Generating a Real-Time Constraint Engine for Network Protocols

Mohamed Sami Rakha[1], Fahim T. Imam[2](✉), and Thomas R. Dean[2]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada
m.rakha@queensu.ca

[2] Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada
{fahim.imam,tom.dean}@queensu.ca

Abstract. In this paper, we present a practical approach to generate the constraint engine for an effective constraint-based intrusion detection system (IDS). The IDS framework was designed for safety-sensitive networks that involve limited-access closed networks such as the networks for command and control systems or Air Traffic Control (ATC) systems. The constraint engine generated by the framework supports real-time performance while ensuring the intended, normal behaviour of its target networks. We present the IDS framework in terms of its internal DSL representation as well as its transformation mechanisms to generate the constraint engine code. Comparing the autogenerated version against a manually implemented, optimized version of the constraint engine indicates no significant difference in terms of their performance.

Keywords: Intrusion detection · Domain-specific language · Network security · Real-time systems · Source code generation

1 Introduction

As part of a cybersecurity project, we designed an effective framework to generate a robust constraint-based Intrusion Detection System (IDS) [9] for limited-access closed networks. The IDS was designed to support real-time detection of intrusions for safety-sensitive networks while keeping the intended, normal behaviour of the networks intact for its authentic systems and agents. The IDS framework is envisaged to generate efficient executable code for all of its components, including the constraint engine, from a single specification document. The framework, therefore, supports practical usability for its constraint engine, allowing only limited human interventions for its code maintenance and change management. The key strength of the IDS framework is its realization of three levels of abstractions as follows:

1. The high-level description of the network protocols along with their constraints that can be written by the IDS users like Network Engineers;

O. Blazy and C. Y. Yeun (Eds.): WISTP 2018, LNCS 11469, pp. 44–60, 2019.
https://doi.org/10.1007/978-3-030-20074-9_5

2. The mid-level specification of the constraints represented in an internal DSL that can be transformed from the high-level description;
3. The low-level abstraction representing the executable constraint engine code that can be autogenerated from the internal DSL.

In this paper, we present the current state of our IDS framework in terms of its internal DSL representation that corresponds to the mid-level abstraction above. We also present the transformation mechanism of the internal DSL that we have implemented to generate the low-level, executable code for the constraint engine. The DSL allows expressing different protocol-specific constraints in terms of their internal data structures and other computational details. Essentially, the DSL was designed to ensure optimal memory management and real-time performance for the constraint engine code. Extending based on our previous work [9], the DSL presented in this paper allows generalizing its mechanism to accommodate complex constraints that involve multiple valid sequences of packets with arbitrary order and length.

Prior to proposing the original DSL, we have also presented an optimization approach for the constraint engine which was manually implemented in *C* with promising results [9]. The manual version of the constraint engine code was used as a guide for our automatic code generation framework presented in this paper. Evaluating the generated constraint engine on a set of test cases with different constraints validates the correctness of the autogenerated code. Comparing the performance of the autogenerated constraint engine against the manually implemented version shows that there is no significant performance penalty to our autogenerated code.

The focus of our constraint engine is to achieve real-time performance when inspecting the network packets against a set of constraints and successfully detect any intrusion. The subsequent response to a detected intrusion, however, is outside the scope of our current research. We are currently working on implementing the high-level language to specify different constraints which will be used to autogenerate the internal DSL code discussed in this paper. We will present the high-level language along with its transformation mechanism in our next article.

2 Background

The scope of our research is an anomaly-based IDS for limited-access, closed networks such as industrial control networks or command and control systems such as the Air Traffic Control (ATC) systems. A key feature of these systems is that they all involve a limited number of known protocols with restrictive operations. The network traffic of these systems therefore involve much less variability than that of a conventional network. The queries, responses, and commands involved in these networks tend to have regular patterns that can be predefined and specified as a set of logical constraints. These constraints can then be used to characterize the traffic in terms of their normal, predefined patterns. The IDS can detect potential intrusions based on traffic that deviates from the specified constraints.

The IDS Framework. Figure 1 presents the high-level architecture of our IDS framework. The two main components of the framework are the packet parser

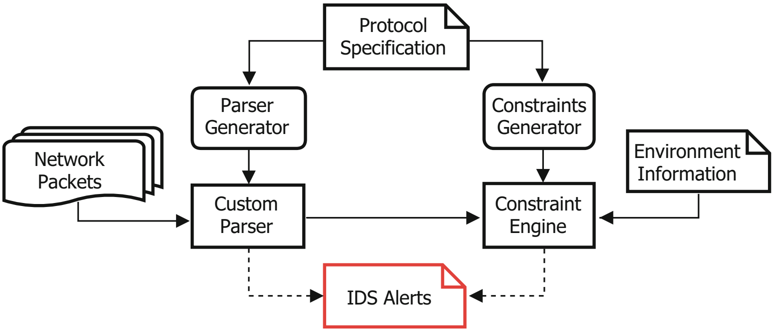

Fig. 1. The IDS framework.

and the constraint engine. Both of these components are automatically generated from the protocol specification document at the top. The specification document is meant to be written by the Network Engineers in a language called SCL [13] using their familiar ASN.1-based notations. SCL allows modular description of multiple protocols in terms of the syntax and semantics of their network traffic packets. The generated parser [6] reads the network packets and converts them into an internal structure for the constraint engine to use. In the process, the parser also validates the internal structure of the packets [6].

The main task of the constraint engine is to validate different constraints among the network packets. The engine is responsible for storing constraint-specific information from the incoming packets that can be used to check the validity of the later packets. The engine also supports the mechanism of automatically deleting the stored information when no longer needed. The interface between the parser and the constraint engine consists of dedicated callback entries for each of the protocol-specific packet type recognized by the parser.

As presented in Fig. 1 our IDS framework only requires minimal human intervention for its change management and code maintenance. Any changes to the network such as the addition of new systems or protocols simply require updating the protocol specification at the top. The underlying framework of the IDS will then take care of generating the new executable code for the parser and the constraint engine based on the updated specification.

The Evaluation Framework. While our IDS framework was designed to support any kind of network protocol, in order to demonstrate our approach we consider the Real Time Publish and Subscribe (RTPS) protocol [16] along with the Internet Group Message Protocol (IGMP) [3] as part of our evaluation framework. The RTPS protocol is a real-time implementation of the Data Distribution Service (DDS) framework [15], where some systems publish data (e.g. radar tracks) while the other systems subscribe to the data (e.g., air traffic control terminal, flight service station). RTPS protocol has been used in many critical domains and real-time applications such as NASA's Launch Control Networks and the Canadian Automated Air Traffic Management System [1]. One of the key concepts in DDS is that of a *topic* which refers to a particular message type and a quality of service. An example might be a radar track, or flight plan information. RTPS is a UDP protocol which uses both multicast and unicast messages. Multicast messages allow a publisher to send a single message that can be received by all subscribers. However, to track which systems are listening to multicast packets means we must also analyze the IGMP protocol, which is used to manage multicast UDP messages. Our IDS framework is being extended to support other UDP protocols such as the NTP [14], TFTP [21], and NFS [4].

The Threat Model. Our threat model is that the attacker has compromised one or more nodes of the network through an alternate channel such as USB and is looking to infect another node or to compromise the network function at some point in the future. In our case, the IDS must treat each of the incoming network packets as a suspicious entity. The contents of an incoming packet as well as its arriving sequence must be evaluated against a set of protocol-specific constraints in order to validate the packet's right to exist in the network traffic as a safe entity. The constraint engine for the anomaly-based IDS, therefore, requires a set of constraints for each of the protocol-specific packet types that collectively specify the normal behaviour of its target network.

There are three types of constraints that we are interested in. The first type involves the constraints specified by the protocol parser to validate the well-formedness of the network packets. The second type involves describing the constraints between the incoming packets and the network environment. For example, radar data and ADS-B [2] data may only originate from a specific set of systems. Finally, the third type of constraints involves describing the constraints among multiple packets along with their valid sequences. Three examples of constraints for the RTPS protocol are: RTPS packets have the correct format and structure, application data packets originate from the correct source addresses, and that an intruder has not introduced a new topic in order to suborn the DDS framework to provide communication between malware components.

3 Generating Constraint Engine Code

Each of the packet types within a network stream must have a set of associated constraints. A packet type refers to the protocol-specific type of the packet such as the `RTPS DATA` message packet. In our approach, the constraints are considered to be independent from each other. Each of the multipacket constraints must specify a set of *Packet Types*, $P = \{P_o, P_1, .., P_n, P_t, P_f\}$ that must exist in a particular order; where, P_o is the type of the `Initial Packet` of the sequence, $P_1, .., P_n$ are the types of the `Intermediate Packets` leading up to the `Target Packet` type, P_t, and the `Final Packet` type P_f after the P_t marks the closure of the packet sequence relevant to the constraint.

Each instance of the network constraints corresponds to a tree data structure, which is referred to as the *constraint tree* in this paper. Such a tree gradually collects data for its nodes from the stream of network packets. A constraint tree can be expressed using the prefix notation: `AND(OP(a, b), OP(c, d))`; where,

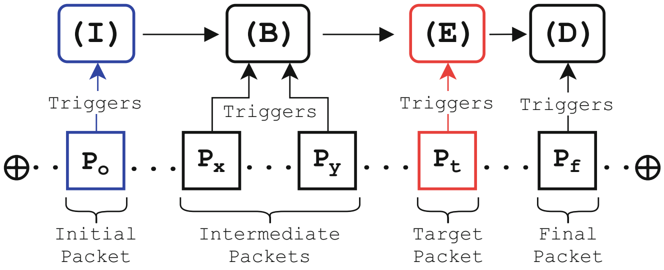

Fig. 2. Sequence of packets with constraint tree phases.

`AND` refers to the logical conjunction, `OP` is a logical operator such as equality (EQ) or non-equality (NEQ), and `a`, `b`, `c`, and `d` represent the leaf node data from different packets that must be evaluated. Hasan et al. [9] proposes a life-cycle model for the constraint trees with four consecutive phases: *Instantiate* (I), *Bind* (B), *Evaluate* (E) and *Destroy* (D). It is the life-cycle model of the constraint trees that dictates the overall memory management and the optimization of the constrain engine in order to achieve the real-time performance of our IDS.

3.1 The Constraint Tree Life-Cycle Model

Based on its corresponding constraint specification, each phase of the constraint tree is triggered by the arrival of a particular packet type within a stream of network packets. Figure 2 illustrates a generic scenario of a constraint that involves different packet types with a particular arriving sequence. In between the initial packet, P_o and the final packet, P_f, there could be many intermediate packets of different types; however, within those packets, the constraint is specific about the existence of three particular types of packets, P_x, P_y, and the target packet P_t, that must arrive in the specified order. The constraint tree phases along with their internal memory management processes are described below.

Instantiate is the first phase of a constraint tree instance which is triggered by the arrival of the initial packet type, P_o from a predefined sequence. In this phase, a memory space is allocated for the constraint tree and a reference to the tree is cached in a hash table that can be used by the later phases. The common values between the initial packet and the consecutive packet(s) are used as the keys for the hash table. Any data needed from the initial packet to evaluate the constraint are stored in the tree instance.

Bind is the second phase of a multi-packet constraint that has more than two packets. This phase involves the following tasks: validate the sequence of the arriving packets and fill out the empty leaves in the constraint tree. While our original approach [9] only allowed a single bind phase, the approach currently allows multiple bind phases to be defined for a constraint. Figure 2 illustrates that scenario by multiple intermediate packet types between the initial and the target packet types. In the first bind phase, the constraint engine retrieves the

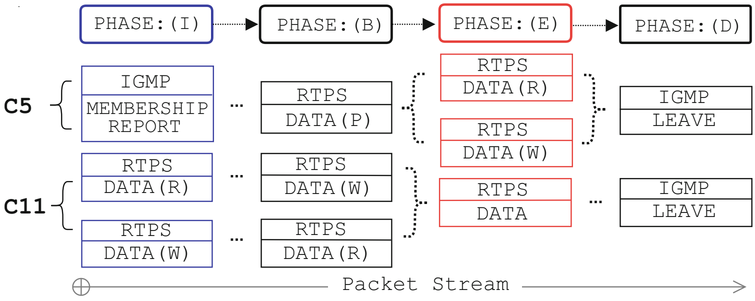

Fig. 3. The constraint tree phases in 2 different constraints.

tree from the instantiate hash table. In the case of more than one bind phases, the constraint tree will be passed from one bind phase to the next through one or more hash tables.

Evaluate is the phase in which the validity of the target packet P_t is evaluated. The triggering packet type P_t of this phase should provide the data for any remaining leaf nodes of the constraint tree. A security violation occurs if the evaluate phase fails to retrieve the tree from the hash table of the previous phase or the tree evaluates to a false value. The evaluate phase for the same tree instance can be executed every time a triggering packet arrives.

Destroy is the final phase of the life cycle which is triggered by the arrival of the final packet, P_f in the predefined packet sequence. As a result of this phase, the constraint tree is deleted from all the hash tables to free up memory.

3.2 The DSL Representation of Network Constraints

It should be noted that for the single-packet constraints, the life-cycle model only involves the evaluate phase for the P_t contents against a known set of environmental values. For two-packet constraints, the model simply involves the instantiate phase for the P_o and the evaluate phase for the P_t. The life-cycle model presented above is particularly useful for those constraints that involve more than two packets in a sequence with multiple alternative valid orders. In this section, we describe the internal DSL representation using two such constraints namely the $C5$ and $C11$. Figure 3 illustrates the possible sequences of packets involved in these two constraints in terms of their constraint tree phases.

Constraint $C5$**.** All subscribers and publishers must be valid members of an IGMP multicast group. These participants must send their membership reports to specific group addresses before showing their interests in a topic.

Constraint $C11$**.** The data of a certain topic is considered valid if it is produced from a valid publisher and consumed by a valid subscriber.

As reflected in Fig. 3, the target packet type to evaluate constraint $C5$ is `RTPS.DATA(W)` (the publisher) or `RTPS.DATA(R)` (the subscriber) submessage of an `RTPS` packet. The previous packet types, in order, are an `RTPS` packet containing a participant submessage, `RTPS.DATA(P)`, and an `IGMP` membership report packet. The IDS parser that we implemented can parse callbacks not only for the entire packets but also for the meaningful parts of the packets such as the RTPS submessages. Therefore, the constraint engine can use the RTPS submessage as a trigger for its phase logic.

```
 1 CONSTRAINT C5 /* Constraint ID */
 2 /* Constraint Tree Expression for C5 */
 3 V(AND(EQ(SrcIPAddressNewJoin, SrcIPParticipant),
 4       EQ(GroupDestIPToBeJoined, DestIPParticipant)))
 5 INSTANTIATE
 6 IGMP Packet.Type is V2Report
 7  if not SEARCH Packet.srcIP, IGMP.groupaddr :Hash=hashIC5
 8     Tree.SrcIPAddressNewJoin = Packet.srcIP
 9     Tree.GroupDestIPToBeJoined = IGMP.groupaddr
10     Key = Packet.srcIP, IGMP.groupaddr
11     HashInstantiate = hashIC5
12  endif
13 BIND
14 RTPS FULL_RTPS.Type is DATAPSUB
15      REPEAT
16      HashInstantiate = hashIC5
17      KeyInstantiate = Packet.srcIP, Packet.dstIP
18      Tree.SrcIPParticipant = Packet.srcIP
19      Tree.DestIPParticipant = Packet.dstIP
20      Key = Packet.srcIP
21      HashBind = hashBC5
22 EVALUATE
23 RTPS FULL_RTPS.Type is DATARSUB Or FULL_RTPS.Type is DATAWSUB
24      HashBind = hashBC5
25   if SEARCH Packet.srcIP :Hash=hashBC5
26      EVAL Packet.srcIP
27   endif
28 DESTROY
29 IGMP Packet.Type is V2Leave
30   if SEARCH Packet.srcIP:Hash=hashBC5
31      HashBind = hashBC5
32      Key = Packet.srcIP
33   endif
34 END
```

Listing 1.1. The DSL Code for Constraint $C5$

Listing 1.1 shows the DSL representation for the constraint $C5$. The first line of the code specifies the constraint ID (line 1). The constraint ID is a unique identifier value that is used as a suffix in the generated code for each constraint. Next, in line 3–4, we have the constraint tree expression in prefix notation. The keywords `INSTANTIATE`, `BIND`, `EVALUATE`, and `DESTROY` correspond to the life-cycle phases of the constraint tree, each followed by a triggering packet type as specified in Fig. 3. For instance, the instantiate phase is triggered when reporting the multi-cast group address of a network; i.e., arrival of the IGMP Membership Report packet. The `V2REPORT` in line 6 refers to the Membership Report packet from the IGMP version 2 specification. The packet type `DATAPSUB` in line 14 refers to an RTPS participant submessage, `RTPS.DATA(P)` which triggers the bind phase. The packet types `DATARSUB` and `DATAWSUB` in lines 23 correspond to the RTPS subscriber and publisher submessages. In line 29, the `V2Leave` packet type corresponds to the leave submessage from the IGMP version 2 specification.

Tree Instance Hashing. The DSL proposed by Hasan et al. [9] had two implicit hash tables to hold the constraint tree between the instantiate and bind phase, and between the bind and evaluate phase. In our DSL, each of the instantiate and bind phase can have an unlimited number of defined hash tables. In Listing 1.1, line 10 shows an example of assigning key fields from the packet while line 11 defines the hash table name `hashIC5`. During the bind phase, the constraint engine should be able to find the constraint tree in the instantiate hash table. The constraint engine uses key fields from the triggering packet to lookup the constraint tree instance created in the instantiate phase. In Listing 1.1, the hash table name for the instantiate is defined in line 16 while the necessary key fields are defined in line 17.

The bind phase updates the constraint tree by filling the empty leaves with the appropriate packet fields as in lines 18–19 of Listing 1.1. The bind phase stores the updated constraint tree in the bind hash table. In Listing 1.1, the hash table name and key fields for bind phase are defined in lines 20–21. The hash table name and key fields used by the evaluate phase are defined in lines 24 and 26. Finally, the destroy phase deallocates the constraint tree from the defined hash tables. Based on the constraint specification, the autogenerated constraint engine may have an unlimited number of hard-coded definitions of hash tables for the instantiate and bind phases.

Hash Table Naming. In our current DSL, each hash table should have a unique name throughout the constraint engine code. In Listing 1.1, the variable `HashInstantiate` holds the hash table name for the instantiate phase of constraint $C5$. While the variable `HashBind` holds the hash table name for the bind phase. The constraint phases should access any of the hash tables by their unique names. For example, the bind phase should be able to find the constraint tree allocated previously at the instantiate phase. Therefore, the `HashInstantiate` is specified twice in the DSL of Listing 1.1 - first in the instantiate clause and the next in the bind clause. The DSL grammar allows the naming of multiple hash tables when the phase clause holds multiple cases. Listing 1.2 shows an example of multiple hash tables within the same phase clause for constraint $C11$. In $C11$ example, the instantiate phase has two hash tables: `hashIC11_DATARSUB` and `hashIC11_DATAWSUB`. Each one of these hash tables is accessed at different cases based on the logical sequence.

Hash Table Management. Figure 4 summarizes the hashing mechanism among the four phases of a constraint tree. During the instantiate phase, the constraint engine instantiates a tree for the constraint C_x and inserts its reference to a hash table. The location of the tree reference in the hash table is determined by the hash key value. The packet fields used to calculate the key value should be specified in the DSL. For example, Listing 1.1 for $C5$ defines `srcIP` and `groupaddr` as the key fields for its hash table `hashIC5`. Based on these two fields, a reference to constraint tree is located in the hash table `hashIC5`. In

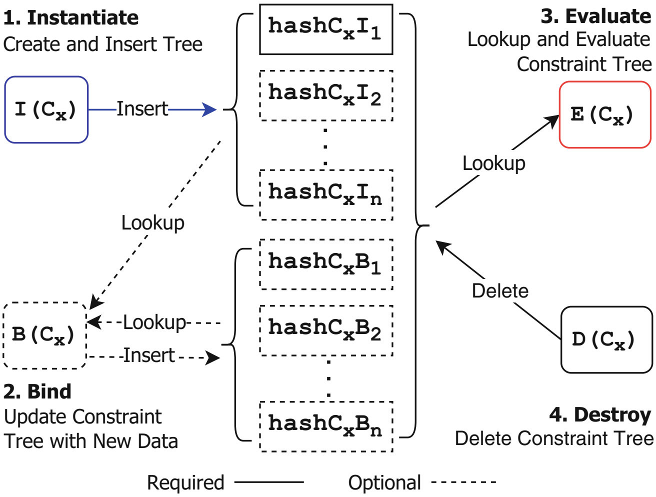

Fig. 4. The constraint tree phases and hashing.

some cases, the calculated key value may point to an already existing hashing slot causing a collision [23]. In such a case of collision, the old tree is deleted and the hash slot is updated with a new tree instance. At the bind phase, the instantiate tree reference is copied from the instantiate hash table into the bind hash table. In $C5$ example, the bind hash table is `hashBC5`. At the end of the bind phase, the copied tree reference is deleted from the instantiate hash table to free up space. However, the DSL allows disabling this deletion by adding the keyword `SPLIT` in the bind clause. In case of collision, the tree update can be ignored by adding the keyword `REPEAT`. For example, Listing 1.1 for $C5$ has the keyword `REPEAT` at line 15. During the evaluate phase, the constraint tree is retrieved from the hash table specified by the name in variable `HashBind`, if available. Otherwise, the constraint engine should report a security violation. The evaluate phase does not change the content of the hash tables. In our approach, the generated hashing uses the linear probing algorithm with a relatively prime step size [7]. However, other hashing algorithms that support deletion can be used.

Constraint Generalization. The automatic generation approach takes into consideration the generality of constraint specifications. One of the challenging generalizations is the cross scenarios handling. The cross scenarios happen when the constraint specification permits the same triggering packets for more than one phase. For instance, the constraint $C11$ in Listing 1.2. The target of the constraint determines if a data message of a given RTPS topic is produced by a valid publisher and consumed by a valid subscriber. Both the instantiate and bind phases can be triggered by a `DATA(W)` or a `DATA(R)` as depicted in Fig. 3. For such a case, the generated constraint engine must conform to the accurate scenario by checking if the other packet has already arrived. We use the `SEARCH` function (in Line 8 and Line 24) to check if the other packet has arrived by the availability of constraint tree in hash table.

In Listing 1.2, we added the `if` condition at the start of each phase case to handle the cross scenario generalization. Another example of generalization is the multiple triggering packets for the same phase. We can observe that $C11$ specification handles such a case by applying a `switch` block. Each packet type in such case will represent a triggering packet type. The same approach can be applied to extend the $C5$ specifications. In $C5$ instantiate and destroy phases, we add switch statements to handle older versions of `IGMP Report` packets. Listing 1.3 shows the two phases after generalization.

```
 1 CONSTRAINT C11
 2 V(AND(EQ(SrcIPPublisher, DesIPfromSubscriber), EQ(PublisherSrtPort),
 3       EQ(entityIDPublisher), EQ(SrcIPSubscriber)))
 4 INSTANTIATE
 5 RTPS FULL_RTPS.Type is DATAWSUB Or FULL_RTPS.Type is DATARSUB
 6 switch (FULL_RTPS.Type)
 7  case DATAWSUB:
 8     if not SEARCH HashKey(.pidtopicname_rtps.topicname.name
 9         IN Protoco~serializeddata~topicdata:type,
10         PIDTOPICNAME_RTPS_VAL), Packet.srcIP, Packet.dstIP
11         :Hash=hashIC11_DATARSUB
12     endif
13  case DATARSUB:
14     if not SEARCH HashKey(.pidtopicname_rtps.topicname.name
15         IN Protocol~serializeddata~topicdata:type,
16         PIDTOPICNAME_RTPS_VAL), Packet.dstIP, Packet.srcIP
17         :Hash=hashIC11_DATAWSUB
18     endif
19 endswitch
20 BIND
21 RTPS FULL_RTPS.Type is DATAWSUB Or FULL_RTPS.Type is DATARSUB
22 switch (FULL_RTPS.Type)
23 case DATARSUB:  /* Case Logic */
24     if SEARCH HashKey(.pidtopicname_rtps.topicname.name IN Protocol~
       serializeddata~topicdata:type,PIDTOPICNAME_RTPS_VAL), Packet.dstIP,
       Packet.srcIP :Hash=hashIC11_DATAWSUB
25     endif
26 case DATAWSUB:  /* Case Logic */
27     if SEARCH  HashKey(.pidtopicname_rtps.topicname.name IN Protocol~
       serializeddata~topicdata:type,PIDTOPICNAME_RTPS_VAL), Packet.srcIP,
       Packet.dstIP:Hash=hashIC11_DATARSUB
28     endif
29 endswitch
```

Listing 1.2. The DSL Code for Constraint $C11$

```
INSTANTIATE
IGMP Packet.Type is V2Report Or Packet.Type is V3Report
switch (Packet.Type)
case V2Report: /* Case Logic */
    if  not SEARCH Packet.srcIP, IGMP.groupaddr :Hash=hashIC5
    endif
case V3Report: /* Case Logic */
    loop for groupaddr in IGMP.grouprecordinfo
        if not SEARCH Packet.srcIP, Iterator.groupaddr :Hash=hashIC5
        endif
    endloop
endswitch
/* The Bind phase followed by the Evaluate phase [Same as Listing 1] */
DESTROY
IGMP Packet.Type is V2Leave Or Packet.Type is V3Leave
```

```
switch (Packet.Type)
case V2Leave:   /* Case Logic */
    if SEARCH Packet.srcIP:Hash=hashBC5
    endif
case V3Leave:   /* Case Logic */
        if SEARCH Packet.srcIP:Hash=hashBC5
        endif
endswitch
END
```

Listing 1.3. DSL Code for Generalized Constraint C5

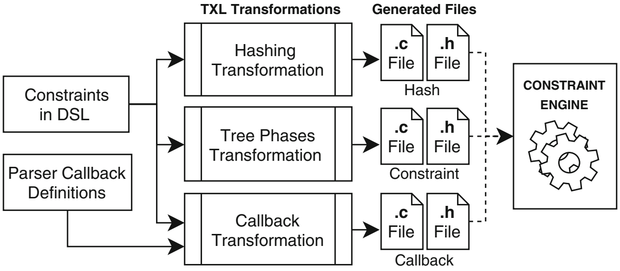

Fig. 5. Automatic generation of the constraint engine code.

3.3 Implementation

An overview of our implemented autogeneration framework for the constraint engine code is presented in Fig. 5. Based on the constraints specified in the DSL input, the framework requires three kinds of transformations to generate the necessary C files for the constraint engine. These include the transformations for the hashing, the constraint tree phases, and the parser callbacks, as depicted in the figure. We used the TXL [5] as the transformation language between the DSL and the constraint engine code. After the transformations, all the generated C files of the IDS are compiled to build a binary executable file. The binary file includes the parser and the autogenerated constraint engine.

Four snippets of the generated code for constraint $C5$ are presented in this section. First is the example of `hash.h` file in Listing 1.4 that includes the function prototypes of the generated hashing operations for the $C5$ constraint tree. Next, in Listing 1.5, we have the generated code for `constraint.h` which includes the function prototypes of the constraint tree phases. Notice how the function parameters correspond to the $C5$ DSL in Listing 1.1. For example, the function `instantiateC5_V2Report(..)` in Line 1 requires a pair of key fields, followed by a pair of leaf node values based on lines 8–10 of the instantiate clause in $C5$ DSL 1.1. Listing 1.6 presents the generated implementation of the $C5$ bind function for the `constraint.c` file. The bind implementation calls different hash functions to retrieve and update each tree instance.

```
typedef struct {
    uint32_t SrcIPAddressNewJoin;
    uint32_t SrcIPParticipant;
    uint32_t GroupDestIPToBeJoined;
    uint32_t DestIPParticipant;
} treeHashC5;
int insertIValueC5V2Report (uint32_t key1, uint32_t key2, uint32_t
     SrcIPAddressNewJoin, uint32_t GroupDestIPToBeJoined);
int deleteValueIhashIC5 (uint32_t key1, uint32_t key2);
treeHashC5* GetValuefromfhashIC5 (uint32_t key1, uint32_t key2);
int insertIValueC5V3Report (uint32_t key1, uint32_t key2, uint32_t
     SrcIPAddressNewJoin, uint32_t GroupDestIPToBeJoined);
int insertBValueC5hashBC5 (uint32_t key1, uint32_t SrcIPParticipant,
     uint32_t DestIPParticipant, treeHashC5* treeInst);
int deleteValueBhashBC5 (uint32_t key1);
treeHashC5* GetValuefromfhashBC5 (uint32_t key1);
void clearC5 ();
```

Listing 1.4. Generated Code in `hash.h` for $C5$.

```
int instantiateC5_V2Report (uint32_t key1, uint32_t key2, uint32_t
     SrcIPAddressNewJoin, uint32_t GroupDestIPToBeJoined, unsigned long
     pktCount);
int instantiateC5_V3Report (uint32_t key1, uint32_t key2, uint32_t
     SrcIPAddressNewJoin, uint32_t GroupDestIPToBeJoined, unsigned long
     pktCount);
int bind1C5 (uint32_t instatiate_key1, uint32_t instatiate_key2, uint32_t
     key1, uint32_t SrcIPParticipant, uint32_t DestIPParticipant, unsigned
     long pktCount);
int evaluateC5 (uint32_t key1, unsigned long pktCount);
int destroyC5_V2Leave (uint32_t key1, unsigned long pktCount);
int destroyC5_V3Leave (uint32_t key1, unsigned long pktCount);
```

Listing 1.5. Generated Code in `constraint.h` for $C5$.

```
int bind1C5 (uint32_t instantiate_key1, uint32_t instantiate_key2, uint32_t
      key1, uint32_t SrcIPParticipant, uint32_t DestIPParticipant, unsigned
      long pktCount){
if (true){
  treeHashC5* insTree=GetValuefromfhashIC5(instantiate_key1,
     instantiate_key2);
  if (insTree==NULL){iC5f++; iC5bindf++; return -1;}
   else {}
  treeHashC5* bindTree=GetValuefromfhashBC5(key1);
  bool noRepeat=false;
  if (bindTree!=NULL){if (!noRepeat) return 1;}
  bool split=false;
  if (split){
    treeHashC5* splitedTree=malloc(sizeof(treeHashC5));
    memcpy (splitedTree, insTree, sizeof(treeHashC5));
    bindTree=splitedTree;}
    else if (!noRepeat){
     treeHashC5* splitedTree=malloc(sizeof(treeHashC5));
     memcpy (splitedTree, insTree, sizeof(treeHashC5));
     bindTree=splitedTree;}
       else {bindTree=insTree;}
  if (insertBValueC5hashBC5 (key1, SrcIPParticipant, DestIPParticipant,
     bindTree) == -1)
       {iC5f++; iC5bindf++; return -1;}
  if (noRepeat){if (!split){if (deleteValueIhashIC5 (instantiate_key1 ,
     instantiate_key2) == -1){
       iC5f++; iC5bindf++; return -1;}}}
  iC5bind++; return 1;}
else return 1;}
```

Listing 1.6. Snippet of Generated Code in `constraint.c` for $C5$

```
void V2Report_IGMP_callback(V2Report_IGMP *v2report_igmp, PDU *thePDU){
    struct HeaderInfo *Packet = thePDU->header;
    instantiateC5_V2Report (Packet->srcIP,
        v2report_igmp->groupaddr, Packet->srcIP,
        v2report_igmp->groupaddr, Packet->pktCount);}
```

Listing 1.7. `IGMP V2Report` with a trigger call to $C5$ Instantiate Phase.

Finally, Listing 1.7 shows an example of triggering the $C5$ instantiate phase within the generated callback function of `IGMP V2Report` based on the instantiate phase of the $C5$ DSL in Listing 1.1. The callback function headers are predefined in the parser and used as inputs for the callback transformation. The callback functions passes the triggering packets defined in the constraint specifications and fills the required parameters of the tree phases. The transformation keeps appending the callback functions into the `callback.c` file for different triggering packets based on the constraint tree phases.

4 Results and Evaluation

The generated IDS has been tested against four different constraints with the test cases listed in Table 1. Each of the cases in the table was intentionally induced in separate packet capture (pcap) files. As shown in Table 1, each of the four constraints has a normal scenario where the IDS should pass with no violations, along with a set of abnormal cases with anomalous packets. The violation checks reported by the generated constraint engine are consistent with the test case expectations, proving the correctness of the constraint engine code.

Table 1. Test cases with four constraints.

	$C5$ Tests	$C8$ Tests
✓	Normal	Normal
✗	Missing `IGMP` Packet in the Sequence	Wrong `Topic Name` in `DATA(W)`
✗	Wrong `Group Address` in `IGMP Report`	-
	$C11$ Tests	$C12$ Tests
✓	Normal	Normal
✗	Wrong `SrcIP` in `IGMP DATA(W)`	Wrong `SrcIP` in `IGMP DATA(W)`
✗	Wrong `DestIP` in `IGMP DATA(W)`	Wrong `SrcIP` in `IGMP DATA`

Figure 6 shows the performance results on the run-time averages of the two IDS versions on three pcap files. For our evaluation, we used the same set of pcap files used by Hasan et al. [9] against the same three constraints: $C5$, $C11$ and $C8$. We run each IDS 10 times for each pcap file. Each of the IDS was ran on an Ubantu Linux 64-bit VM with Intel core i7 processor using 2 GB of RAM.

As we can observe in Fig. 6, the autogenerated IDS is slower with a maximum drop of less than 6%. The results indicate that there is a slight drop in the

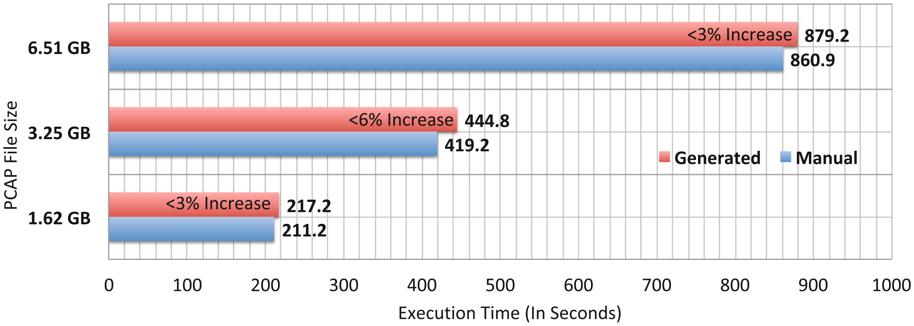

Fig. 6. Performance comparison between the constraint engines.

performance for the autogenerated IDS in contrast to the manually implemented version. However, the performance difference is insignificant. These results highlight that the automatic generation of the constraint engine presented in this paper does not lead to a significant performance penalty.

5 Related Work and Discussion

While there exist various approaches to develop an IDS [12], the premise of our approach is an anomaly-based IDS [8] for limited-access closed-networks which involve a limited number of protocols. Various anomaly-based approaches have been studied that apply Machine Learning (ML) techniques [22] to identify suspicious patterns in the network traffic. However, the ML-based approaches are typically useful for conventional public networks that involve a broad variety of protocols with variable traffic patterns that are hard to prognosticate. Since the target networks of our IDS solely involve a limited number of protocols with restrictive operational patterns, we did not consider any ML-based technique for our approach. Our approach is based on the idea of inspecting each of the network packets against a finite set of protocol-specific constraints that are already known. We considered achieving real-time performance and automatic generation of our constraint-based IDS [9] to be the key challenges for our approach.

Satisfying network constraints can be considered similar to finding frequent patterns in the data stream. Various tree-based approaches have been considered in literature for different applications [11,18,20]. A key limitation of these approaches is that they require maintaining large constraint trees to express their desired patterns. Against streaming data in large scale, even with efficient pruning algorithms, traversing and updating such constraint trees would require continuous memory operation which is bound to cause major bottlenecks in performance. In our approach, using the constraint tree life-cycle model we instantiate a simple tree for each of the individual constraints with optimized memory management. Using the hashing mechanism presented in this paper, our approach allows accessing the required tree nodes in the memory or removing a tree from memory, both in constant time.

Open source tools such as SNORT [17] support rule-based intrusion detection techniques. The SNORT rules [10] can be used to express similar concerns comparable to the single-packet environmental constraints in our approach. However, when it comes to multi-packet constraints involving a set of valid sequence of packets, expressing them using SNORT-based rules are not feasible. Writing such rules would require specifying the exact time frames for the expected arrival of related packets. In contrast, our approach allows defining the partial-order patterns on the required set of packets without the need to specify any time window. In a related effort of using a DSL to express network constraints, Salgueiro et al. [19] present a DSL to describe common attacks on TCP/IP protocols that can generate solution code. However, the DSL was not designed to handle the kind of multi-packet constraints that we needed for the DDS networks.

Generating automated code has been used in various Model-Driven Engineering projects. These projects usually involve generating operational code from a model specifying the system to be created. Our code generation approach is distinct in that we are generating the code to validate the network data against the models. The generated constraint engine code by our IDS framework checks the validity of the traffic data against a set of protocol-specific behavioural models of the target networks.

6 Conclusion and Future Work

In this paper, we have presented an effective approach to autogenerate a constraint engine through a practical IDS framework. The approach implements and extends our earlier DSL proposed by Hasan et al. [9] and generalizes its representation to accommodate any kind of complex multi-packet constraints that involve alternative sequences of packets with arbitrary length and order. We have successfully implemented a mechanism to transform the internal DSL into executable constraint engine code using TXL.

Comparing the performance of the autogenerated constraint engine against the manually implemented version displays no significant performance overhead. The correctness of the generated constraint engine was evaluated based on four different test cases. For our next evaluation, we plan to involve a third-party Red team to incorporate a comprehensive collection of malicious activities on our simulation environment for an ATC network.

We are currently working on generating the internal DSL from an SCL-based high-level constraint specification language suitable for the Network Engineers to use. The SCL language, already being used to generate the parser for our IDS framework, is modular, which allows easy specification of multiple protocols for a given network. Finally, since the IDS framework only allows autogenerating its constraint engine from a high-level specification, the approach requires minimum human interventions. The intricate implementation details of the constraint engine are not needed to be coded or managed by humans which eliminates the possibility of having human-induced errors and heavy maintenance overhead when adapting a new set of protocols and systems for the IDS.

Acknowledgements. This research is supported by the Department of National Defence (DND), Canada, The Natural Sciences and Engineering Research Council of Canada (NSERC), and The Ontario Research Foundation (ORF), Canada.

References

1. NAV CANADA. The Canadian Automated Air Traffic System. https://takecharge.navcanada.ca/. Accessed 11 May 2018
2. Baud, O., Honore, N., Taupin, O.: Radar/ADS-B data fusion architecture for experimentation purpose. In: 2006 9th International Conference on Information Fusion, pp. 1–6. IEEE (2006)
3. Cain, B., Deering, S., Kouvelas, I., Fenner, B., Thyagarajan, A.: Internet group management protocol, version 3. Technical report (2002)
4. Callaghan, B., Pawlowski, B., Staubach, P.: NFS version 3 protocol specification. Technical report (1995)
5. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3), 190–210 (2006)
6. ElShakankiry, A., Dean, T.: Context sensitive and secure parser generation for deep packet inspection of binary protocols. In: 2017 15th Annual Conference on Privacy, Security and Trust (PST), Calgary, Canada, pp. 77–7709. IEEE (2017). https://doi.org/10.1109/PST.2017.00019
7. Flajolet, P., Poblete, P., Viola, A.: On the analysis of linear probing hashing. Algorithmica **22**(4), 490–515 (1998)
8. Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., Vázquez, E.: Anomaly-based network intrusion detection: techniques, systems and challenges. Comput. Secur. **28**(1–2), 18–28 (2009)
9. Hasan, M.S., Dean, T., Imam, F.T., Garcia, F., Leblanc, S.P., Zulkernine, M.: A constraint-based intrusion detection system. In: Proceedings of the 5th European Conference on the Engineering of Computer-Based Systems, ECBS 2017, pp. 12:1–12:10. ACM, New York (2017). https://doi.org/10.1145/3123779.3123812
10. Khamphakdee, N., Benjamas, N., Saiyod, S.: Improving intrusion detection system based on SNORT rules for network probe attack detection. In: 2014 2nd International Conference on Information and Communication Technology (ICoICT), pp. 69–74. IEEE (2014)
11. Kurien, L.S., Sreekumar, K., Minu, K.K.: Survey on constrained based data stream mining. Int. J. Comput. Appl. **107**(16), 12–15 (2014)
12. Liao, H.J., Lin, C.H.R., Lin, Y.C., Tung, K.Y.: Intrusion detection system: a comprehensive review. J. Netw. Comput. Appl. **36**(1), 16–24 (2013)
13. Marquis, S., Dean, T.R., Knight, S.: SCL: a language for security testing of network applications. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 155–164. IBM Press (2005)
14. Mills, D., Martin, J., Burbank, J., Kasch, W.: Network time protocol version 4: Protocol and algorithms specification. Technical report (2010)
15. Pardo-Castellote, G.: OMG data-distribution service: architectural overview. In: 2003 Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops, pp. 200–206. IEEE (2003)
16. Object Management Group: The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification OMG Document Number: formal/2014-09-01. http://www.omg.org/spec/DDSI-RTPS/2.2. Accessed April 2019

17. Park, W., Ahn, S.: Performance comparison and detection analysis in SNORT and Suricata environment. Wirel. Pers. Commun. **94**(2), 241–252 (2017)
18. Potharst, R., Feelders, A.J.: Classification trees for problems with monotonicity constraints. In: Special Interest Group (SIG) on Knowledge Discovery and Data Mining Explorations (SIGKDD), vol. 4, pp. 1–10 (2002)
19. Salgueiro, P., Abreu, S.: On using constraints for network intrusion detection. In: INForum (2010)
20. Silva, A., Antunes, C.: Pushing constraints into data streams. In: Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, pp. 79–86. ACM (2013)
21. Sollins, K.: The TFTP protocol (revision 2) (1992)
22. Tsai, C.F., Hsu, Y.F., Lin, C.Y., Lin, W.Y.: Intrusion detection by machine learning: a review. Expert Syst. Appl. **36**(10), 11994–12000 (2009)
23. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_6