Using CSP to detect Insertion and Evasion Possibilities within the Intrusion Detection Area

Gordon Thomas Rohrmair and Gavin Lowe

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
{gtr, gavin.lowe}@comlab.ox.ac.uk

Abstract. In this paper we will demonstrate how one can model and analyse Intrusion Detection Systems (IDSs) and their environment using the process algebra Communicating Sequential Processes (CSP) [11, 21] and its model checker FDR [9]. An Intrusion Detection System (IDS) is a system that detects abuses, misuses and unauthorised uses in a network. We show that this analysis can be used to discover two attack strategies that can be used to blind every Intrusion Detection System (IDS), even a hypothetical perfect one that knows all weaknesses of its protected host. We will give an exhaustive analysis of all such attack possibilities and will briefly discuss prevention techniques.

Keywords: intrusion detection, Communicating Sequential Processes, model checking, automated verification.

1 Introduction

This paper introduces the use of Communicating Sequential Processes (CSP) [11, 21] within the intrusion detection area. We will show how one can use CSP and its model checker FDR [9] to analyse the interactions between Intrusion Detection Systems (IDSs) and their environment. Special account is given to unexpected side effects that allow an attacker to render the IDS useless.

An Intrusion Detection System is used to detect abuses, misuses and unauthorised uses in a network, caused by either insiders or outsiders. These systems identify intrusions by spotting known patterns or by revealing anomalous behaviour of protected resources (e.g., network traffic or main memory usage).

With two techniques it is possible to blind Intrusion Detection Systems, even a hypothetical perfect one that knows all weaknesses of its protected host. Although they are easy to understand it took the security community a long time to reveal them [18]. We will give an exhaustive analysis about all attack possibilities that are based on this classes and will briefly discuss prevention techniques.

The motivation for using formal methods is that conventional ways of specifying systems rely heavily on natural language and diagrammatic methods. Such approaches make it harder to write unambiguous specifications and therefore make it more difficult to analyse. If omissions and errors introduced during the specification phase go undetected until late in the development cycle, they become very expensive to rectify. In the security sector, every undetected bug has the potential to cause the loss of reputation, clients and/or information. The real power of formal methods, as mentioned in [6], becomes apparent when we try to detect emergent faults. These originate from unexpected interactions between the single processes of the overall system.

Our approach involves:

1. Building two models of small networks employing an IDS. The first model uses only two fields of the Internet Protocol version 4 (IPv4) [8] within a small network that is protected by a firewall. The firewall is built upon the screened subnet architecture described in [7]. This topology consists of two routers, placed either side of the IDS.
   The extended model allows fragmentation and out-of-order communication between the nodes, based upon IPv4. The new fields required for this are the fragment offset and more fragments bits. The overall architecture of the network stays unchanged.
2. Developing a specification that permits one to discover every possibility of an attacker launching an attack that is not seen by the IDS.
3. Using the model checker Failures Divergences Refinement (FDR) [9] for automated verification of our models. FDR finds that the specifications are not met and provides us with counterexamples which represent blinding possibilities.

The next section provides background information regarding Intrusion Detection. Section 3 shows how we modelled the Time-To-Live model described in [18, 17] and discusses results that can be drawn from the FDR analysis. Section 4 deals with a more complex model that emulates the reassembly process of IPv4. Finally, in Section 5 we present some conclusions, together with a discussion of interesting questions for future work.

2 Background

In this section we describe necessary background information.

2.1 Classes of Intrusion Detection Principles

We can classify three different detection principles of Intrusion Detection Systems (IDSs).

Misuse detection Misuse detection based systems look for known signatures of attacks. A *signature* is the pattern that is used by the IDS to spot attacks [12]. It is a specific manifestation of a certain attack. Examples of such systems can be found in [3]. The signatures that are necessary for these systems are mostly developed by hand. The IDS usually obtains the required information from a network adaptor, which feeds it with raw data packets, or from the log-files of the hosting operating system. In industry the most used systems are network signature based IDSs, because of their low total cost of ownership.

Advantages The system knows exactly how a certain attack manifests itself. This leads to a low false-positive ratio. The detection algorithm is based on pattern matching, for which efficient solutions exist.

Disadvantages Defining the manifestations of certain attacks is a time consuming and difficult task. Due to the working principle of these systems, it is nearly impossible for them to detect novel attacks. Moreover subtle variations in the attack can mislead them.

Anomaly detection Such systems distinguish between normal and anomalous behaviour of guarded resources. Examples of monitored resource characteristics include CPU utilisation, system call traces, and network links. The decision regarding which behaviour class currently relates to certain events is made by means of a set of profiles. The profiles of normal behaviour for a resource are maintained by a self-learning algorithm. Examples of such systems can be found in [3].

Advantages The cost of maintaining the system is usually low, because the profiles are updated by the self-learning algorithm. Additionally, it can detect novel attacks as well as variations of already known ones.

Disadvantages The self-maintaining algorithm is usually computationally expensive. Sometimes unusual behaviour is not a precise indicator of an ongoing intrusion. The result is a high false-positive ratio. Finally, these systems can learn to classify intrusive event traces that are performed slowly as normal behaviour, which renders them useless.

Specification-based detection [13, 14] and [24] were some of the first papers that recommended this approach. They distinguished between normal and intrusive behaviour by monitoring the traces of system calls of the target processes. A specification that models the desired behaviour of a process tells the IDS whether the actual observed trace is part of an attack or not. With this approach, they attempt to combine the advantages of misuse and anomaly detection. It should reach the accuracy of a misuse detection system and have the ability to deal with future attacks of anomaly detection. Their systems managed the detection by inspecting log files. This differs from [26], where a run time engine was developed to detect violations in real time. This approach is even capable of intercepting intrusions.

Advantages More or less the same as for misuse detection. However these systems manage to detect some types/classes of novel attacks. Additionally, they are more resistant against subtle changes in attacks.

Disadvantages Usually for every program that is monitored, a specification has to be designed. Furthermore, the modelling process can be regarded as more difficult than the design of patterns for misuse detection systems. Additionally some classes of attacks are not detectable at all.

2.2 Classes of Intrusion Detection Data Sources

Another classification style that is important is to distinguish the IDS according to their data source.

Network Intrusion Detection Systems (NIDS) A NIDS gets its information from a network adapter operating in promiscuous mode. It examines the traffic for an attack symptomatic signature. Although anomaly detection has been implemented for these systems, the main detection principle remains the misuse detection. A NIDS can provide surveillance for a whole network, because it is working with the raw network packets. In our further examination we model a NIDS because it is the most used system type [25].

Advantages Due to the fact that a single NIDS can monitor a whole network its implementation and maintenance cost are low. Additionally these systems, since they work at the packet level, have all the information to sift out the difference between hostile intentions and friendly intentions.

Disadvantages These systems are largely unable to read the traffic of encrypted connections. The only exception would be to include them into the security association; however this can be regarded as computationally too expensive. Nevertheless, users have increasingly encrypted their communication, rendering the system obsolete. Additionally, increasingly networks are switched rather then broadcasted. Due to the Ethernet working principle [4], the NIDS is only able to collect packets that travel through its collision domain. In a switched environment there is no real collision domain, hence the NIDS is not able to retrieve vital information [28].

Host Intrusion Detection Systems (HIDS) The HIDS runs on a specific host and watches its logging activity. Hence, these systems are operating system dependent and every protected host needs a separate IDS [27]. They can keep track of all actions that are made by the users of that host which include browsing for files with the wrong read/write permissions, the adding and deleting of accounts, and the opening and closing of specific files. This gives these systems a greater aptitude for surveillance of security policy violations.

Advantages The system knows whether or not an attack is successful. It usually produces a reduced number of false-alarms caused by unsuccessful attacks; for example, a HIDS protecting a Linux host would not raise an alarm if an attacker sends a Microsoft IIS Buffer Overflow Attack against this host. HIDS have more monitoring variables than NIDS. Because HIDS reside on the target host, these systems are able to keep track of encrypted end-to-end connections. Some of them even watch the packets as they traverse up the TCP/IP stack, allowing them to cut out of the stack a packet that would lead to a security policy violation. Another advantage of this structure is that HIDS do not require additional hardware.

Disadvantages For every monitored node a HIDS is required. This makes them very expensive in maintenance. HIDS are operating system dependent, therefore different implementations for different operating systems are required which makes them expensive in development. They also reduce the operational capacity of a network node, because they run their analysing processes in parallel with the business applications. Additionally, once an attack succeeds the security officer has to trust information collected from a corrupted host.

3 The Time-To-Live model

In this section we present our first model. We consider whether the Internet Protocol version 4 (IPv4) [8] gives an attacker the opportunity to launch an undetected attack against the target. We first discuss how we can represent the protocol.

1. We need the data field, otherwise we could not communicate with the nodes in our simulated network.
2. Additionally, we include the Time-To-Live (TTL) field. The TTL maintains the distance a packet can travel; every router decreases its value by one; once zero is reached the packet is discarded.

As shown in [18] the TTL offers an interesting evasion possibility.

Modelling assumptions The model is built under certain assumptions. One general assumption is that the IDS is a NIDS based on signature detection. We believe that this is the most relevant IDS because of the large usage of these systems. The IDS itself is considered to be perfect, in a sense that it knows all vulnerabilities that could be used to cause a security breach. Additionally we consider only one-way, in-order communication.

We now consider the network topology. We model a network with just one sender and one receiver node. We use a DeMilitarised Zone (DMZ) configuration, which is commonly used in industry [7]. It consists of an exterior filtering router and an internal filtering router (see Figure 1 below); the exterior one is responsible for protecting the network from most attacks; the interior one is the most restrictive one, as it only allows traffic that is permitted for the internal network. The DMZ resides between these two routers; this is the place where companies maintain their public servers, such as the web server. This is also the preferred place for the IDS; due to the limitations of a network IDS, this is the only place where the IDS receives all the traffic that comes from outside. (An alternative place would be in front of the external router; however this IDS would then detect more alerts than are actually relevant: it would include all attacks that are confounded by the exterior router.)

If we find an attack under these restricted conditions, we will know that there is an attack in the real-world.

3.1 The CSP Model

Each datagram consists of a TTL value and some data; hence the channels have the following structure: TTL.DATA. In order to reduce the state space of our model we ought to introduce further restrictions.

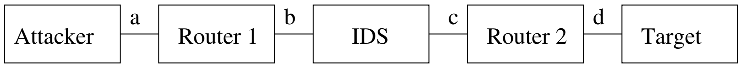

Fig. 1. The network topology

- The TTL value will only range from 4 to 0. We believe that the range of the TTL value is enough because the diameter of the resulting network will be smaller then 4.
- The Data field will communicate the bit patterns `A`, `B` and `C`. `A` represents the bit patterns that, once received, force the target to move into a pre-crashed state, or the IDS into a pre-alerted state. The pre-crashed or pre-alerted state indicates that the system will fail or alert on receiving a `B` bit pattern (which stands for the attack suffix). If the system is not in a pre-crashed or pre-alerted state and receives a `B` it stays in its initial state. In the real world, `A` followed by `B` represent all possible real-world attacks that forces a target or IDS to fail or alert. The final class of bit patterns, `C`, represents all strings that are not part of class `A` or `B`, i.e. the set of innocent patterns that are not part of an attack in any way.

For simulating an appropriate network we require two routers, an attacker, one target and one IDS, as shown in Figure 1; we describe each of these below.

The routers The routers are used for navigating packets from source to destination. Since we have not modelled the source and destination address we can ignore the whole routing functionality; our routers, therefore, act like relay-stations. Taking this simplification into account, we achieve a very simple router that decreases the TTL field by one and checks the result. In the case where the value is zero, the packet will be dropped. Otherwise the packet will be forwarded with the new TTL value. From this we get the following CSP description:

```
Router(in,out) = in?x?y ->
                 if 1<y then out!x!y-1 -> Router(in,out)
                 else Router(in,out)
```

`x` contains the data and `y` the TTL value. The parameters `in` and `out` are channel names that represent the input and output ports of the router.

The attacker The attacker process should be able to execute the same actions as an attacker in the real-world. We model the attacker nondeterministically, so as to impose no limitation on the sequence of packets it sends. Consequently, FDR has to explore every possible input stream that the attacker process may create. The process is modelled by the following CSP description.

```
Attacker = a?x?y -> Attacker
```

The target The target process receives fragments and then reassembles them. Once the packet is reassembled, if an attack signature is found, the target should fail. The following CSP description describes this process.

```
Target(sigs, vulnerability) =
  d?x?y ->
  if member(<x>,vulnerability)
  then fail -> STOP
  else let vulnerability' =
             {s | <x'>^s <- union(sigs,vulnerability), x'==x}
       within Target(sigs, vulnerability')
```

This process is initialised by the two variables `sigs` and `vulnerability`. `sigs` is a set of sequences, namely all complete attack signatures. `vulnerability` keeps track of the progress of security breaches and indicates what the target has to receive in order to fail. The list comprehension is used to update `vulnerability`.

One note about the `fail` event: this event does not mean that the computer crashes literally; it only indicates that the security policy has been violated. This can range from stealing or compromising data to root access and even to crashing.

The IDS The IDS protects the target. We assume here that the IDS is a perfect signature based IDS and therefore knows all vulnerabilities that cause the target to fail. In practice this is impossible because many vulnerabilities are not revealed yet. We have to make this assumption to generalise all current existing IDSs. The following CSP description for the IDS differs from the above target component only in the following two ways: firstly, it forwards all received packets after inspecting them; secondly, it engages in an `alert` event rather than in a `fail` event once it has received an attack pattern.

```
IDS(sigs,alerts) = b?x?y ->
      if member(<x>,alerts)
      then alert -> c!x!y -> STOP
      else let alerts' = {s|<x'>^s <- union(sigs,alerts), x'==x}
           within c!x!y -> IDS(sigs,alerts',dist))
```

The complete model We use parallel composition to synchronise the different processes according to the given network structure (Figure 1). We hide all internal events, leaving just the `alert` and `fail` events visible.

The Specification The specification expresses that there always has to be an `alert` before a `fail` event. In other words, the IDS should have a log-entry once a successful attack was performed. We can model this with a simple recursive CSP process.

```
Spec = alert -> fail -> Spec
```

We use FDR to check whether `Spec` $\sqsubseteq_T$ `Model1` holds, that is, whether the traces of `Model1` are a subset of the traces from `Spec`. The process `Spec` allows precisely the valid traces, so if the refinement holds then the traces of `Model1` are just valid ones, where the IDS detects all attacks; if not, then we have discovered an attack not detected by the IDS.

Results FDR reveals that the refinement check fails, and provides us with the following trace:

```
< a.A.4, b.A.3, c.A.3, a.C.2, b.C.1, d.A.2,
  c.C.1, a.B.4, b.B.3, c.B.3, d.B.2, fail >
```

This trace is displayed in the following sequence diagram.

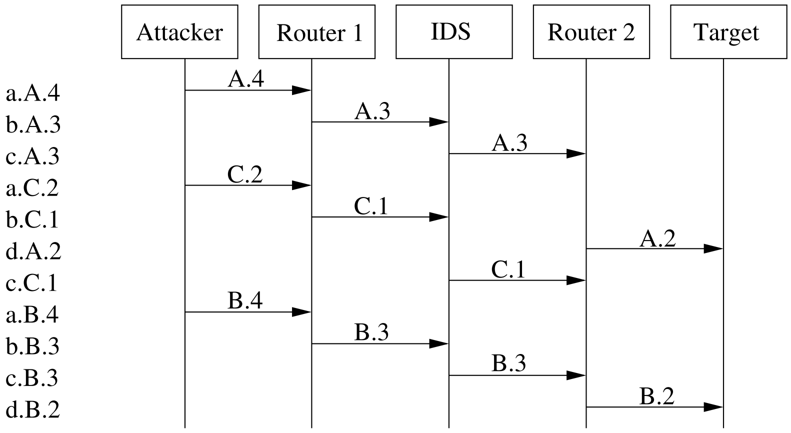

Fig. 2. TTL Attack

This is similar to the observation of Ptacek in [18]. The attacker sends three packets with data `A`, `C` and `B`, respectively, where the packet with data `C` has a TTL value that is lower than its distance to the target. Therefore, this fragment will be discarded from the last router. The IDS, however, takes it into account, so the reassembled packet deviates from the packet that is processed by the target. Hence the target fails, but the IDS does not raise an alert

Attacks like these, where the states of the IDS and target become de-synchronised, are called *de-synchronisation attacks*.

3.2 Discussion

The attack presented above uses the fact that the IDS has not enough information about the topology of the network. We can solve this problem in two ways.

- We could redesign the IDS so that it takes the different distances into account. The drawback of this solution is that the count has to be updated if changes in the network topology occur.
  We designed a CSP model corresponding to this proposed solution; the analysis found no attacks.
- The second possibility is harder to implement; we could implement a reassembly algorithm that raises an alert if the TTL value of one fragment in the stream is different from the others, and this TTL value is lower then the diameter of the network.

Finally, we have to point out that these days it is very unusual for packets to get fragmented; therefore an IDS based on anomaly detection is well suited for spotting this kind of attack.

4 The Fragment-Overlapping Model

In this section we examine the behaviour of a system where data packets are fragmented and reassembled based upon RFC 791 [8].

Sometimes an IP packet has to be routed through different networks. Not all networks have the same properties. Therefore, a packet might have to be split up into fragments tagged with their position in the original packet (fragment offset); this process is termed *fragmentation*. The target receives an increased supply of smaller fragments instead of one IP packet, and therefore has to reconstruct the initial packet; This process is called *reassembly*. The algorithm in [8] collects the fragments and puts them into the right place of the reserved buffer.

Sometimes data is received at the same fragment offset as a previously received fragment. In such a case, a decision has to be made whether to favour old or new data. RFC 791 [8] leaves unspecified which should be preferred, but the recommendation is to prefer new data, so that if the algorithm receives data from the same position twice, the new data will overwrite the old.

However, not all implementations follow this suggestion: favouring new data introduce great danger, as stated in [23], making it possible to circumvent filtering devices; for this reason some operating systems favour old data. Combining operating systems that favour new data (e.g. 4.4 BSD and Linux) with those that favour old data (e.g. Windows NT 4.0 and Solaris 2.6) introduces an evasion possibility if the IDS does not know what type of operating system the target is running; this was first discovered by Ptacek and Newsham [18].

4.1 The CSP Model

To analyse the interactions between the various types of operating systems and IDSs we have designed the following CSP model. The network topology is similar to that in Figure 1, although, For simplicity, we omit the routers.

Channels The channels of this model have to be extended. We require all fields that are necessary to re-assemble the fragment stream, namely the more fragments (MF) bit, which indicates whether this is the final fragment in the packet, and the fragment offset (FO) bit, which indicates the offset of this fragment within the packet.

We use the following channel description:

```
more_fragment_bit . fragment_offset . TTL . data
```

Therefore, event `a.1.0.1.A` represents a packet that travels along channel `a` with its more fragment bit set to one, a fragment offset of zero, a TTL value of one, and a data field containing a bit sequence `A`.

Attacker The attacker process remains unchanged.

Target The target should satisfy the same properties as the target process of the TTL model. Additionally, it should be able to deal with fragments and out-of-order traffic. Thus, it should be able to re-assemble an out-of-order fragment stream just as it is described in RFC791.

In order to consider the behaviour of the different types of operating system—favouring old or new data—we arrange for the target process to choose an operating system initially. The reassembly buffer is initialised to be empty (`<N,N,N>`).

```
Target(sigs) = os_target?os -> Target(os,<N,N,N>,sigs,0)
```

The process `Target(OS,buff,sigs,max)` requires the following parameters: `OS` states whether the IDS is preferring old or new data; `buff` represents the allocated resources that are required for reassembling a packet; `sigs` carries the set of attack signatures; and `max` keeps track of the maximum size of the original packet. The target first receives a datagram and calculates the new buffer `b1` with the function `overwrite`.

```
Target(OS,buff,sigs,max) =
  in?mf?fragmentoffset?ttl?data ->
    let b1 = overwrite(buff,fragmentoffset,data)
    within Target(OS,buff,sigs,max,mf,fragmentoffset,data,b1)
```

The following process models the case were the more fragments flag is equal to zero, indicating that this will be the last fragment. First the process checks whether a fragment with this offset has already been received, and if so acts according to its update policy (favouring old or new data). It then checks whether or not the packet is complete: if the packet is not complete it stores the maximum size of the packet; otherwise it verifies whether it has received an attack or not. The function `nth(a,b)` returns the value stored in position `b` of buffer `a`. `allFilled(a,b)` checks whether all data in buffer `a` up to position (`b`) have been received. `check(a,b,c)` compares the buffer `a` with the set `b` to depth `c` to validate the existence of any attack patterns in the buffer. After the reconstruction of a packet the buffer is initialised with `<N,N,N>`, which indicates a clear buffer.

```
Target(OS,buff,sigs,max,0,fragmentoffset,data,b1) =
    if nth(buff,fragmentoffset) != N and OS == 0
    then Target(OS,buff,sigs,max)
    else if allFilled(b1,fragmentoffset)
         then if check(b1,sigs,fragmentoffset)
              then fail -> STOP
              else Target(OS,<N,N,N>,sigs,0)
         else Target(OS,b1,sigs,fragmentoffset)
```

The following process models the case were a fragment with more fragments bit set to one arrives, indicating that more fragments are following. The structure is nearly the same, except that we do not set any maximum.

```
Target(OS,buff,sigs,max,1,fragmentoffset,data,b1) =
  if nth(buff,fragmentoffset) != N
  then if OS == 0 then Target(OS,buff,sigs,max)
       else Target(OS,b1,sigs,max)
  else if allFilled(b1,max) and max != 0
       then if check(b1,sigs,max) then fail -> STOP
            else Target(OS,<N,N,N>,sigs,0)
       else Target(OS,b1,sigs,max)
```

The IDS The IDS process structure is similar to the IDS model in the improved TTL version in that it considers the distance to the target. It is also capable of re-assembling fragments arriving out-of-order, as the target process presented above. We will not give a full account here because of the similarities to the target. The IDS raises an `alert` instead of a `fail` event and indicates its operating system with `os_ids` instead of `os_target`.

The Complete Model The complete system model is composed of an attacker, a target, and an IDS.

The specification and refinement assertion remain the same as in the TTL example.

4.2 Results

FDR provided us with two distinct attacks that could both elude the IDS.

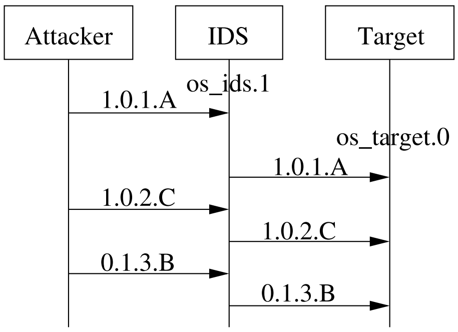

Fig. 3. Attack 1

Attack 1 5 The IDS chooses to use an operating system that favours new data (indicated by the event `os_ids.1`), whereas the target chooses to favour old data (`os_target`). The attacker sends two fragments with fragment offset zero, the first containing a bit sequence `A` (`1.0.1.A`), the second containing an innocent bit sequence `C` (`1.0.2.C`). The result is that the IDS receives the `A` fragment and then overwrites it with `C`. However, the target receives the `A` fragment and refuses to store the `C` fragment, because it favours old data. Therefore we have the situation where in the reassembly buffer of the IDS a `C` bit sequence is stored and in the buffer of the target process an `A` bit sequence is stored. Finally the attacker creates the last packet (`0.1.3.B`), with a fragment offset of one, and the more fragment bit set to zero. Hence on receiving this fragment, both the target and the IDS re-assemble their packets. The IDS re-assembles `<C,B>` and the target re-assembles `<A,B>`, which causes a `fail` event without an `alert`.

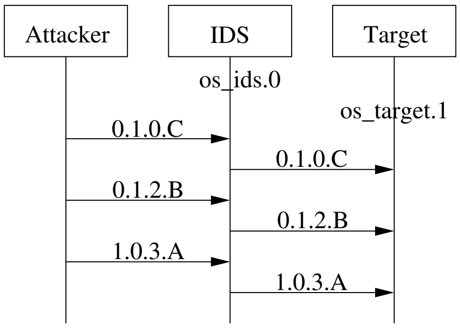

Fig. 4. Attack 2

Attack 2 This attack is the reverse of the former. The IDS chooses to be based on a type zero OS (favouring old data) and the target chooses to be based on a type one OS (favouring new data). The attacker send a fragment with fragment offset one, more fragment bit set to zero, and a `C` bit sequence to the IDS (`0.1.0.C`). Afterwards he submits a fragment with the same fragment offset but with a different bit sequence (`0.1.2.B`). This leads to a deviation of the IDS buffer from the target buffer: the IDS has stored a `C` on its second place, whereas the target overwrites the `C` with a `B`. The attacker then sends the final packet (`1.0.3.A`). The IDS re-assembles `<A,C>`, which is innocent, and the target re-assembles `<A,B>`, which leads to the `fail` event without an `alert`.

4.3 Discussion

There is only really one way to prevent these attacks; that is, to arrange for the IDS to take account of both possibilities for the target, i.e. favouring old or new data. However, this solution does not appear to scale well. There are many differences in the way implementations treat the TCP/IP stacks, and it would appear that the IDS needs to consider all such possibilities. Even if we only consider the drop points—operations were the packet is completely rejected—described in [18], there are a vast number of desynchronisation possibilities. The consequent state-space explosion in the IDS would appear unmanageable.

Another subtle point in our reassembly model was missing, namely time: both the IDS and target will timeout if a packet is not completely received within a certain time.

The use of timeouts allows a different de-synchronisation attack. Consider the case were the timeout value of the IDS is smaller than the value for the target: then the attacker can send its first fragment and wait until the IDS times-out before sending the remaining fragments, causing the agents to become de-synchronised. This attack reveals an interesting point: we can never be certain whether our abstractions have removed details that allow attacks. Because of this we need a proper framework to prove formally that our abstractions have not lost too much detail.

5 Conclusion

We have seen how small deviations in implementations can have a considerable impact on the security of a system. Even when the individual subcomponents of a system are secure, the overall system may be still not free from flaws. Such emergent faults can be spotted easily by testing the system as a whole against a specification.

5.1 Generalising the retrieved results

Whenever it is possible to create a difference between the input stream of the IDS and the protected system we can successfully hide an attack. More generally, both the protected system and the IDS have state transition graphs; if we create a situation where these systems change into different states, they require different stimuli to reach the deprecated state where the target fails and the IDS raises the alert. We can distinguish between three de-synchronising possibilities:

1. De-synchronisation due to the systems behaving exactly the same, but the input streams being different; this is the method that appears in our first model.
2. De-synchronisation because the input streams are the same, but the systems behave differently under certain conditions; this is the type of flaw exploited in our second model.
3. De-synchronisation because both the input streams and the behaviour of the systems are different.

5.2 Related Work

Vulnerability identification and analysis has been a topic for some time now. One of the first attempts to build an automatic vulnerability detector was COPS. In common with other approaches (SATAN, TIGER and NESSUS [15]) it looks for already known attacks. It manages this by firing known attack patterns against a particular host. However this kind of direct testing is not suitable for spotting unknown attacks.

In [19], Ritchey and Ammann propose a high level approach to detect whether it is possible for an attacker to leverage the existing vulnerabilities in the network to get a higher degree of access. The configurations of the network nodes were abstracted to state machines and the attacks were represented as state transitions in these machines.

In contrast to that [22] uses a low level description of a UNIX file system to spot single configuration vulnerabilities. The focus is more on finding new attacks rather then finding a combination of attacks that enables the attacker to penetrate the system even more. The differences between ours and Sekar's approach lies in the state space: they use an infinite model and we use approximation to restrict our model to a finite one; therefore, we can use a common model checker such as FDR. Additionally they use invariants to manifest their security policy in contrast to using a specification, as we do. The difference becomes clear during the examination of the counter examples: they have to establish an intentions model to prune away all paths that are against the defined invariant but do not violate the security policy; we have no such paths.

Another approach of using model checkers for vulnerability identification is proposed in [1, 2]. However they use the model checker combined with mutation testing techniques to generate and recognise test sets rather then testing direct.

The advantages of our approach are:

- The approach finds all possible attacks, not just known ones, due to the working principle of our model checker FDR.
- Easy to understand counter examples.
- Due to the modularity of our models the workload to add new processes or to change the network is low.
- We use a finite model, which allows us to use common model checkers.
- Our testing is specification based, which keeps the effort of encoding security policies low.

5.3 Future work

As shown so far, the CSP approach is a suitable way for identifying emergent faults. However, most of the above examples are outdated. Therefore, we will explore some new protocols and their impact on the intrusion detection area. IPv6 [16, 10] is the upcoming protocol, so we will model and analyse it.

On a different track, in the long term it is quite unsatisfying only to be capable of verifying IDSs that are based on signature detection. Therefore we have to solve following problems:

- How can we classify and represent vulnerabilities and attacks? We have to find a better way than representing an attack as a bit pattern, because some IDSs are not signature based.
- How does this technique apply to industrial-scale problems? Usually model checking can only be applied to small or medium sized problems. A multi layer abstraction framework needs to be established to address large-scale problems. One possibility could be to close the gap between the Ritchey and Ammann [19] approach and ours. Hence it would be possible to evaluate the interactions between single modules on a low level and to convert the obtained results into a high level model to analyse the relations between the spotted vulnerabilities.
- How can we abstract a network or a network node without losing too many details? As mentioned before the problem with abstraction is that one may lose important detail. It would be of great use to design a technique that allows us to prove whether the lost detail was important or not.

– How can we prove that our observed scope is enough? Due to the working principle of FDR we were forced to restrict the state space. We achieved this by choosing small ranges for all variables. We want to prove that these restrictions do not hide other attacks. To do so we have to show that any attack which depends on the usage of larger values can be reduced to one that uses values within our scope. Data-independence techniques [5, 20] might be useful to base this on a formal foundation.

References

1. Paul E. Ammann and Paul Black. Test generation and recognition with formal methods. 2000.
2. Paul Ammann, Wei Ding, and Daling Xu. Using a model checker to test safety properties. 2000. ISE Department, MS 4A4, George Mason University, 4400 University Drive Fairfax, VA 22030 USA.
3. Stefan Axelsson. Research in intrusion-detection systems: A survey. page 98, 1999.
4. Uyless Black. *TCP/IP and Related Protocols*. Computer Communications. McGraw-Hill, 1998.
5. P.J. Broadfoot, Gavin Lowe, and A.W. Roscoe. Automating data independence. *In Proceedings of ESORICS*, pages 175 – 190, 2000.
6. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28 (4), Dec 1996.
7. D. Brent Chapman, Elizabeth D. Zwicky, and Simon Cooper. *Building Internet Firewalls*. O'Reilly, Jun 2000. ISBN: 1-56592-871-7.
8. Marina del Rey. RFC 791 Internet Protocol DARPA Internet program protocol specification. Sep 1981. California 90291.
9. Paul Gardiner, Michael Goldsmith, Jason Hulance, David Jackson, A.W. Roscoe, and Bryan Scattergood. *FDR2 User Manual*. Formal Systems (Europe) Ltd., fifth edition edition, 2000.
10. R. Hinden and S. Deering. RFC 2460 Internet Protocol, Version 6 (IPv6) specification. Dec 1998.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN: 0-13-153271-5.
12. Frederick Karen Kent. Network Intrusion Detection Signatures - Part 1. `http://www.securityfocus.com`, Dec 19 2001.
13. Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. 1994. Department of Computer Science University of California, Davis, CA 95616.
14. Calvin Ko. *Execution Monitoring of Security Critical Programs in a Distributed System: A Specification-Based Approach*. PhD thesis, Department of Computer Science, University of California at Davis, 1996.
15. Nessus a remote security scanner. `http://www.nessus.org/`.
16. Commission of the European Communities. New generation Internet - priorities for action in migrating to the new Internet protocol IPv6. COM(2002) 96 final:15, Feb 2002.
17. Vern Paxton. BRO: A system for detecting network intruders in real-time. *Computer Networks*, 31:2435 – 2463, Dec 1999.
18. Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. *Secure Networks*, Jan 1998.

19. R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. *IEEE Oakland Symposium on Security and Privacy*, pages 156 – 165, May 2000.
20. A.W. Roscoe and P.J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security: Special Issue CSFW12*, 7 (2,3):147–190, Jul 1999.
21. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. ISBN: 0-13-674409-5.
22. C.R. Ramakrishan and R. Sekar. Model-based analysis of configuration vulnerabilities. Department of Computer Science State University of New York Stony Brook NY 11794.
23. D. Reed, G. Ziemba, and P. Traina. RFC 1858 Security considerations for IP fragment filtering. Oct 1995.
24. R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. *USENIX Intrusion Detection Workshop*, 1999.
25. Network Flight Recorder Security. `http://www.nfr.com/`.
26. R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. Master's thesis, State University of New York at Stony Brook, NY 11794.
27. Internet Security Systems. Network- vs. host-based intrusion detection. Technical report, `http://www.iss.net/support/documentation/whitepapers/index.php`, Oct 02 1998.
28. Internet Security Systems. Intrusion detection systems - whitepaper. `http://www.iss.net/support/documentation/whitepapers/index.php`, 1999.