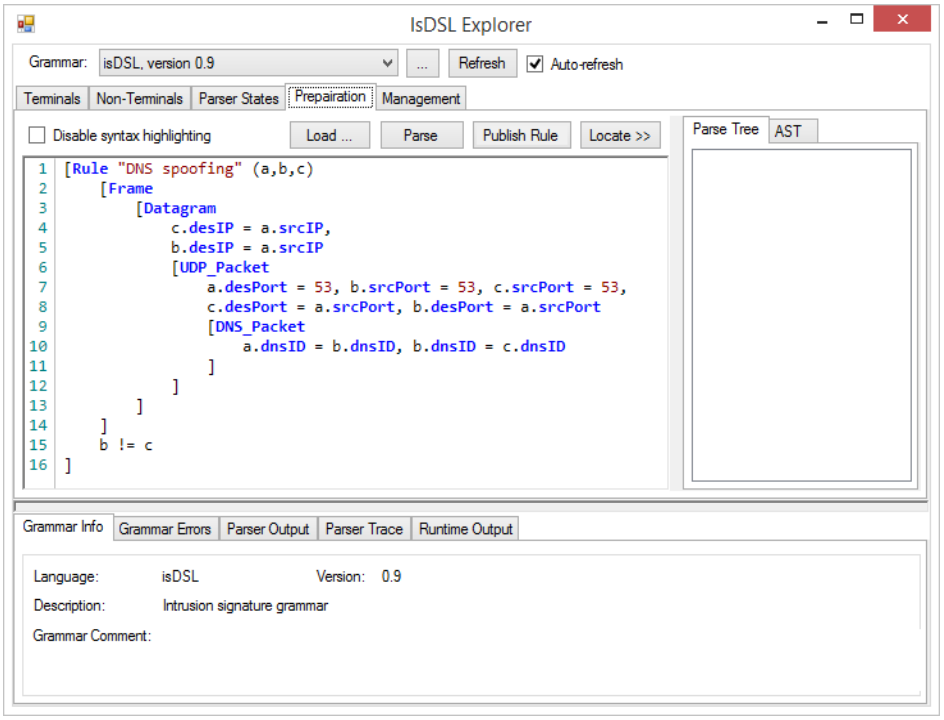

Figure 9. Example Interface of isDSL Parser (isDSL IDE)

4.2. Traffic Monitoring Engine

The Traffic monitoring engine is responsible for sniffing the network traffic in order to capture the packets and output them to the Network intrusion detection engine for further processing. The Pcap.Net library is used for implementing this module. Figure 10 illustrates the packets initially captured in Hex stream format. They will then be decoded to the readable structure with Pcap.Net library as shown in Figure 11.

```
0000  f0 de f1 d1 ce b6 00 1b  d4 dd 81 e0 08 00 45 00   ........ ......E.
0010  02 c9 83 03 40 00 4b 06  18 7c 45 ab f8 10 9d 3c   ....@.K. .|E....<
0020  b6 b7 01 bb 3c bd 6e 65  db 5f 56 58 a4 0c 50 18   ....<.ne ._VX..P.
0030  01 1b 77 0d 00 00 17 03  01 02 9c a1 df 46 29 4d   ..w..... .....F)M
0040  86 b2 df c2 83 ed 58 c6  0d 54 d4 f6 3a ec 4c c2   ......X. .T..:.L.
0050  f0 70 3f 05 e5 6e a4 7d  61 da 09 13 58 c4 05 9f   .p?..n.} a...X...
0060  e3 3c 9c d7 ef a1 8b 46  be a3 ac 27 18 86 b9 b9   .<.....F ...'....
0070  53 2d 9c 1e 18 19 1f 50  fa 30 e3 9f 1b a7 d7 63   S-.....P .0.....c
0080  18 7e ec 5b f5 67 1d 41  a3 c4 a1 04 99 02 ec b4   .~.[.g.A ........
0090  cd 80 d8 46 2c 2a 93 8e  a3 34 e3 5b eb da cc 18   ...F,*.. .4.[....
00a0  0e c3 56 56 69 26 7e c0  cf 21 65 bb 64 b6 68 34   ..VVi&~. .!e.d.h4
00b0  8a e1 54 4e 17 42 5c 18  12 a0 87 24 41 34 7d 9b   ..TN.B\. ...$A4}.
00c0  fe 78 db a0 b9 a9 25 14  65 d1 b9 25 09 8b f8 62   .x....%. e..%...b
```

Figure 10. Network Packets Captured in Hex Stream Format

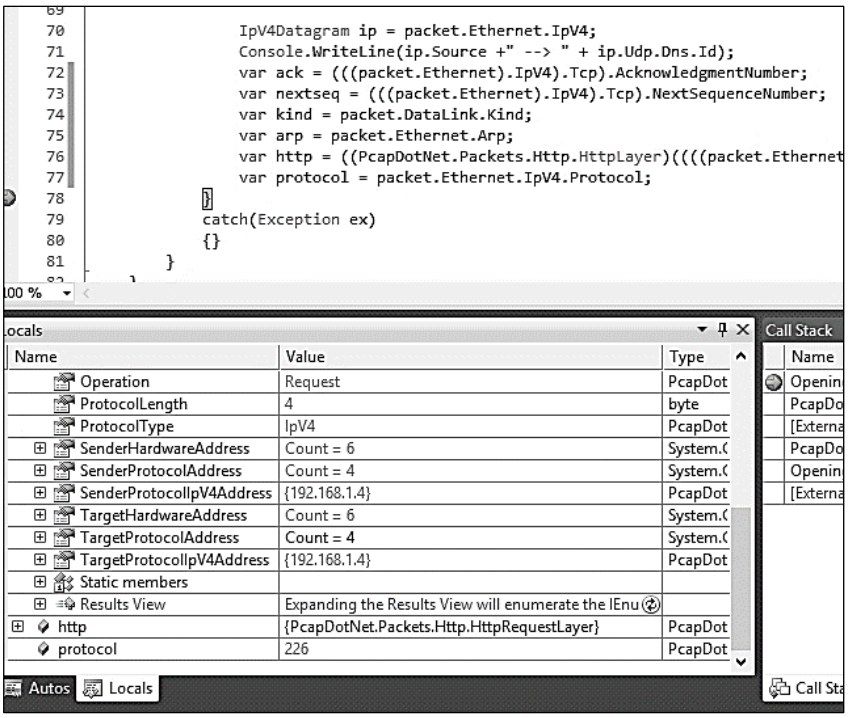

Figure 11. Packet Transformation to Readable Format with Pcap.Net

4.3. Network Intrusion Detection Engine

The backend module, Network intrusion detection engine, is responsible for detecting the signs of intrusions, and if found, it will alert and log the incident. Referring to Figure 8, the Network intrusion detection engine consists of two major components: 1) Classifier, and 2) GA component.

Classifier is the component in charge of filtering those which are not TCP/IP packets. Only TCP/IP packets will then be stored in the buffer, which is the large working space capable of supporting concurrent multiple rule matching. A number of packets residing the buffer are framed for malicious inspection against an individual rule. For example, in this work, a window of 100 packets contained in the buffer is used for each round of inspection. Next, since a particular rule involves examining one or more TCP/IP layers, those packets relevant to the layers specified in the rule will be selected for generating the initial chromosome population during the GA process. The list of indexes of these relevant packets will be stored in the attribute "packets" within the GA class as shown in Figure 12.

The GA component is responsible for searching the network intrusion matched with the conditions prescribed in a rule. The GA process starts with the generation of the initial chromosome population. Each GA chromosome is encoded with the list of packet indexes corresponding to the number of the rule's arguments. Considering the rule of "Attack type1" with three arguments: a, b, c, indicating that three packets will be simultaneously examined, the chromosome is thus encoded with the list of indexing numbers of three packets randomly selected from the pool of relevant packets. Example of a packet combination as one of the initial chromosome population is shown in Figure 13. Hashing is used to provide direct access to the details of properties and values contained in the packets being indexed.

Next, each chromosome will be evaluated for its fitness. The procedure of fitness function evaluation is excerpted and depicted in Figure 14. The target score signaling the malicious state is pre-defined for each rule based on the number of matching conditions. The absolute matching indicates a great probability of the existence of intrusion. Figure 15 illustrates the GA process as part of the process flow of the Network intrusion detection.

It is observed that the maximum number of iterations of genetic algorithm is subject to the complexity of the rule, *i.e.*, the number of constituent conditions and parameters.

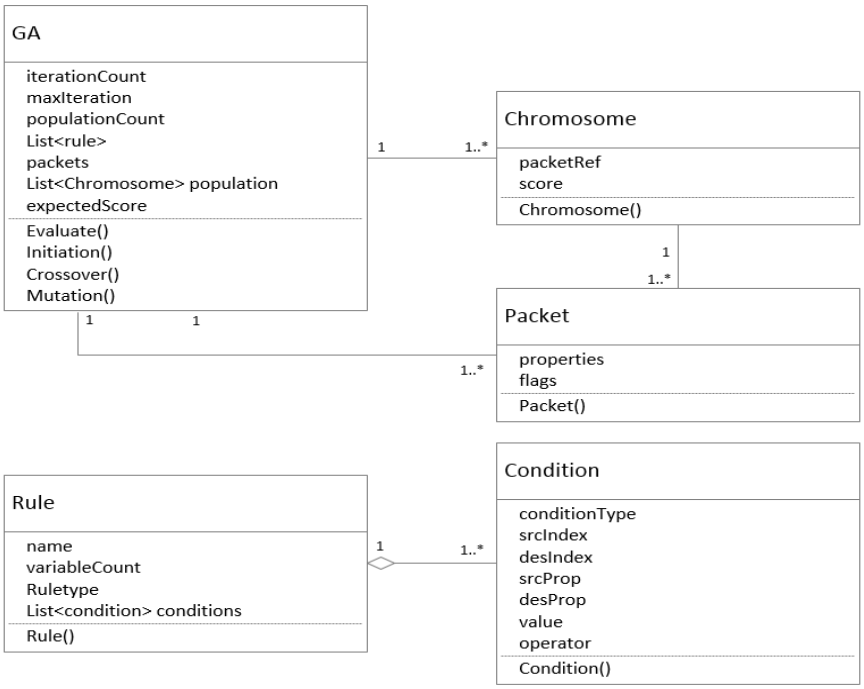

Figure 12. Class Diagram of Network Intrusion Detection Engine

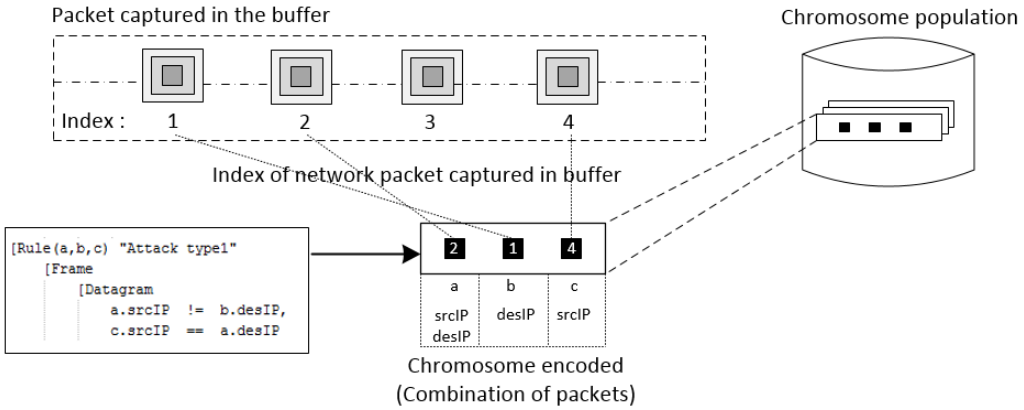

Figure 13. Mapping between Rule and Chromosome Encoding

```
if (rule.type == RuleType.accross){
    Parallel.ForEach(rule.conditions, c =>{
        switch (c.type){
            case ConditionType.PIDoPID:{
                    foreach (Chromosome p in population){
                        if (c.op == Operator.Equal && p.packetRef[c.srcIndex] == p.packetRef[c.desIndex]){
                            p.score++;
                        }
                        else if (c.op == Operator.NotEqual && p.packetRef[c.srcIndex] != p.packetRef[c.desIndex]){
                            p.score++;
                        }}} break;
```

Figure 14. Excerpt of Fitness Function Evaluation

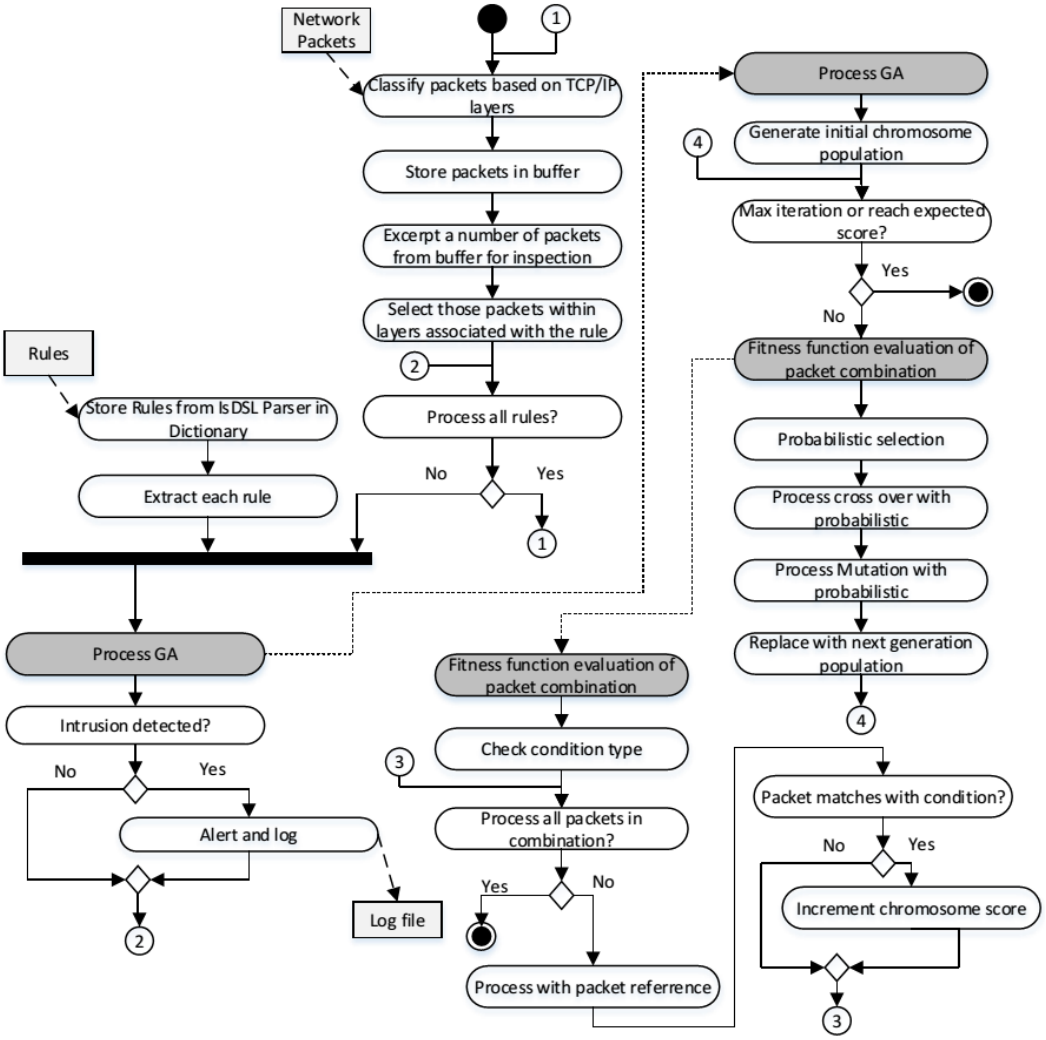

Figure 15. Process Flow of Network Intrusion Detection

5. Preliminary Experiments

Preliminary experiments were conducted to study the efficiency of the presented approach. Two network intrusions: ARP spoofing, and DNS spoofing were selected as the tests. The results were reported on the execution with the hardware specification: Intel Core i5-2410M Processor (2.3GHz, 3MB L3, 1333MHz FSB) and 4 GB of memory.

The GA parameters were defined as follows:

- 19 chromosomes were maintained for each generation.
- Replacement rate for Crossover = 0.3 (three pairs of chromosomes were probabilistically selected for Crossover)
- Single-point Crossover was applied using the Crossover mask '100' (after the first packet).
- The experimental scenarios were varied due to the factors of Mutation rate and the number of Mutation positions, applied after the Crossover operation.

The average times in seconds from 100 runs of the detection of ARP spoofing and DNS spoofing using various combinations of Mutation rates and the number of Mutation positions, are reported in Table 2, and Table 3, respectively.

It is observed that the less Mutation rate (0.2) yields the best performance for both cases. For a particular Mutation rate, the choice of 3 Mutation positions results in the worst performance compared to the alternatives of 1, and 2 Mutation positions. Both the ARP spoofing rule (Figure 4) and the DNS spoofing rule (Figure 5) contain the same number of rule's arguments, *i.e.*, three arguments. However, compared to the ARP spoofing rule, the DNS spoofing rule is more complicated as it contains more conditions resulting in the longer chromosome. This may affect the better performance with the increasing number of Mutation positions, (*i.e.*, 2 Mutation positions) as reported in Table 3.

Table 2. Average Time of ARP Spoofing Detection (in Seconds)

Mutation rate	***Number of Mutation Positions***		
	1	2	3
0.1	0.1181	0.2166	**0.2724**
0.2	**0.0959**	0.1667	0.3496
0.3	0.1115	**0.1606**	0.3698
0.4	0.1220	0.2007	0.3057
0.5	0.1234	0.2216	0.3405
0.6	0.1295	0.1952	0.3399

Table 3. Average Time of DNS Spoofing Detection (in Seconds)

Mutation rate	***Number of Mutation Positions***		
	1	2	3
0.1	0.4656	0.4341	0.4985
0.2	**0.4505**	**0.3557**	0.5367
0.3	0.4607	0.4197	**0.4758**
0.4	0.4692	0.4251	0.5230
0.5	0.4995	0.4206	0.5063
0.6	0.5388	0.4225	0.5399

6. Conclusion

Several protocols manage and control the transmission of information across the networks, including TCP/IP. Despite of its popularity, there exist a number of serious security flaws reported on the design of TCP/IP protocol suite. Since security attacks, such as IP spoofing, SYN flooding, and others, can be described in the state and condition for properties among

packets, declarative languages are considered as an alternative to expressively describe the signs of attacks. This research thus creates a domain specific language, isDSL, for declaring the rules containing the conditions that would indicate the malicious states. The language is capable of describing the relation of properties in a single or across protocol layers as it is designed on the basis of TCP/IP stack. The domain experts, such as network security professional, can easily understand and code the isDSL rules to express intrusion signatures.

Genetic algorithm is used for searching the combination of network packets that would form the state of security breaches described in a particular rule. Unlike other traditional methods, genetic algorithm searches for the solution from several diverse hypotheses in parallel. Therefore, the method could avoid being trapped in the local optimal solution.

The prototype of the signature-based network intrusion detection system is implemented. Preliminary experiments were carried out to study the performance of the presented approach. The results were promising. Future work includes the enhancement of isDSL to be more expressive that would result in less false positives. Additional control functions could be defined to prescribe the conditions comprising a detection rule. More programming elements could be investigated and introduced to improve the efficiency of the prototype system.

References

[1] M. Roesch, "Snort-Lightweight Intrusion Detection for Network", Proceedings of LISA '99L 13th System Administration Conference, Seattle, Washington, USA, **(1999)** November 7-12.
[2] K. Ilgun, A. Kemmerer, Fellow and P. A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach, Software Engineering, IEEE Transactions, vol. 21, **(1995)**.
[3] ISS-RealSecure, http://www.iss.net/.
[4] P. N. Raju, "State-of-the-art intrusion detection: Technology, challenges, and evaluation", Linkoping, **(2005)**.
[5] P. Patel, C. Langin, F. Yu and S. Rahimi, "Network Intrusion Detection Types and Computation", International Journal of Computer Science and Information Security, vol. 10, no. 1, **(2012)**.
[6] V. Jaiganesh, S. Mangayarkarasi and P. Sumathi, "Intrusion Detection Systems: A Survey and Analysis of Classification Techniques", International Journal of Advanced Research in Computer and Communication Engineering, vol. 2, **(2013)**.
[7] M. Fowler, "Domain-Specific Languages", Addison-Wesley Professional, **(2010)**.
[8] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, **(1998)**.
[9] P. Salgueiro, D. Diaz, I. Brito and S. Abreu, "Using Constraints for Intrusion Detection: The NeMODe System", PADL, **(2011)**, pp. 115-129.
[10] J. T. Alander, "An indexed bibliography of genetic algorithms: Years1957-1993. Art of CAD Ltd. Vaasa, Finland, **(1994)**.
[11] Irony - .NET Language Implementation Kit, http://irony.codeplex.com/.

Author

Kanin Chotvorrarak, he received his bachelor degree in Computer Engineering from King Mongkut's Institute of Technology Ladkrabang in 2012. After graduation, he has worked as a software design engineer in test at Microsoft (Thailand) Ltd. Currently, he is pursuing the Master degree in Software Engineering at Chulalongkorn University, Bangkok 10330, Thailand.