Midpoints Versus Endpoints: From Protocols to Firewalls*

Diana von Bidder-Senn[1], David Basin[1], and Germano Caronni[2]

[1] ETH Zürich, 8092 Zürich, Switzerland
[2] Google Inc., Mountain View
diana.bidder@inf.ethz.ch, basin@inf.ethz.ch, gec@acm.org

Abstract. Today's protocol specifications only define the behaviour of principals representing communication endpoints. But in addition to endpoints, networks contain midpoints, which are machines that observe or filter traffic between endpoints. In this paper, we explain why midpoints should handle protocols differently from endpoints and thus midpoint specifications are needed. With a case study, using the TCP protocol and three different firewalls as midpoints, we illustrate the consequences of the current lack of protocol specifications for midpoints, namely that the same protocol is implemented differently by the different firewalls. We then propose a solution to the problem: We give an algorithm that generates a midpoint automaton from specifications of endpoint automata. We prove that the resulting midpoint automata are correct in that they forward only those messages that could have resulted from protocol-conform endpoints. Finally, we illustrate the algorithm on the TCP protocol.

1 Introduction

Networks contain different kinds of principals. Some are communication endpoints, such as clients and servers, while others are midpoints (also called *middleboxes* [Gro02a]) that forward, filter, or, more generally, transform traffic. A midpoint that simply forwards traffic is straightforward to implement. But as soon as stateful filtering comes into play, the midpoint must know the communication protocols used. This is TCP for packet filters and diverse application-level protocols for application-level firewalls. If a midpoint does not know enough about the protocols it filters, there exist ways to bypass a security policy. A prominent example is sending file-sharing traffic over http when using packet filters.

Protocol specifications are normally written for endpoints. Starting from such specifications, it is not clear how a midpoint should enforce the protocol-conform execution by the endpoints, as it can neither observe nor correctly track the protocol states of endpoints (see Section 3 for more details on this problem). Another problem is that filtering midpoints need to be as secure (i.e. as strict)

* This work was partially supported by armasuisse. It represents the views of the authors.

J. Katz and M. Yung (Eds.): ACNS 2007, LNCS 4521, pp. 46–64, 2007.
© Springer-Verlag Berlin Heidelberg 2007

as possible. However, they should also be user-friendly (and therefore not overly strict). This leads to different interpretations on how a midpoint should handle a given protocol.

The implications of the lack of protocol specifications for midpoints are that manufacturers of devices acting as midpoints have no guidelines on how they should implement a protocol. In practice, midpoint manufacturers implement the same protocol differently, based on their own interpretation of how the midpoint should handle the endpoint data. This implementation is then incrementally adapted based on practical experience. To show how this looks in practice, we present the TCP automata of three different firewalls in Section 4 and analyse their differences.

As a solution to this problem, we show how to systematically generate midpoint specifications from endpoint specifications. We propose an algorithm that, given the protocol automata for the endpoints, generates a protocol automaton for the midpoint. Roughly speaking, the algorithm tracks all possible endpoint states at each point in time, taking into account messages in transit and possible network behaviour. We prove that the midpoint automata constructed forward only those messages that could have resulted from protocol-conform endpoints. Overall, our contributions are an analysis of why different protocol specifications are needed for midpoints than for endpoints, what the implications of the lack of such specifications are, and a solution for this problem. We use the TCP protocol as an example.

The remainder of this paper is organised as follows. In Section 2, we briefly describe background and related work. We then, in Section 3, explain why midpoints are different from endpoints and therefore need their own protocol specifications, before presenting, in Section 4, the results of our case study. In Section 5, we present an algorithm to generate a midpoint automaton from endpoint automata and prove it correct. Finally we conclude and report on future work in Section 6.

2 Background and Related Work

We will use the TCP protocol [ISI81] for our examples, which we briefly summarise here. TCP is a connection-oriented protocol, which is used by applications to transport data to communication partners in a reliable way. TCP itself uses IP to transport the data and just adds a header for flow control, reliability, and multiplexing purposes. As TCP is connection-oriented, there is an initiation — called a *three-way-handshake* (see Figure 1) — and a tear down (see Figure 2) for each connection. With the use of sequence numbers and acknowledgements, TCP ensures that (a copy of) every packet reaches its destination.

For protocol specifications, we use Mealy machines (automata) [Mea55]. A Mealy machine is a six-tuple $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_1)$, where $Q = \{q_1, q_2, ..., q_{|Q|}\}$ is a finite set of *states*; $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_{|\Sigma|}\}$ is a finite *input alphabet*;

Alice		message sent		Bob
CLOSED				LISTEN
SYN-SENT	—	SYN	→	
	←	SYN & ACK	—	SYN-RECEIVED
ESTABLISHED	—	ACK	→	ESTABLISHED

Fig. 1. TCP three-way-handshake

Alice	message sent	Bob
	— FIN & ACK →	
	← FIN & ACK —	
	— ACK →	

Fig. 2. TCP connection tear down

$\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_{|\Gamma|}\}$ is a finite *output alphabet*; $\delta : Q \times \Sigma \to Q$ is the *transition function*; $\lambda : Q \times \Sigma \to \Gamma$ is the *output function*; and $q_1 \in Q$ is the *initial state.* Note that we write $q_i \xrightarrow{x/y} q_j$ to denote that $\delta(q_i, x) = q_j$ and $\lambda(q_i, x) = y$.

Despite the fact that midpoint automata are central for the construction of firewalls and other security gateways, the problem we address has only partly been identified before [Gro00, Gro02b]. To our knowledge, we are the first to present a formal treatment of the problem and to provide an approach for solving it. The closest related work is [BCMG01], which describes how to build a monitor to find out if a system correctly implements an endpoint specification. The authors report on the same problems as ours in determining the state of an endpoint. This problem arises as packets can be reordered or lost between the monitor and the endpoint. They propose several monitoring algorithms. Unfortunately, their algorithm that takes arbitrary reordering and loss into account is very inefficient (the authors call it brute-force) and their refinements are too constrained to be useful in our setting. In principle, their solution could be used to solve our problem, by using one monitor per endpoint and attaching the output of one monitor to the input of the other. As their monitors are inefficient, this is not a practical solution since firewalls should execute very efficiently, i.e., make each decision with minimal overhead.[1] Note that we also solve a different problem than they do: We do not care if the endpoints correctly implement a protocol. Our goal is to only let messages arising from correct protocol runs pass the firewall, independently of how the endpoints create them.

Related areas are firewall testing [MWZ05, ASH03], test-case generation for Mealy machines [Gil61, Cho78, SD88, FvBK+91, CVI89] and the testing of TCP endpoint automata [BFN+06, Pax97]. In firewall testing a midpoint is tested. The starting point of previous work has been the firewall rulesets but not the underlying automata. The area of test case generation for Mealy machines is

[1] This is not a problem with our generated automata. The generation takes time but their execution is very fast.

related, as these methods can be used to generate test cases for the midpoint automata resulting from our algorithm.

3 The Source of the Problem

In this section, we explain why midpoints are different from endpoints and why they thus need a different protocol specification. The problem arises with the filtering midpoints. These base their decisions — basically drop or forward — on the protocol states the endpoints are in. Unfortunately the two endpoints of a connection can be in different states and not all of these states are observable by the midpoint.

Consider the following example: the TCP connection initiation (three-way-handshake) shown in Figure 1. Imagine the second packet gets lost after being forwarded by the midpoint (scenario 1). Alice is now in the state `SYN-SENT`, whereas Bob is in state `SYN-RECEIVED`. To the midpoint, this situation looks the same as the situation where the second packet reaches Alice, but the third packet gets lost before being received by the midpoint (scenario 2).

Given that the midpoint cannot differentiate between these scenarios, in what state should the midpoint be? And how should it react upon receiving a SYN packet from Alice? In scenario 1, the SYN is a retransmission and should be forwarded. Whereas in scenario 2, a SYN does not conform with a correct protocol execution (Alice should not send SYN packets in state `ESTABLISHED`) and should therefore be dropped.

The endpoints see the situation differently. Alice can clearly distinguish between scenario 1, where she would repeat her SYN, and scenario 2, where she would repeat her ACK. Bob cannot distinguish between the scenarios (or at least not until he has seen Alice's reaction), but he does not need to: he would repeat his SYN&ACK in any case.

Another scenario is possible: Alice may have crashed and the SYN at hand could represent a new connection initiation (with the same source port as before). This scenario can also happen later on in a connection. What should the midpoint then do? Should it forward the SYN and risk damage to Bob? Should it just block the packet and hinder Alice from communicating with Bob? Should it send a RST in Bob's name? Should it also send a RST in Alice's name to Bob? All these questions must be answered when giving a midpoint specification of TCP.

In this example, we see that one reason why midpoints cannot always track the protocol states of the endpoints lies in packet loss. But packet loss is only part of the problem. Another reason lies in the fact that certain endpoint constructs lead to ambiguity from the midpoint's perspective. These are:

1. Multiple transitions with the same output:
   Consider two transitions $q_i \xrightarrow{-/a} q_j$ and $q_i \xrightarrow{b/a} q_k$. Assume that the midpoint has previously forwarded b to an endpoint with these two transitions and afterwards receives an a. It cannot know from which transition this a originated.

2. Transitions without output:
   In this case, a midpoint cannot distinguish between the start state of the transition and the end state of the transition (at least until it has seen other, unambiguous output from the endpoint). A special case is hidden states, which are states where all incoming and outgoing transitions have no output. Such states can never be identified by a midpoint.
3. A packet can be sent in different states:
   This makes an unambiguous mapping between packets and states impossible. While this is not a source of tracking problems, it does make recovering from them difficult.

4 Case Study: Differences in Midpoints Based on TCP

In the last section, we explained why it is nontrivial to build a midpoint from endpoint specifications. In this section, we now show the implications of a lack of midpoint specifications by documenting the current state of affairs. For this, we took three commonly used firewalls — Checkpoint [Che], netfilter / iptables [ipt], and ISA Server [Isa] — and reverse engineered them, testing them against our (as there is no other) TCP midpoint specification given in [SBC05] and then analysing the results by hand.

As a result, we derived three distinct TCP automata (see [vBBC07] for details). Below we describe three of the differences:

A 'clean' three-way-handshake is not enforced. To initiate a TCP connection, a so called three-way-handshake is used (see Figure 1). So let us assume the firewall has accepted a SYN from an endpoint E_0 (Alice) to another endpoint E_1 (Bob). If there is now a SYN&ACK from E_1 to E_0, then everything works as expected: the packet should be let through and the firewall should enter the next state. If there is another SYN from E_0 to E_1, then this will be a retransmission (it could be that the first SYN was lost between the firewall and E_1) and should be allowed as well. If there is a RST from E_1 to E_0, then E_1 does not want this connection, the packet should be let through, and the TCP automaton initialised. All other packets make no sense at this time and therefore should be blocked. Unfortunately, in all of the tested firewalls, additional packets were let through. As one example, a FIN from E_1 to E_0 is allowed during connection initiation in netfilter. But there is no connection to be closed: If E_1 is not accepting the connection, then it would send a RST.

After a FIN, data from both sides is still accepted. If E_0 sends a FIN to E_1, then this means that E_0 wants to close the connection. After this FIN, E_1 is still allowed to send data, but E_0 is not (it makes not sense to send data after requesting to close the connection), except for the ACK belonging to E_1's FIN. As packets may not arrive in their correct ordering at the firewall, the firewall cannot just drop all packets from E_0 after having seen a FIN from B. But the firewall should just let through older packets (based on the sequence number), a retransmission of the FIN, and the ACK to E_1's FIN&ACK (after having

received E_1's FIN&ACK). To accomplish this task, the firewall has to keep track of the sequence numbers. This appears not to be done and therefore too many packets are let through.

SYNs are accepted during already established connections. SYNs are only used for connection initiation. That means that if a connection is fully established, there will be no more legitimate SYNs (based on the sequence numbers) belonging to that connection. But netfilter and ISA Server accept SYNs (from the initiator of the connection) all the time. Checkpoint does block the SYNs, but always allows SYN & ACK, which is not much better.

These findings show that there is a lack of consensus, at best, and a general lack of understanding, at worst, about how TCP should be handled by a firewall. The result is that every vendor does something different. We would like to contribute a solution to this problem by showing, systematically, how to construct midpoint specifications from endpoint specifications.

5 Construction of a Midpoint Automaton from Endpoint Automata

In the preceding sections, we saw why there is a need for midpoint specifications. Basically there are two ways to construct such a specification: write it directly or generate it from the endpoint specifications. The first alternative has two major drawbacks: 1) the consistency with the endpoint specifications must somehow be assured and 2) it requires additional work for the protocol designers. The second alternative overcomes both problems.

5.1 Setting

We subsequently consider only two-party protocols, i.e. protocols for only two endpoints. This covers most network protocols.[2] Let E_0 and E_1 be the endpoints and M the midpoint through which communication passes. Communication takes place in the form of messages, where the endpoint specification of the communication protocol determines when an endpoint may send which kind of message. For every message arriving at the midpoint, the midpoint can either forward the message or drop it (Figure 3).

Fig. 3. A message m from endpoint E_0 to endpoint E_1 is forwarded (left) or dropped (right) by the midpoint M

We write $X \to Y : m$ to express that the message m is sent from the endpoint X to the other endpoint Y, where $X \in \{E_0, E_1\}$, $Y \in \{E_0, E_1\}$, and $Y \neq X$. As

[2] It is straightforward to extend our approach to protocols with more endpoints.

there is a midpoint M between E_0 and E_1, every $X \to Y : m$ can be divided into the two parts $X \to M : m$ and $M \to Y : m'$. This makes explicit on which side of the midpoint a message is and also simplifies the specification of the actions of the midpoint: $m' = m$ if the midpoint forwards the message unaltered, $m = -$ (where $-$ signifies no external output) if the midpoint drops the message, and $m' \neq m$ if the midpoint alters the message before forwarding it. The messages may be altered, for example, when using Network Address Translation (NAT) in the firewall. For the sake of simplicity, we will not consider this case.

We will construct our midpoints to be *permissive* rather than *restrictive*. This means that our midpoint forwards messages if they could have resulted from protocol-conform endpoints. Thus our midpoint can possibly accept an incorrect message, but only in the cases where there is a scenario where this message could occur.

For the transport of the messages between the endpoints (via the midpoint), we assume a network that either (1) delivers messages, although not necessarily preserving the order, or (2) looses them.

5.2 Idea

Before giving the construction of a midpoint automaton from endpoint automata in Section 5.3, we first sketch the ideas behind our construction.

We model the global state of a system (the endpoints, midpoint, and network) at some time t as a state $st^t = (q_0^t, q_M^t, q_1^t, net^t)$, where q_i^t is the state of endpoint E_i at time t, q_M^t is the state of the midpoint M at time t, and net^t consists of all messages travelling between the endpoints at time t.

The midpoint M has to base its actions on the state of its environment. If M could observe all actions in the system, $q_M^t = (q_0^t, q_1^t, net^t)$ would hold at any time t, meaning that M always knows the exact states of the endpoints and the contents of the network. But midpoints generally cannot always determine the correct values of these components and hence we will let the state of our midpoint be a set of such triples, where each of these triples represents a possibly correct view of the system. Thus the triples of one state q_M^t are equivalent in the sense that they are not distinguishable by the midpoint with its current knowledge, i.e. based on the traffic it has previously observed.

To provide further intuition about the functioning of a midpoint, let us consider an example. Suppose the system starts in the global state $st^1 = (q_1, \{(q_1, q_1, net^1)\}, q_1, net^1)$.[3] Assume the following steps are taken:

1. M sends (forwards) a message x to E_0.
   To track the fact that the network now contains x, M must change its state to $q_M^2 = \{(q_1, q_1, net^2)\}$, where $net^2 = net^1 \cup \{M \to E_0 : x\}$. The global state is then $st^2 = (q_1, \{(q_1, q_1, net^2)\}, q_1, net^2)$.

[3] This means that E_0 and E_1 both are in their start states, the network content is net^1 and the midpoint knows about all this. Note that q_1 of E_0 and q_1 of E_1 are not the same as they do not belong to the same automaton.

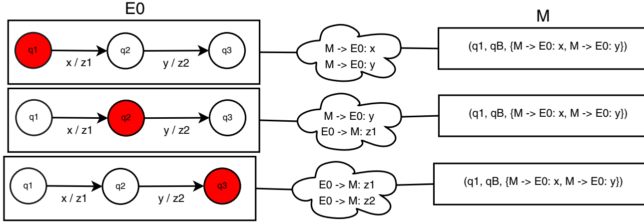

Fig. 4. Two consecutive endpoint transitions

2. x is received by E_0 and used as input to its automaton. Suppose, for E_0 in state q_1, there are only two transitions: $q_1 \xrightarrow{x/y} q_2$ and $q_1 \xrightarrow{-/z} q_3$. As M does not know if and when E_0 makes a transition, it cannot directly act on this step and thus its state is wrong. The global state is $st^3 = (q_2, \{(q_1, q_1, net^2)\}, q_1, net^3)$, where $net^3 = net^2 \setminus \{M \to E_0 : x\} \cup \{E_0 \to M : y\}$.
3. A message y reaches M.
   To take a correct decision, M must compute what could have happened (all possible successor steps) since its last step. This is either:
   - nothing, i.e., (q_1, q_1, net^2),
   - E_0 consumes x, makes a transition to q_2, and outputs y: (q_2, q_1, net^3), or
   - E_0 makes a transition to q_3, and outputs z: $(q_3, q_1, net^2 \cup \{E_0 \to M : z\})$.

   Now, M can determine its reaction on y. If there is one or more triple having y in its net (a possibly correct scenario where y occurred), y is forwarded and M's next state will consist of all these matching triples with their nets updated: $q_M^4 = \{(q_2, q_1, net^4)\}$, where $net^4 = (net^3 \setminus \{E_0 \to M : y\}) \cup \{M \to E_1 : y\}$. The global state is $st^4 = (q_2, \{(q_2, q_1, net^4)\}, q_1, net^4)$.

In the example above, there was only one endpoint transition between two consecutive midpoint transitions. In such a case, tracking (computing all possible successor states) is not difficult. The situation is more complex if more than one endpoint transition can happen between two consecutive midpoint transitions. Figure 4 provides an example. Since communication need not be order preserving, after two consecutive endpoint transitions, the second message $(E_0 \to M : z2)$ may reach the midpoint first. Thus it would not be enough if the midpoint computed only the next possible state (reachable in one step), but all possible successor states are needed, as only this would lead to $\{(q_1, q_B, \{M \to E_0 : x, M \to E_0 : y\}), (q_2, q_B, \{M \to E_0 : y, E_0 \to M : z1\}), (q_3, q_B, \{E_0 \to M : z1, E_0 \to M : z2\})\}$ and thus lead to the correct action — forwarding $z2$ — and the correct next state $\{(q_3, q_B, \{E_0 \to M : z1, M \to E_1 : z2\})\}$.

Let us return to our first example to illustrate why all possibly correct messages must be forwarded. Assume that we have the following sequence of actions:

1. M forwards a message x to E_0.
2. x is lost by the network.
3. E_0 takes the transition to state q_3.

4. An intruder sends message y (which is incorrect at q_3).
5. y reaches M.

For the midpoint, this scenario looks exactly the same as the one before. But here y is an incorrect message. As we never want to block correct messages (our decision for a *permissive* rather than *restrictive* midpoint), we have to accept y here (it could be the correct one from above).

5.3 Construction

We will now give the technical details of our construction of a midpoint automaton from endpoint automata. A two-party protocol p can be specified by two Mealy automata (one for each endpoint):

$$A_0 = (Q_0, \Sigma_0, \Gamma_0, \delta_0, \lambda_0, q_{0,1}) \qquad \text{and} \qquad A_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, \lambda_1, q_{1,1}).$$

Note that $\Sigma_0 = \Gamma_1$ and $\Gamma_0 = \Sigma_1$ since these automata must be able to communicate with each other. Often even A_0 and A_1 are the same.

The network can be modelled as a multiset (also called bag) *net*, which stores all messages in transit between the midpoint and the endpoints. Specifically

$$net \subseteq \mathcal{M}(S) = \{x' \mid x \subseteq S, x' =_s x\},$$

where S is the set of messages allowed by the protocol and $=_s$ denotes set equality (the sets contain the same elements, ignoring repetition).

In our construction, *net* will be part of a state of a deterministic automaton. Since we cannot handle a network of infinite size we must forbid actions of the endpoints and the network that can put infinitely many packets into the network. Hence, for the endpoints, we forbid loops without input in their protocol automata. For the network, we do not model message duplication. These restrictions are not problematic as the former should not be present and the latter can easily be detected and handled on another layer. Thus, it suffices to consider

$$net \subseteq \mathcal{P}(S), \text{where } S = \{X \to Y : m | X, Y \in \{E_0, E_1, M\}, Y \neq X, m \in (\Sigma_0 \cup \Gamma_0)\}.$$

Before starting with the construction of the midpoint automaton, we need to define the actions that are possible in our system. These are either transitions by the endpoints or the midpoint, based on their automata, network loss, or a message inserted by an intruder.

Definition 1. *$st^{t+1} = (q_0^{t+1}, q_M^{t+1}, q_1^{t+1}, net^{t+1})$ is a* successor state *of $st^t = (q_0^t, q_M^t, q_1^t, net^t)$, denoted $st^t \vdash st^{t+1}$, if one of the following conditions holds.*

Midpoint transition: For any $msg \in net$ with $msg = (E_i \to M : m)$,

$$\begin{aligned} q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= \delta_M(q_M^t, msg), \\ net^{t+1} &= (net^t \setminus \{msg\}) \cup \{\lambda_M(q_M^t, msg)\}. \end{aligned} \tag{1a}$$

Correct endpoint transition: For any $msg \in net^t$ *with* $msg = (M \rightarrow E_i : m)$ *(the endpoint taking an input from the network) or for* $m = -$ *(no input)*

$$\begin{aligned} msg' &= \begin{cases} (E_i \rightarrow M : \lambda_i(q_i^t, m)) & if \lambda_i(q_i^t, m)) \neq -, \\ - & otherwise, \end{cases} \\ net^{t+1} &= (net^t \setminus \{msg\}) \cup \{msg'\}, \\ q_i^{t+1} &= \delta_i(q_i^t, m), \\ q_{1-i}^{t+1} &= q_{1-i}^t, \\ q_M^{t+1} &= q_M^t. \end{aligned} \tag{1b}$$

Incorrect transition:

$$\begin{aligned} msg' &\in (\Gamma_i \setminus \{\lambda_i(q_i^t, m) | (M \rightarrow E_i : m) \in net^t \text{ or } m = -\}), \\ net^{t+1} &= net^t \cup \{msg'\}, \\ q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= q_M^t. \end{aligned} \tag{1c}$$

Network loss: for any $msg \in net^t$*,*

$$\begin{aligned} q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= q_M^t, \\ net^{t+1} &= net^t \setminus \{msg\}. \end{aligned} \tag{1d}$$

Note that as the network is modelled by a set, the permutation of messages is handled implicitly. Furthermore, note that an endpoint transition must not produce output. We denote empty output as '$-$'.

Definition 2. *Transitions* (1a), (1b) *and* (1d) *represent* correct transitions*. We denote a message resulting from a correct transition as a* correct message *and all other messages as* incorrect messages*.*

Definition 3. *A* correct trace *is a trace* $st^1 \vdash st^2 \vdash ... \vdash st^n$*, where every transition* $st^i \vdash st^{i+1}$*, with* $1 \leq i < n$*, is a correct transition.*

Definition 4. *The* message history *of a trace* $tr = st^1 \vdash st^2 \vdash ... \vdash st^n$ *is a sequence of messages* $m_1, m_2, ..., m_t$*, where* t *is the number of non-midpoint transitions in* tr *that produce output, and* m_i *is the output from the* i*th of these transitions.*

Definition 5. *The* midpoint message history *of a trace* $tr = st^1 \vdash st^2 \vdash ... \vdash st^n$ *is a sequence of messages* $m_1, m_2, ..., m_s$*, where* s *is the number of midpoint transitions in* tr*, and* m_i *is the input of* M *at its* i*th transition in* tr*.*

Definition 6. *Two* traces *are* midpoint equivalent *if they have the same midpoint message history.*

Definition 7. *Two* triples (a, c, d) *and* (e, g, h) *are* midpoint equivalent *if there exist two midpoint equivalent traces* tr_1 *and* tr_2 *with* $tr_1 = s^1 \vdash s^2 \vdash ... \vdash (a, b, c, d)$ *and* $tr_2 = st^1 \vdash st^2 \vdash ... \vdash (e, f, g, h)$.

To have a correctly functioning midpoint, two properties about a midpoint state q_M must be satisfied: 1) one of the triples in q_M is the correct one (Definitions 8 and 9) and 2) only possibly correct messages are forwarded (Definition 10).

Definition 8. q_M^t *is a* correct tracking at time t *if* q_M^t *is neither too small nor too large.* Not too small *means that* $(q_0^t, q_1^t, net^t) \in q_M^t$. Not too large *means that all* $q \in q_M^t$ *are midpoint equivalent to* (q_0^t, q_1^t, net^t).

Definition 9. M tracks endpoints correctly *if for every midpoint transition* $(q_0^{n-1}, q_M^{n-1}, q_1^{n-1}, net^{n-1}) \vdash (q_0^n, q_M^n, q_1^n, net^n)$ *in a trace,* q_M^n *is a correct tracking at time* n.

Definition 10. M computes outputs correctly *if for every trace* $tr = st^1 \vdash \ldots \vdash st^n$ *and every* t, $1 \leq t \leq n$ *we have:*

$$\lambda_M(q_M^t, E_i \rightarrow M : m) = \begin{cases} M \rightarrow E_j : m & \text{if } E_i \rightarrow M : m \text{ occurs in the message history of any trace } tr' \text{ which is midpoint equivalent to } tr, \\ - & \text{otherwise.} \end{cases}$$

where $j = 1 - i$.

As M cannot distinguish between tr and tr' in the above, it must forward all messages that occur in any of these traces, in order to avoid ever dropping a correct message.

Based on A_0 and A_1, we will now construct a Mealy automaton A_M for the handling of protocol p by the midpoint:

$$\begin{aligned} A_M &= (Q_M, \Sigma_M, \Gamma_M, \delta_M, \lambda_M, s_M) \\ Q_M &= \mathcal{P}(Q_0 \times Q_1 \times net) \\ \Sigma_M &= \{E_0 \rightarrow M : a \mid a \in (\Gamma_0 \setminus \{-\}\} \cup \{E_1 \rightarrow M : a \mid a \in (\Gamma_1 \setminus \{-\}\} \\ \Gamma_M &= \{M \rightarrow E_0 : a \mid a \in (\Sigma_0 \setminus \{-\}\} \cup \{M \rightarrow E_1 : a \mid a \in (\Sigma_1 \setminus \{-\}\} \cup \{-\} \\ q_{M,1} &= \{(q_{0,1}, q_{1,1}, \{\})\} \end{aligned}$$

Before we define the functions δ_M and λ_M, we first analyse the different possible scenarios. We do this with the help of Figure 5. There we describe the relationship between the actions of an endpoint E_0 (the situation for E_1 is analogous), the network, and the midpoint M. In particular, we consider how the four different types of transitions an endpoint can take ($x/-$, x/y, $-/y$, and $-/-$, for

Endpoint E_0	Network	Midp.	correct midpoint transition	Eq.
-	E0 x M	x M	$\delta_M(q_M, -) = \{(q_1, q_{E_1}, net_M \setminus \{M \to E_0 : x\})\}$ $\lambda_M(q_M, -) = -$	(2c)
q1 x / - q2	E0 x M		$\delta_M(q_M, -) = \{(q_2, q_{E_1}, net_M \setminus \{M \to E_0 : x\})\}$ $\lambda_M(q_M, -) = -$	(2d)
q1 x / y q3	E0 x y M		$\delta_M(q_M, -) = \{(q_3, q_{E_1}, net_M \setminus \{M \to E_0 : x\})\}$ $\lambda_M(q_M, -) = -$	(2d) (2c)
	E0 x y M	x y M	$\delta_M(q_M, E_0 \to E_1 : y) = \{(q_3, q_{E_1}, (net_M \setminus \{M \to E_0 : x\}) \cup \{M \to E_1 : y\})\}$ $\lambda_M(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(2d)
q1 - / y q4	E0 x y M		$\delta_M(q_M, E_0 \to E_1 : y) = \{(q_4, q_{E_1}, (net_M \setminus \{M \to E_0 : x\}) \cup \{M \to E_1 : y\})\}$ $\lambda(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(2c) (2e)
	E0 y M	y M	$\delta_M(q_M, E_0 \to E_1 : y)$ $= \{(q_4, q_{E_1}, net_M \cup \{M \to E_1 : y\})\}$ $\lambda(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(2e)
	E0 y M	M	$\delta_M(q_M, -) = \{(q_4, q_{E_1}, net_M)\}$ $\lambda_M(q_M, -) = -$	(2e) (2c)
q1 - / - q5	E0 M		$\delta_M(q_M, -) = \{((q_5, q_{E_1}, net_M)\}$ $\lambda_M(q_M, -) = -$	(2e)

Fig. 5. A transition in an endpoint, from a midpoint's view

$x \in \Sigma_0$, $y \in \Gamma_0$) look from the endpoints', the network's, and the midpoint's respective point of view. These are shown in columns 1 – 3, where one row represents one case. Note that one view (row) of one principal can belong to several views of another principal.

In the fourth column, the correct midpoint transition is shown. That is the transition the midpoint must take if it wants to correctly track the endpoint's state and the messages in the network.[4] For this we assume q_M=$\{(q_1, q_{E_1}, net_M)\}$ to be the state of the midpoint after its last transition (where applicable, this is forwarding x). Note that net_M contains all the messages in the network. Therefore a message has to be removed from net_M if it is no longer in the network, either because it was consumed by an endpoint or midpoint, or lost by the network.

As an example, let us explain the contents of the third row (in the first column, the third and the forth row coincide). Here a message x is forwarded by M to E_0 (3rd column), i.e. $M \to E_0 : x$. This message then reaches E_0 (2nd column), which uses it as input to its x/y-transition (1st column). After this transition, E_0 is in state q_3 (1st column) and a message y ($E_0 \to M : y$) has been inserted

[4] We will later give definitions of δ_M and λ_M that incorporate all these scenarios. The fifth column gives the number of the corresponding equations.

into the network (2nd column). This message is then lost by the network (2nd column). To correctly represent these actions, M has to change its state as given: E_0 is now in state q_3 and the *net* no longer contains x (4th column). Note that net_M does not contain y as its insertion is compensated by its removal.

With the help of Figure 5, we will now define the successor function. This function computes all direct successor states of a triple of the midpoint state (the transition function will then later choose some of these triples, based on its input). The figure illustrates why we sometimes have more than one possible successor state: the midpoint (3rd column) cannot distinguish all the scenarios (rows). Note that Figure 5 only considers one endpoint, whereas *succ* considers both endpoints (the equations (2d) and (2e) for E_0 correspond to the equations (2f) and (2g) for E_1).

$$succ(q_M) = \bigcup_{q \in q_M} \bigcup_{(M \to E_0 : m_1) \in net_M} \bigcup_{(M \to E_1 : m_4) \in net_M} \bigcup_{msg \in net_M} \tag{2a}$$

$$\{(q_0, q_1, net_M), \tag{2b}$$

$$(q_0, q_1, net_M \setminus \{msg\}), \tag{2c}$$

$$(\delta_0(q_0, m_1), q_1, (net_M \setminus \{M \to E_0 : m_1\}) \cup m_2) \tag{2d}$$

$$(\delta_0(q_0, -), q_1, net_M \cup m_3) \tag{2e}$$

$$(q_0, \delta_1(q_1, m_4), (net_M \setminus \{M \to E_1 : m_4\}) \cup m_5), \tag{2f}$$

$$(q_0, \delta_1(q_1, -), net_M \cup m_6)\} \tag{2g}$$

where

$$m_2 = \begin{cases} \{E_0 \to M\} : \lambda_0(q_0, m_1) & \lambda_0(q_0, m_1) \neq -, \\ \emptyset & \text{otherwise}, \end{cases} \tag{2h}$$

$$m_3 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, -)\} & \lambda_0(q_0, -) \neq -, \\ \emptyset & \text{otherwise}, \end{cases} \tag{2i}$$

$$m_5 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, m_4)\} & \lambda_1(q_1, m_4) \neq -, \\ \emptyset & \text{otherwise}, \end{cases} \tag{2j}$$

$$m_6 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, -)\} & \lambda_1(q_1, -) \neq -, \\ \emptyset & \text{otherwise}. \end{cases} \tag{2k}$$

The function *succ* computes all the states that are reachable in one step by an endpoint or the network. Since we are interested in all possible successor states, we must compute the closure of *succ*, defined as

$$cl(succ(x)) = \bigcup_{i=0}^{\infty} succ^i(x). \tag{3}$$

Observe that the closure is monotonic. It also has an upper bound, namely

$$cl(succ(q_M^t)) \subseteq \mathcal{P}(\{(q_0, q_1, n) | q_0 \in Q_0, q_1 \in Q_1, n \in net\}).$$

Hence, as Q_0, Q_1, and *net* are finite, $cl(succ(q_M^t))$ is also finite.

We now can define δ_M. The idea is to let our midpoint track all possible actions. We do this by first calculating the closure of all possible next states before actually executing a transition based on them.

$$\delta_M(q_M^t, m) = \bigcup_{(q_0, q_1, net_M) \in cl(succ(q_M^t))} \{(q_0, q_1, (net_M \setminus \{m\}) \cup \lambda_M(q_M, m))\} | m \in net_M\} \quad (4)$$

Note that $cl(succ(q_M^t))$ represents all possible successor states of q_M^t, whereas $\delta_M(q_M^t, m)$ only contains those successor states of q_M^t that can be reached with a message m. λ_M is now straightforward: If there is any triple where the input occurs, i.e. the message is correct in some midpoint-equivalent trace, the input is forwarded.

$$out((q_0, q_1, net_M), E_i \rightarrow M : y) = \begin{cases} (M \rightarrow E_j : y) & \text{if } \{E_i \rightarrow M : y\} \in net_M, \\ - & \text{otherwise.} \end{cases} \quad (5)$$

where $j = 1 - i$, and

$$\lambda_M(q_M, m) = \begin{cases} out(q, m) & \text{if there exists a } q \in cl(succ(q_M)) \text{ with } (out(q, m) \neq -), \\ - & \text{otherwise.} \end{cases} \quad (6)$$

Note that for each m, there is at most one non-empty (not '$-$') value for $out(q, m)$. Hence, λ_M is well-defined. This is due to the fact that our midpoint either drops or forwards a message. This would have to be revised for a midpoint that alters messages (e.g. a firewall performing Network Address Translation).

5.4 Correctness

As stated in Section 5.2, a correctly functioning midpoint must satisfy two properties: 1) one of its state triples is correct and 2) only possibly correct messages are forwarded. In this section, we prove that a midpoint, constructed as described in Section 5.3, satisfies the above properties.

Property I: one state triple is correct.
We prove the first property with the help of the following lemmas.

Lemma 1. *Given an M produced by our midpoint construction and a trace $st^1 \vdash st^2 \vdash ... \vdash st^n$, if there is a correct tracking at time $t1$, then the tracking after the next midpoint transition $st^{t2} \vdash st^{t2+1}$, $t2 \geq t1$, is also correct.*

Lemma 2. *Given an M produced by our midpoint construction and a trace $tr = st^1 \vdash st^2 \vdash \cdots \vdash st^n$, for any midpoint transition $st^{t2} \vdash st^{t2+1}$, there is a correct tracking at each time $t1$, with $1 \leq t1 \leq t2 < n$.*

Lemma 3. *Given an M produced by our midpoint construction, M tracks endpoints correctly (as defined in Definition 9).*

Lemma 3 follows from the other two. If for every midpoint transition there exists an earlier correct tracking (Lemma 2), then the tracking after the midpoint transition is correct (Lemma 1). Hence the tracking after every midpoint transition is correct.

Proof sketch for Lemma 1
We will only sketch the proof of the first lemma. The proof in its entirety can be found in [vBBC07].

Recall that δ_M consists of two parts: 1) compute all possible successor states using $cl(succ(q_M^{t1}))$, and 2) keep only the state triples that are feasible with respect to a given message. The second part directly reflects the definition of a midpoint transition. To prove the first part, we show that *succ* can track one non-midpoint transition correctly, and that taking the closure of *succ* computes any number of non-midpoint transitions correctly.

Proof of Lemma 2
We prove the lemma by induction on the midpoint transitions in a trace.

Base Case, the first midpoint transition in a trace.
The first state st^1 is a correct tracking at time 1:

$$st^1 = (q_0^1, q_M^1, q_1^1, net^1) = (q_{0,1}, (q_{0,1}, q_{1,1}, \{\}), q_{1,1}, \{\}).$$

The first midpoint transition cannot take place before $st^1 \vdash st^2$. By Lemma 1, this means that the tracking after the first midpoint transition is correct.

Step Case, the nth midpoint transition, $n > 1$.
By the induction hypothesis, the tracking is correct after the $(n-1)$th midpoint transition. By Lemma 1, this implies that the tracking is also correct after the nth midpoint transition. QED.

Property II: only possibly correct messages are forwarded.

Lemma 4. *M computes outputs correctly (as defined in Definition 10).*

We show the correctness of $\lambda_M(q_M^t, msg)$ in two steps:

1. $msg = (E_i \rightarrow M : m)$ was inserted by a correct transition.
   There is is a triple $q_{corr}^t \in q_M^t$ with $q_{corr}^t = (q_0^t, q_1^t, net^t)$ (see the proof of Lemma 1 in [vBBC07]). The output of this triple, defined by Equation (5), is correct, namely $msg' = (M \rightarrow E_j : m)$. For every other triple q, $out(q, msg)$ is either msg' or $-$. Thus, by Equation (6), $\lambda_M(q_M^t, E_i \rightarrow M : m)$ is correct.
2. msg was inserted by an incorrect transition.
   As seen above, there can only be a $(q_0, q_1, net) \in q_M^t$ with $msg \in net$ if this represents a possibly correct scenario. But, in this case, forwarding msg is correct.[5] QED.

[5] Note that in this case, the endpoints might not be able to continue their run of the protocol; the incorrect endpoint is only able to continue to send messages if they belong to a possibly correct scenario. This is the price of having a permissive firewall.

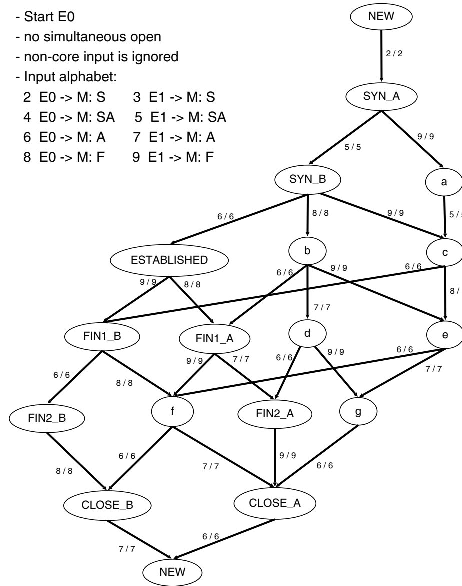

Fig. 6. Midpoint Automaton for TCP

5.5 Discussion

In Section 4, we analysed the TCP automata of several firewalls. We now compute the TCP midpoint automaton using the construction just presented. For endpoint automata, we use the automaton from the TCP specification for endpoints [ISI81, page 23].

The endpoint automaton in the TCP specification combines the initiator and the responder role. These roles are handled differently by firewalls, which distinguish between the networks outside and behind the firewall. Normally only one side is allowed to initiate a connection. Therefore we made two copies of the TCP endpoint automaton from the specification, one for each of the roles, which we adapt as follows. We chose E_0 to play the role of the *initiator*. Therefore we denote the state `CLOSED` as the start state of automaton A_0 and delete the state `LISTEN` from A_0. Furthermore, as we are only interested in one run of the protocol, we name the end state of A_0 `CLOSED2` (instead of `CLOSED`). To let E_1 play the role of the *responder*, we denote the state `LISTEN` as the start state of A_1 and delete the state `SYN-SENT` and the transitions from state `CLOSED` to state `LISTEN` from A_1. The resulting, minimised midpoint automaton can be found in Figure 6. Note that although this automaton represents only a subset of the TCP protocol this is not a limitation of our algorithm. Our algorithm can handle sequence numbers and the like if they are part of an endpoint automaton.

As expected, in each state of the midpoint, there is considerable uncertainty about the exact state of the endpoints. This is reflected in the fact that some midpoint states consist of over 60 triples. Despite this, the automaton is of

manageable complexity, in particular the number of outgoing transitions per state is small (1 – 3). The multiple transitions reflect the (limited) ways that messages can be sent independently by the endpoints and how they can be reordered by the network.

Note too that our midpoint automaton has 7 more states (a – g) than our reverse-engineered TCP automata. This reflects the additional complexity necessary to properly track possible network events. Let us illustrate this with an example. In our midpoint automaton, it can clearly be seen that, to get from state `SYN_B` to state `FIN1_B`, two messages — an ACK from E_0 and a FIN from E_1 — are needed. These messages are independent and thus can arrive in either order at the firewall. If we look how actual firewalls handle this, we see that the intended order of sending the ACK before the FIN leads to the same result, but that the opposite order ends in state `ESTABLISHED`, leaving us without an explanation why the FIN needs to be allowed in state `SYN_B`.

Our construction builds permissive midpoint automata. This reflects our decision not to penalise protocol-conform endpoints for actions of the environment (here the network). But it is a simple matter to modify the approach to construct restrictive midpoint automata. These can be built by stopping — i.e. dropping everything from then on — at states that consist of more than one triple. But building a restrictive automaton makes little sense with current protocols: It requires dropping more or less everything, as there will be uncertainty already after a few packets.

As discussed in Section 3, a midpoint may send (spoofed) messages to the endpoints to tear down (reject) an unwanted connection. Note that the decision whether such a message is sent is part of the midpoint's policy, not of the automaton. Therefore, similar to instantiating a new instance of the midpoint automaton after receiving the first packet of a connection, the sending of such messages should result in the deletion of the corresponding instance of the midpoint automaton.

6 Conclusion and Future Work

In this paper, we have shown why midpoints must behave, and hence be specified, differently from endpoints. Furthermore we have given an algorithm to generate midpoint automata from endpoint automata. Our solution should be of interest to at least two groups: those building midpoints and those analysing (e.g. testing) them. Both groups will benefit from having a general method to systematically construct midpoint specifications from those for endpoints.

The construction presented has two minor limitations: it requires that the endpoint automata do not have loops without input and it does not take duplication in the network into account. The first point is unproblematic, as loops without input should not be present since these would enable one endpoint to loop infinitely without communicating (only "talking" not "listening") with the other endpoint. We believe the problem of duplicates (or retransmissions) should be solved independent of protocol automata. The midpoint should remember the

packets seen (unique id) and its decision, and then apply the same decision to duplicates received later. We intend to investigate if this solution is feasible.

We plan to use our algorithm in the area of firewall conformance testing. Namely, when testing a given firewall, we first determine the differences between the automaton implemented in the firewall and the generated midpoint automaton A_{M_p} for every protocol p.[6] Then the user can decide if the additional or missing transitions represent a problem. If not, we continue with the test of the policy, as described in [SBC05], based on the protocol automaton of the given midpoint. In this way, firewall conformance testing (and testing of other kinds of midpoints as well) is possible and we can give the users information (and control) on the strictness of their firewalls.

References

[ASH03] Ehab Al-Shaer and Hazem Hamed. Management and translation of filtering security policies. In *Proc. 38th Int. Conf. Communications (ICC 2003), IEEE*, pages 256–260, May 2003.

[BCMG01] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *POPL*, pages 206–219, 2001.

[BFN+06] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL*, pages 55–66. ACM, 2006.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. In *IEEE Transactions on Software Engineering*, volume SE-4, pages 178–187, May 1978.

[CVI89] Wendy Y. L. Chan, Son T. Vuong, and M. Robert Ito. An improved protocol test generation procedure based on UIOS. In *SIGCOMM*, pages 283–294, 1989.

[ipt] Harald Welte et al. netfilter/iptables (ip_conntrack 2.1). `http://www.netfilter.org/`.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. volume 17, pages 591–603, 1991.

[Gil61] A. Gill. State-identification experiments in finite automata. In *Information and Control, vol. 4*, pages 132 – 154, 1961.

[Gro00] Network Working Group. RFC 2979: Behavior of and requirements for internet firewalls, October 2000.

[Gro02a] Network Working Group. RFC 3234: Middleboxes: Taxonomy and issues, February 2002.

[Gro02b] Network Working Group. RFC 3360: Inappropriate tcp resets considered harmful, August 2002.

[ISI81] University of Southern California Information Sciences Institute. RFC 793: Transmission control protocol, September 1981.

[6] In the ideal case there are no differences at all or any differences are at least given (and justified) by the midpoint vendor.

[Che] Checkpoint Software Technologies Ltd. Checkpoint R55W. `http://www.checkpoint.com/`.

[Mea55] G.H. Mealy. Method for synthesizing sequential circuits. In *Bell System Technical Journal*, volume 34, pages 1045–1079, 1955.

[Isa] Microsoft. ISA server v4.0.2161.50. `http://www.microsoft.com/isaserver/default.mspx`.

[MWZ05] A. Mayer, A. Wool, and E. Ziskind. Offline firewall analysis. In *International Journal of Information Security*, pages 125–144, 2005.

[Pax97] Vern Paxson. Automated packet trace analysis of TCP implementations. In *SIGCOMM*, pages 167–179, 1997.

[SBC05] Diana Senn, David Basin, and Germano Caronni. Firewall conformance testing. In *Lecture Notes in Computer Science*, volume 3502, pages 226–241, May 2005.

[SD88] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. In *Computer Networks and ISDN Systems 15*, pages 285–297, 1988.

[vBBC07] Diana von Bidder, David Basin, and Germano Caronni. Midpoints versus endpoints: From protocols to firewalls. Technical report 552, ETH Zürich, Department of Computer Science, March 2007.