Using Data-Independence in the Analysis of Intrusion Detection Systems

Gordon Thomas Rohrmair Gavin Lowe*

June 4, 2004

Abstract

In this paper we demonstrate the modelling and analysis of intrusion detection systems and their environment using the process algebra Communicating Sequential Processes and its model checker FDR. We show that this analysis can be used to discover attack strategies that can be used to blind an intrusion detection system, even a hypothetically perfect one that knows all the weaknesses of its protected host. We give an exhaustive analysis of all such attack possibilities. We discuss how to strengthen the intrusion detection systems to prevent these attacks, and finally we show how we can use data independence techniques to verify the corrected versions.

1 Introduction

This paper introduces the use of the process algebra Communicating Sequential Processes (CSP) [Hoa85, Ros97] within the intrusion detection area. We will show how one can use CSP and its model checker FDR [GGH$^+$00] to analyse the interactions between intrusion detection systems (IDSs) and their environment. Special attention is given to the unexpected side-effects that allow an attacker to render the IDS useless.

An intrusion detection system is used to detect security policy violations 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).

There are techniques, known as *desynchronisation attacks*, that make it possible to blind an intrusion detection system, even a hypothetically perfect one that knows all weaknesses of its protected host. Although these attacks are

*Authors' address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; e-mail {gordon.rohrmair, gavin.lowe}@comlab.ox.ac.uk

easy to understand, it took the security community a long time to discover them [PN98, Pax99].

In this paper we will show how to use CSP to discover such attacks automatically. Briefly, our approach is as follows. We build a CSP model of a small network employing an IDS; we also model the most general attacker who can inject packets into the network (although not interfere with packets once they are within the network). We then use the model checker FDR to search the state space, and to detect attacks upon a target that are not detected by the IDS. We consider two examples, both based upon simplified versions of the Internet Protocol version 4 (IPv4) [dR81]: the first displays an attack based upon the time-to-live field; the second displays an attack based upon the target and IDS treating differently a fragment of a packet that arrives after another fragment with the same offset within the packet.

We show how to adapt the intrusion detection systems to prevent these attacks, and give an exhaustive analysis of all attack possibilities that are based on this class. The model checker fails to find any attacks upon the adapted versions. However, in order to perform this analysis—for example, to keep the state space finite—we have to perform various abstractions, particularly concerning the set of network packets and the set of attack signatures in the model. It is not immediately clear that these abstractions are sound: our failure to find attacks on the adapted models could be caused by an over-abstraction that loses attacks. To show that our abstractions are valid we have to show that any attack can be reduced to one that could be found by our model. We change the focus of our models, and then use data independence techniques to show that our abstractions are sound.

The next section provides background information regarding intrusion detection systems and CSP. Section 3 describes our modelling and analysis of the time-to-live model. In Section 4 we show the generality of our approach, by applying it to a different example, based upon the packet reassembly algorithm of the Internet Protocol. In Section 5, we adapt the focus of the models, so as to take a slightly different view; we then use results from data-independence to show that this analysis is complete, in the sense that it is independent of the underlying types of network packets and attack signatures used in the model. We consider the effects of other abstractions in Section 6. We sum up, and discuss related and future work in Section 7.

2 Background

In this section we describe relevant background information, about both intrusion detection systems and CSP.

2.1 Intrusion detection systems

An intrusion detection system is used to detect security policy violations in a network, caused by either insiders or outsiders. We briefly review the different types of intrusion detection systems, and the different data sources they use.

2.1.1 Classes of intrusion detection techniques

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 [Ken01]; examples of such systems can be found in [Axe00]. 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.

The system knows exactly how a certain attack manifests itself. This leads in theory to a low false-positive ratio [Axe00, DDW00]. The detection algorithm is based on pattern matching, for which efficient solutions exist. However, 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.

We deal only with misuse detection based systems in this paper.

Anomaly detection Anomaly detection based 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 as to whether observed behaviour is normal or anomalous is based on 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 [Axe00].

Specification-based detection Specification-based intrusion detection systems [KFL94, Ko96, SBS99] distinguish 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.

2.1.2 Classes of intrusion detection data sources

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

Network intrusion detection systems A network intrusion detection systems (NIDS) gets its information from a network adapter operating in promiscuous mode. It examines the traffic for an attack signature. Although anomaly detection has been implemented for these systems, the main detection principle is misuse detection. A NIDS can provide surveillance for a whole network, because it is working with the raw network packets. In this paper we model a NIDS, because it is the most used system type [NFR, SNO, Sys98, Axe00].

Due to the fact that a single NIDS can monitor a whole network, its implementation and maintenance costs are low. Additionally these systems, since they work at the packet level, have all the information to detect the difference between hostile and friendly intentions. However, these systems are largely unable to read the traffic of encrypted connections. Further, increasingly networks are switched rather then broadcasted: due to the Ethernet working principle [Bla98], 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 [Sys99].

Host intrusion detection systems A host intrusion detection system 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 [Sys98]. 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.

2.2 CSP

An event represents an atomic communication; this might either be between two processes or between a process and the environment. Channels carry sets of events; for example, $c.5$ is an event of channel c.

The process $STOP$ represents a deadlocked process, that can perform no events. The process $a \rightarrow P$ can perform the event a, and then act like P. The process $c?x \rightarrow P_x$ inputs a value x from channel c and then acts like P_x.

The process $P \sqcap Q$ represents an internal or nondeterministic choice between P and Q; the process can act like either P or Q, with the choice being made according to some criteria that we do not model. The process $\bigsqcap_{i:I} P_i$ represents a replicated nondeterministic choices, indexed over set I.

The process $P \underset{A}{\|} Q$ represents the processes P and Q composed in parallel, synchronising on event from the set A. Note that in larger parallel compositions, several processes may synchronise on the same event; for example, in the process $(P \underset{A}{\|} Q) \underset{B}{\|} R)$, all three processes will synchronise on events from $A \cap B$.

A *trace* is a sequence of events that a process can perform. Process P is trace-refined by process Q if all the traces of Q are also traces of P:

$$P \sqsubseteq_T Q \Leftrightarrow traces(P) \supseteq traces(Q).$$

A *failure* of a process is a pair (tr, X), representing that the process can perform the trace tr to reach a stable state (i.e. where no internal activity is possible), where none of the events from X can be performed. Process P is failures-refined by process Q if all the failures of Q are also failures of P:

$$P \sqsubseteq_F Q \Leftrightarrow failures(P) \supseteq failures(Q).$$

The model checker FDR can be used to test whether a model of a system trace-refines or failures-refines a specification process. Typically, the specification will capture all allowable traces or failures.

3 The time-to-live model

In this section we present our first model. We consider whether a simplified version of the Internet Protocol version 4 (IPv4) [dR81] gives an attacker the opportunity to launch an undetected attack against the target.

We model just two fields of the protocol:

- We model the data field, so as to capture network communication, and possible network-based attacks.
- We also model the time-to-live (TTL) field. The TTL represents the distance a packet can travel; every router decreases its value by one; once zero is reached the packet is discarded. As shown in [PN98], the TTL offers an interesting evasion possibility; our analysis below reveals this attack.

Modelling assumptions We assume that the IDS itself is perfect, in the sense that it knows all signatures corresponding to vulnerabilities that could be used to cause a security breach. Of course, representing attacks by signatures is unsatisfactory: it would be better to talk about the effects on the target itself; however, signatures are a de-facto standard, and it seems very difficult to do better.

Additionally we consider only one-way, in-order communication.

The network topology will be as in Figure 1. We model a network with just one receiver node, or *target*. We use a *demilitarised zone configuration* [CZC00], which consists of an exterior filtering router and an internal filtering router; 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 demilitarised zone resides between these two routers; this is the place where companies maintain their public servers, such as the web

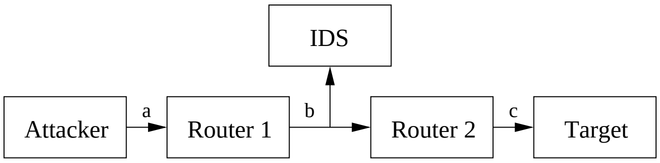

Figure 1: The network topology

server. It is also the preferred place for the IDS: it is the only place where the IDS can receive all the traffic that originates outside the local network. The use of a demilitarised zone is not necessary to discover the attack below: many other topologies would reveal the same attack; we consider a demilitarised zone because they are commonly deployed in industry.

Note, that the IDS monitors communications passing between the two routers. We will model this as a three-way synchronisation between the corresponding processes.

3.1 The CSP model

Each channel carries some data and a TTL value. In order to keep the state space of our model finite, we need to take fixed finite ranges for these types.

- The data field will range over the type $DATA = \{A, B, C\}$.
- The TTL value will only range from 1 to 3; we believe that this range for the TTL is enough, because the diameter of the resulting network will be smaller then 3.

Further, we need to model a particular set of attack signatures:

- We will take the set $sigs$ of attack signatures to be $\{\langle A, B\rangle\}$.

For modelling 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 main purpose of the routers is for navigating packets from source to destination; however, we will completely ignore this routing functionality, and concentrate on their effect on the TTL field; we therefore treat them as relay-stations. The router receives a packet, and decreases the TTL value by one; if the resulting value is zero, the packet is dropped; otherwise, the packet is forwarded. From this we get the following CSP description:

$$
\begin{array}{l}
Router(in, out) = \\
\quad in?x?y \rightarrow \\
\quad \mathrm{let}\, y' = y - 1 \\
\quad \mathrm{within\, if}\ y' > 0\ \mathrm{then}\ out.x.y' \rightarrow Router(in, out).
\end{array}
$$

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 all the actions a real-world attacker could perform. We consider here only an *external* attacker, who can inject packets into the network, but not delete or modify packets once they are within the network; further, we assume that the attacker can not compromise the components of the system.

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.

$$Attacker \;=\; \bigsqcap_{x:DATA, y:TTL} a.x.y \rightarrow Attacker \sqcap STOP.$$

The target The target process receives packets. If a sequence of packets corresponding to an attack signature is received, then this corresponds to a violation of the security policy, which we model by the event *fail*. The following CSP description describes this process.

$$\begin{array}{l}
Target(sigs, vulnerabilities) = \\
\quad c?x?y \rightarrow \\
\quad \text{let } vulnerabilities' = \{s \mid \langle x \rangle ^\frown s \in sigs \cup vulnerabilities\} \\
\quad \text{within if } \langle x \rangle \in vulnerabilities \\
\qquad\quad \text{then } fail \rightarrow Target(sigs, vulnerabilities') \\
\qquad\quad \text{else } Target(sigs, vulnerabilities').
\end{array}$$

This process has two parameters, *sigs* and *vulnerabilities*. *sigs* is the set of a set of all attack signatures. *vulnerabilities* keeps track of the progress of security breaches, and indicates what the target has to receive in order to fail; the set comprehension is used to update *vulnerabilities*.

The IDS The IDS monitors the traffic that travels through the network. We assume here that the IDS is a perfect signature based IDS and therefore knows all vulnerabilities that cause the target to fail. The following CSP description for the IDS differs from the above target component by engaging in an *alert* event rather than in a *fail* event once it has received an attack pattern.

$$\begin{array}{l}
IDS(sigs, alerts) = \\
\quad b?x?y \rightarrow \\
\quad \text{let } alerts' = \{s \mid \langle x \rangle ^\frown s \in sigs \cup alerts\} \\
\quad \text{within if } \langle x \rangle \in alerts \text{ then } alert \rightarrow IDS(sigs, alerts') \\
\qquad\quad \text{else } IDS(sigs, alerts').
\end{array}$$

The complete system We combine these processes in parallel, as depicted in Figure 1, and then hide all events other than *fail* and *alert* (i.e. turn them into internal events). Let *System* be the resulting process.

The specification The specification expresses that for every *fail* event, there should be a corresponding *alert* (and vice versa). In other words, the IDS should have a log-entry for each successful attack. However, because of the asynchronous nature of the system, the *fail* might precede the *alert*, but in this case the *alert* must subsequently become available (i.e. cannot be refused). We can model this with a simple recursive CSP process.

$$Spec \;=\; alert \rightarrow fail \rightarrow Spec \sqcap fail \rightarrow alert \rightarrow Spec \sqcap STOP.$$

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

3.2 Results

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

$$\langle a.A.4,\ b.A.3,\ c.A.3,\ a.C.2,\ b.C.1,\ d.A.2,\\ c.C.1,\ a.B.4,\ b.B.3,\ c.B.3,\ d.B.2,\ fail\rangle$$

after which the system refuses *alert*. That is, a successful attack has been launched by the attacker, without the IDS detecting it. This trace is displayed in the sequence diagram in Figure 2. This is similar to the attack noted in [PN98].

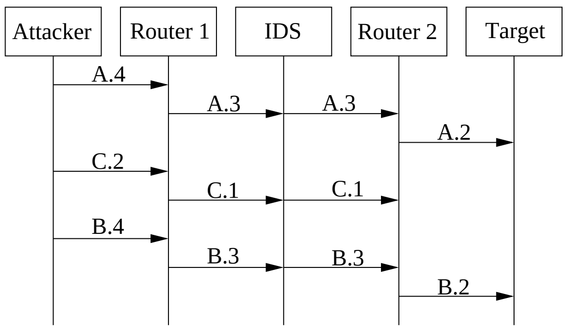

Figure 2: TTL Attack

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 by the last router, so the target receives the sequence $\langle A, B\rangle$, corresponding to an attack signature. The IDS, however, takes the C fragment into account, so receives the sequence $\langle A, C, B\rangle$, which does not correspond to an attack signature. Hence the target fails, but the IDS does not raise an alert

The attack, as presented, supposes that the packets are accepted in the order in which they are received, i.e. they do not have sequence numbers, as in TCP. However, it is straightforward to adapt the model to include sequence numbers, and a similar attack would be found.

Attacks like these, where the states of the IDS and target become desynchronised, are called *desynchronisation attacks*.

The attack presented above uses the fact that the IDS does not consider information about the topology of the network. The obvious way to solve this is to adapt the model so that the IDS only considers packets with a TTL high enough to reach the target. When we make this change, FDR finds that, *for this model*, the IDS detects all attacks. However, it is not clear whether the same would be true if we consider different instantiations of the sets $DATA$ and $sigs$, or whether we over-abstracted; this is the question we consider in Section 5.

4 The packet reassembly model

In this section we consider another example, so as to demonstrate the general nature of our technique. We will examine the behaviour of a system where data packets are fragmented and reassembled based upon RFC 791 [dR81].

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 [dR81] collects the fragments and puts them into the right place in 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 [dR81] leaves unspecified which should be preferred, but the recommendation is to prefer new data, so that if the algorithm receives data for the same position twice, the new data will overwrite the old. However, not all implementations follow this suggestion. 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 [PN98].

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 reassemble the fragment stream, namely:

- the more fragments (MF) bit, which indicates whether this is the last fragment in the reassembled packet (although it might not be the last to arrive);
- the fragment offset (FO) bit, which indicates the offset of this fragment within the packet;
- the time-to-live field, as in the previous model;
- the data field, as in the previous model.

Therefore, event $a.1.0.3.A$ 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 three, and a data field containing 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 reassemble an out-of-order fragment stream, as described in RFC 791.

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. For the purposes of this model, we take the reassembly buffer to be of size five, which is sufficient to discover the attack below; we model an unfilled entry in the buffer by the null value N; hence the reassembly buffer is initialised to $\mathit{emptyBuff} = \langle N, N, N, N, N \rangle$.

$$\begin{aligned} Target(\mathit{sigs}) &= \textstyle\bigsqcap_{os:OS} \mathit{os\_target.os} \rightarrow Target(\mathit{os}, \mathit{emptyBuff}, \mathit{sigs}, 0), \\ \mathit{emptyBuff} &= \langle N, N, N, N, N \rangle. \end{aligned}$$

The process $Target(\mathit{os}, \mathit{buff}, \mathit{sigs}, \mathit{max})$ uses the following parameters: os indicates whether the operating system favours old or new data (with old represented by 0, and new by 1); buff represents the reassembly buffer; sigs represents the set of attack signatures; and max represents the maximum size of the original packet. The target first receives a datagram and calculates the new buffer using the function *overwrite*.

$$\begin{array}{l} Target(os, buff, sigs, max) = \\ \quad in?mf?fo?ttl?data \rightarrow \\ \quad Target'(os, buff, sigs, max, mf, fo, data, overwrite(buff, fo, data)). \end{array}$$

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[1], and if so, if its update policy favours old data, this fragment is ignored. Otherwise, it checks whether the packet is complete[2]. If the packet is complete, it tests whether it has received an attack or not[3], failing if so; in either case the buffer is reinitialised to an empty buffer. If the packet is not complete it stores the fragment offset as the new maximum size of the packet.

$$\begin{array}{l} Target'(os, buff, sigs, max, 0, fo, data, b1) = \\ \quad \text{if } nth(buff, fo) \neq N \wedge os = 0 \text{ then } Target(os, buff, sigs, max) \\ \quad \text{else if } allFilled(b1, fo) \\ \quad\quad \text{then if } check(b1, sigs, fo) \text{ then } fail \rightarrow Target(os, emptyBuff, sigs, 0) \\ \quad\quad\quad \text{else } Target(os, emptyBuff, sigs, 0) \\ \quad\quad \text{else } Target(os, b1, sigs, fo). \end{array}$$

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

$$\begin{array}{l} Target'(os, buff, sigs, max, 1, fo, data, b1) = \\ \quad \text{if } nth(buff, fo) \neq N \wedge os = 0 \text{ then } Target(os, buff, sigs, max) \\ \quad \text{else if } allFilled(b1, max) \wedge max \neq 0 \\ \quad\quad \text{then if } check(b1, sigs, max) \text{ then } fail \rightarrow Target(os, emptyBuff, sigs, 0) \\ \quad\quad\quad \text{else } Target(os, emptyBuff, sigs, 0) \\ \quad\quad \text{else } Target(os, b1, sigs, max). \end{array}$$

The IDS The IDS process is capable of reassembling fragments arriving out-of-order, similar to the target process presented above. It also takes the time-to-live field into account, as in the adapted TTL model. 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*.

[1] The function $nth(a, b)$ returns the value stored in position b of buffer a.

[2] $allFilled(a, b)$ checks whether all data in buffer a up to position b has been received.

[3] $check(a, b, c)$ compares the buffer a with the set b to depth c to test for the existence of any attack patterns in the buffer.

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 discovers two distinct attacks that could both elude the IDS.

Attack 1 The first attack is illustrated in Figure 3. The IDS chooses to use

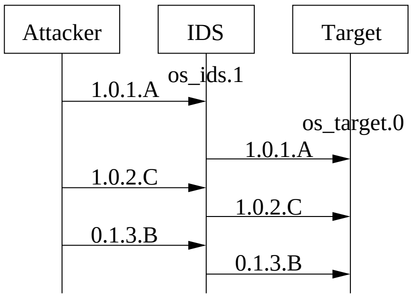

Figure 3: Attack 1

an operating system that favours new data (indicated by the event $os\_ids.1$), whereas the target chooses to favour old data ($os\_target.0$). 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, 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 reassemble their packets. The IDS reassembles $\langle C, B\rangle$ and the target reassembles $\langle A, B\rangle$, which causes a *fail* event without an $alert$.

Attack 2 The second attack is illustrated in Figure 4. This attack is the opposite of the former. The IDS chooses to be based on a type zero operating system (favouring old data) and the target chooses to be based on a type one operating system (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

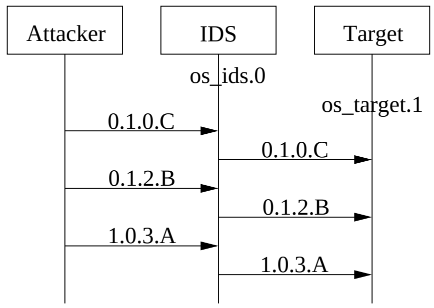

Figure 4: Attack 2

target buffer: the IDS stores a C, whereas the target overwrites the C with a B. The attacker then sends the final packet ($1.0.3.A$). The IDS reassembles $\langle A, C\rangle$, which is innocent, and the target reassembles $\langle A, B\rangle$, which leads to a *fail* event without an *alert*.

4.3 Discussion

Assuming the IDS cannot be sure of the operating system used by the target, it appears that the IDS needs to take account of both possibilities for the target, i.e. favouring either old or new data. When we do this, FDR finds no attacks on our revised version. We will see in section 5 that this result still holds for more general model parameters.

However, this general strategy for avoiding desynchronisation attacks does not appear to scale well to a more general setting. 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 where the packet is completely rejected—described in [PN98], there are a vast number of desynchronisation possibilities. The consequent state-space explosion in the IDS would appear unmanageable.

Whenever it is possible to create a difference between the input stream of the IDS and the protected system, an attack can be successfully hidden. 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 therefore distinguish between three desynchronising possibilities:

1. Desynchronisation due to the systems behaving exactly the same, but the input streams being different; this is the type of flaw exploited in our first model.

2. Desynchronisation 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. Desynchronisation because both the input streams and the behaviour of the systems are different.

5 Towards a more complete analysis

As stated above, our analysis in the preceding sections is not complete, since we do not know whether the inability to spot attacks in the improved models stems from an over-abstraction or from the case that there is really no attack. In this section, we generalise our results, so that we are able to verify the model for *all* values of the type of network packets and the set of attack signatures.

We begin by describing the relevant parts of the theory of data independence. We then show how to generalise the network packet and attack signatures parameters for the time-to-live model. Finally we show that precisely the same technique works for the packet reassembly model.

5.1 Data independence

Often the behaviour of a CSP process is dependent on various parameters, such as the underlying types used in events, or the number of nodes in a network. However, analysis using explicit model checking, for example with FDR, can consider only a single value for each parameter at a time. We would like to be able to verify a system for all values of the parameters via a small number of explicit checks; this is often known as the *Parameterised Verification Problem*.

Data independence is a tool that can be used to address this problem. The goal of this approach is so come up with a bound N such that if a refinement holds when a type parameter is instantiated with a type of size N, then the refinement also holds for all larger types.

A process P is said to be data independent with respect to a type T if the only operations it performs on values of T are to input them, store them, and output them; in particular, P never performs any computation on values of T that constrains what T might be.

A process satisfies the condition NoEq_T if it performs no test of equality between members of T: such tests could be explicit, for example within 'if-then-else' constructs, or implicit via synchronisations between parallel components on single elements of T. A process satisfies Norm_T if, essentially, it contains no nondeterminism the effects of which are not immediately apparent (see [Laz97, Ros97] for a formal definition).

The following theorem is from [Laz97, Ros97]:

Theorem 1 Suppose that $Spec$ and $Impl$ are data independent processes, that both satisfy NoEq_T, and that $Spec$ satisfies Norm_T. If $Spec \sqsubseteq Impl$ when T is taken to be of size 2, then $Spec \sqsubseteq Impl$ for all finite or infinite values of T of size at least 1.

5.2 Generalising the types of packets and signatures

We now change the focus of the time-to-live model from Section 3, in order to move towards a more complete analysis, independent of the set *sigs* of attack signatures. We observe that the IDS is really doing two things: filtering out packets that will not reach the target; and performing pattern matching on the remaining packets. It is therefore possible to split the IDS into two different processes corresponding to these functions. In related models, such as the packet reassembly model of Section 4, the target process similarly performs a combination of filtering and pattern matching, and so it is possible to split the target into two processes. This gives a topology as in Figure 5.

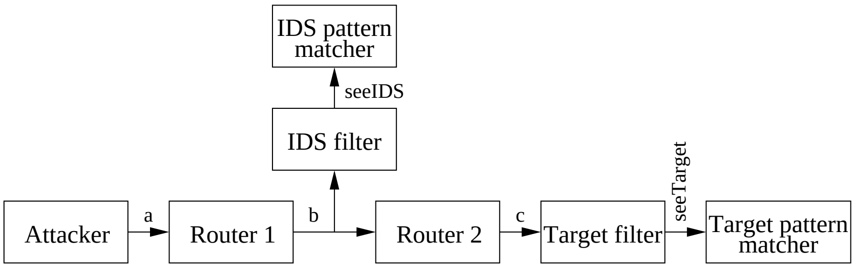

Figure 5: Network topology for the revised time-to-live model

We can then make the following observation:

Observation 1 If the stream of packets passed to the pattern matching component of the IDS is the same as the stream of packets passed to the pattern matching component of the target, then the IDS will detect all attacks.

The hypothesis of Observation 1—that the two components see the same stream of packets—is an easy property to test. In fact it does not even require us to model the two pattern matching components, simply the messages passed to them on the channels $seeIDS$ and $seeTarget$. The advantage of this change of focus is that is allows us to remove the parameter *sigs* from the model, thus leading towards a more general verification.

The two filtering processes (where the IDS takes the distance to the target into account) can be modelled by:

$$\begin{aligned} \mathit{IDS} &= b?x?y \rightarrow \text{if } \mathit{dist} \leq y \text{ then } \mathit{seeIDS}.x \rightarrow \mathit{IDS} \text{ else } \mathit{IDS}, \\ Target &= c?x?y \rightarrow seeTarget.x \rightarrow Target. \end{aligned}$$

where *dist* is the distance from the IDS to the target. The rest of the network is unchanged.

It is easy to capture the hypothesis of Observation 1 as a refinement assertion; there is a certain amount of buffering in the system, and the specification has to take this into account:

$$
\begin{aligned}
Spec &= \bigsqcap_{x:T} (seeIDS.x \to Spec'(x) \sqcap seeTarget.x \to seeIDS.x \to Spec) \\
&\quad \sqcap \\
&\quad STOP, \\
Spec'(x) &= seeTarget.x \to Spec \\
&\quad \sqcap \\
&\quad \bigsqcap_{y:T} seeIDS.y \to seeTarget.x \to Spec'(y).
\end{aligned}
$$

The specification captures the property that the IDS and target see the same stream of packets, except that the IDS might at any point have seen up to two more packets than the target, or the target might have seen one more packet that the IDS.

We can then use FDR to check that the system failures-refines *Spec*. The refinement holds, and we can then use Observation 1 to deduce that the IDS detects all attacks *for all possible sets of attack signatures*.

However, this appears to leave us only slightly better off than before: we can verify the system for a fixed type *DATA*; but does this tell us anything about systems with different values for *DATA*? It turns out that we can use Theorem 1 to show that this is indeed the case. The system and specification processes are both data independent with respect to the type *DATA* (when the pattern matching against attack signatures was included, the IDS and target were not data independent); both satisfy NoEq_{DATA}; and the specification satisfies Norm_{DATA}. The theorem therefore tells us that we have only to check the refinement for a type *DATA* of size 2, say $DATA = \{A, B\}$, to have verified it for all values of *DATA*.

5.3 The packet reassembly model

The same technique can be used for our second example, from Section 4. We can model the reassembly algorithms, at both the target and IDS, as separate processes. We can then ask whether the stream of packets passed to the pattern-matching components are the same, using essentially the same specification as for the time-to-live model; this generalises our analysis to all possible sets of attack signatures. We can then use Theorem 1 to show that we need only perform this analysis for a type *DATA* of size 2 to deduce that the same result holds for all larger types.

6 Remaining points

In this section we generalise the parameters of the system further. We show that the range $\{1\,..\,3\}$ for the TTL field is sufficient to capture all essentially-distinct behaviours of the system for arbitrary TTL values. We further show that a buffer size of five (in the buffer-reassembly model) is sufficient. We also discuss why the network topology is not as restricted as it appears to be.

6.1 The TTL-value range

Consider the network in Figure 1. It is obvious that the diameter of the network, counted in TTL-decreasing hops, is two. Hence there appears to be no need to consider TTL values greater than three. We formalise this argument in this section.

Let $System(N)$ be the CSP process representing a system using TTL values $\{1\,..\,N\}$, and let $System_0(N)$ be the corresponding process before internal events are hidden, i.e. such that all channels are visible.

The technique we use is to define a collapsing function ϕ over events, that maps behaviour of a system using arbitrary-sized TTL values to corresponding behaviours of the small system[4]:

$$\begin{aligned} \forall\, tr : \Sigma^* \,;\, X : \mathbf{P}\,\Sigma \bullet (tr, \phi^{-1}(X)) \in failures(System_0(N)) \Rightarrow \\ (\phi(tr), X) \in failures(System_0(3)). \end{aligned} \tag{1}$$

We define ϕ over events, to replace TTL values greater than 3 on the channel a by 3, and similarly for other channels:

$$\begin{aligned} \phi(a.x.y) &= \text{if } y \leq 3 \text{ then } a.x.y \text{ else } a.x.3, \\ \phi(b.x.y) &= \text{if } y \leq 2 \text{ then } b.x.y \text{ else } b.x.2, \\ \phi(c.x.y) &= \text{if } y \leq 1 \text{ then } c.x.y \text{ else } c.x.1, \end{aligned}$$

and ϕ is the identity function over all other events. We lift ϕ to traces, point-wise.

It is then straightforward to see that each component of the system respects ϕ—i.e. such that, analogously to equation (1), if $(tr, \phi^{-1}(X))$ is a behaviour of the component with N TTL values, then $(\phi(tr), X)$ is a behaviour of the component with 3 TTL values. Hence equation (1) itself is satisfied. However, within $System$, the channels a, b and c are hidden, so ϕ is the identity function over the visible events, and we deduce

$$failures(System(N)) \;=\; failures(System(3)).$$

But we already know that $Spec \sqsubseteq_F System(3)$; hence we can deduce that $Spec \sqsubseteq_F System(N)$, for all N.

[4] Σ represents the set of all events; $\phi(tr)$ represents ϕ applied point-wise to trace tr; $\phi^{-1}(X)$ represents the inverse image of X under ϕ.

6.2 Buffer size

We now argue that in the reassembly model it is sufficient to consider a buffer size of five, in the sense that if there is an attack upon a system that uses a larger buffer, then there is also an attack upon a system that uses a buffer of size five.

Suppose there is an attack trace tr in the general case. We map this onto an attack trace tr' in the restricted case of buffer size five; we do this by uniformly remapping fragment offsets onto the set $\{0 \mathinner{..} 4\}$. The construction is slightly complicated by the fact that we need to ensure that the target and IDS reassemble packets at the same points in tr' as they did in tr.

Suppose, firstly, that the attack trace tr is such that the IDS and target reassemble the initial packet differently. We arrange that in tr' they again reassemble the initial packet differently. We perform a case analysis.

Case 1 The IDS and target reassemble packets of different sizes, m and n respectively. Without loss of generality suppose $m < n$. Suppose the last fragments of those packets to be received by the IDS and target (i.e. last according to the order of the trace, as opposed to necessarily the fragments with the greatest offsets) are at offsets x and y respectively. We perform a further case analysis:

- Case x, y, m, n all distinct, with $x, y < m < n$. We define the remapping function ϕ over fragment offsets as follows:

$$\begin{aligned}\phi(i) \;=\; & \text{if } i = x \text{ then } 1 \text{ else if } i = y \text{ then } 2 \\ & \text{else if } i = m \text{ then } 3 \text{ else if } i = n \text{ then } 4 \text{ else } 0.\end{aligned}$$

  We then lift ϕ to events by, for example

$$\phi(a.mf.fo.ttl.data) \;=\; a.mf.\phi(fo).ttl.data.$$

  Consider the effect of applying ϕ point-wise to all communications on channels other than $seeIDS$ and $seeTarget$. It is then easy to see that the routers and the attacker respect ϕ. It is also easy to see that the IDS and target reassemble different packets: the IDS reassembles a packet of size 4 when it receives the fragment with offset 1; and the target reassembles a packet of size 5 when it receives the fragment with offset 2.

- Case otherwise. One can construct a similar remapping function in each case, mapping onto a proper subset of $\{0 \mathinner{..} 4\}$ if some of x, y, m, n coincide.

Case 2 The IDS and target reassemble packets of the same size, n say, but that differ at some point, say at offset z. Again suppose the last fragments of those packets to be received by the IDS and target are at offsets x and y respectively.

- Case x, y, z, n all distinct. We define the remapping function ϕ over fragment offsets as follows:

$$\begin{aligned}\phi(i) \;=\; & \text{if } i = x \text{ then } 1 \text{ else if } i = y \text{ then } 2 \\ & \text{else if } i = z \text{ then } 3 \text{ else if } i = n \text{ then } 4 \text{ else } 0.\end{aligned}$$

  We lift ϕ to events as above. Then it is easy to see that the IDS and target both reassemble packets of size 4, after receiving the fragments with offsets 1 and 2 respectively. However, these packets differ at offset 3: the two components receive the same sequence of data packets at this offset as they received at offset z in the general case; and so they will, as in the general case, reassemble packets that differ at this offset.

- Case otherwise. All other cases are similar.

Suppose now that the IDS and target reassemble a packet other than the first differently, say the nth packet. Then the above construction will either cause the IDS and target to reassemble a packet earlier than the nth differently (introducing a new undetected attack); or the construction will leave the first $n-1$ packets matching, but again cause them to construct different nth packets.

Finally the case where one agent reassembles a packet and the other doesn't can be treated in a similar manner.

6.3 Network Topology

We have considered only a single, fixed, network topology. This turns out to be an important consideration. If there are two or more routes between the IDS and the target, then it is possible for the IDS and target to see fragments in a different order, which opens up the possibility of another desynchronisation attack. However, most local networks have only a single route from the gateway to each host, which prevents such attacks. Under this condition, we believe that our abstraction has not lost any attacks:

- It is safe to abstract away from the parts of the network before the IDS, because the attacker can effectively choose what fragments are sent past the IDS (channel b in the earlier models).

- It is also safe to abstract away those parts of the local network not on the direct route from the IDS to the target, because they do not affect what the target sees.

- The remaining issue is the number of routers between the IDS and the target; there does not seem to be any intrinsic difference between one router reducing the TTL field by one, and N routers reducing the TTL by N, because the attacker is able to choose the TTL appropriately; however, the number of routers does affect the buffering of the system and hence the appropriate specification process.

6.4 Protocol abstractions

Finally, we have, of course, abstracted away from many of the details of the underlying network protocol, such as length of packets, proper routing functionality, or packets being dropped because of congestion. Our long-term aim is to remove these abstractions, so as to model the whole of a real network protocol, such as the Internet Protocol.

7 Conclusion

In this paper we have demonstrated how one can automatically detect desynchronisation attacks upon intrusion detection systems. We modelled a small network deploying an intrusion detection system, together with a completely general attacker, and used a model checker to explore the state space to discover attacks that went undetected by the IDS.

We then described how to perform a more complete analysis, so as to show the absence of such attacks. We changed the focus of our previous models so as to ask whether the IDS and target, after appropriate filtering, see the same stream of packets; this makes our analysis independent of the set of attack signatures. We then used a result from data independence to show that the analysis is independent of the underlying type of data. Finally, we argued that our analysis was independent of the range of time-to-live values, the size of the reassembly buffer, and—to a certain extent—the network topology.

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 still not be free from flaws. Such emergent faults can be spotted easily by testing the system as a whole against a specification.

It is worth noting that the converse of Observation 1 is not, in general, true: if the attacker can cause the target and IDS to see different streams of packets, this does not necessarily mean that a desynchronisation attack is possible. However, we believe that the converse of Observation 1 is true in all but contrived examples. Even if this is not the case, the worst that will happen is that we detect a false attack, which will give us a better understanding of the system.

There is a further abstraction that we have not addressed, 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 desynchronisation attack. Consider the case where 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 desynchronised.

This attack reveals an interesting point: it is very hard to be certain in analyses such as these that abstractions have not removed details that allow attacks. Because of this we need a proper framework to prove formally that abstractions have not lost essential details.

We intend to extend our analyses, so as to model more aspects of the Internet Protocol, and other network protocols, with the aim of detecting further desynchronisation attacks.

We close by briefly discussing some other approaches to automatic vulnerability identification and analysis.

One of the first attempts to build an automatic vulnerability detector was COPS [FS90]. It fires known attack patterns against a particular host. However, this kind of direct testing is not suitable for spotting unknown attacks.

In [RA00], Ritchey and Ammann propose a high level approach to detect whether it is possible for an attacker to leverage 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, [RS02] 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 more. The differences between that approach and ours lies in the state space: they use an infinite model, whereas we use abstraction and data independence to restrict our model to a finite one. This allows to use a common model checker such as FDR. Also, 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 [AB00, ADX00]. However, they use a model checker combined with mutation testing techniques to generate and recognise test sets rather then testing directly.

The advantages of our approach are:

- The approach finds all possible attacks, modulo the abstractions of the model. Although the attacks detected in this paper were known ones, there is no reason why new, novel, attacks could not be detected using this technique.

- It generates easily understood 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.
- We can deal with unbounded models by using data independence techniques.

Acknowledgements

We would like to thank members of the Security Research Group at Oxford University, and anonymous referees of earlier versions of this paper, for useful comments and suggestions.

References

[AB00] P. Ammann and P. Black. Test generation and recognition with formal methods. In *First International Workshop on Automated Program Analysis, Testing, and Verication (WAPATV'00)*, pages 64–67, 2000.

[ADX00] Paul Ammann, Wei Ding, and Daling Xu. Using a model checker to test safety properties. Technical report, ISE Department, George Mason University, 2000.

[Axe00] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers University, Mar 2000.

[Bla98] Uyless Black. *TCP/IP and Related Protocols.* Computer Communications. McGraw-Hill, 1998.

[CZC00] D. Brent Chapman, Elizabeth D. Zwicky, and Simon Cooper. *Building Internet Firewalls.* O'Reilly, 2000.

[DDW00] M. Dacier H. Debar and A. Wespi. A revised taxonomy for intrusion detection systems. Technical Report 55(7–8):361–378, Annales des Telecommunications, 2000.

[dR81] Marina del Rey. *RFC 791 Internet Protocol: DARPA Internet program protocol specification*, 1981.

[FS90] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *USENIX Summer*, pages 165–170, 1990.

[GGH+00] Paul Gardiner, Michael Goldsmith, Jason Hulance, David Jackson, A.W. Roscoe, and Bryan Scattergood. *FDR2 User Manual*. Formal Systems (Europe) Ltd., 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Sys99] Internet Security Systems. Intrusion detection systems - whitepaper. Technical report, ISS, 1999.

[Ken01] Frederick Karen Kent. Network intrusion detection signatures — part 1. `http://www.securityfocus.com`, 2001.

[KFL94] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. Technical report, Department of Computer Science, University of California at Davis, 1994.

[Ko96] 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.

[Laz97] Ranko Lazić. *A Semantic Study of Data-Independence with Applications to the Mechanical Verification of Concurrent Systems*. D.Phil., Oxford University, 1997.

[NFR] Network flight recorder security. `http://www.nfr.com/`.

[Pax99] Vern Paxton. BRO: A system for detecting network intruders in real-time. *Computer Networks*, 31:2435–2463, 1999.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. *Secure Networks*, 1998.

[RA00] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Oakland Symposium on Security and Privacy*, 2000.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[RS02] C.R. Ramakrishan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1, 2), 2002.

[SBS99] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *USENIX Intrusion Detection Workshop*, 1999.

[SNO] SNORT. `http://www.snort.org/`.

[Sys98] Internet Security Systems. Network- versus host-based intrusion detection. `http://www.iss.net/support/documentation/whitepapers/index.php`, 1998.