Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks Using Model Checking

Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall
University of New Mexico, Dept. of Computer Science
Mail stop: MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001
{royaen, joumon, kapur, crandall}@cs.unm.edu

Abstract

Idle port scanning uses side-channel attacks to bounce scans off of a "zombie" host to stealthily scan a victim IP address and determine if a port is open or closed, or infer IP-based trust relationships between the zombie and victim. In this paper, we present results from building a transition system model of a network protocol stack for an attacker, victim, and zombie, and testing this model for non-interference properties using model checking. We present two new methods of idle scans in this paper that resulted from our modeling effort, both of which have been verified through implementation. One is based on TCP RST rate limiting and the other on SYN caches. The latter does not require the attacker to send any packets to the victim on the port to be scanned, not even forged packets. This gives the attacker additional capabilities beyond known idle scan techniques, such as the ability to scan ports that are blocked by a firewall or locate hosts on trusted networks, to which the attacker is not able to route packets, and infer their operating system. This is in contrast to the one currently known method of idle scan in the literature, which is based on non-random IPIDs.

For the future design of network protocols, we suggest that a notion of trusted *vs.* untrusted networks and hosts be made explicit in the protocol stack all the way down to the IP layer. This will enable shared, limited resources to be divided in a way that achieves non-interference and makes idle scans impossible. Using symbolic model checking, we verify that separate RST rate limitations for trusted *vs.* untrusted hosts achieves non-interference (without the SYN cache). Using bounded model checking we model a split SYN cache structure with separate buffers for trusted and untrusted hosts and prove non-interference to a depth of 1000 transitions. Because each transition is roughly a packet, this means that idle scans are eliminated by the split SYN cache for all practical purposes.

1 Introduction

Network reconnaissance is the important first step of virtually all network attacks. By scanning the network, the attacker is able to gain valuable information about the hosts that exist and the services they offer, infer IP-based trust relationships between hosts that are enforced by firewall rules and router tables, and collect other information that they can use in the next stage of attack. In this paper, we show that model checking can be a useful framework for predicting and mitigating attacker capabilities. In idle scans, an attacker scans a victim without sending packets to that victim using its own return IP address. The model of idle scans that we describe in this paper led to the discovery of two new forms of idle scan. One of these, based on SYN cache structures that are common to all modern network stacks, gives an attacker capabilities beyond the one currently known form of idle scan in the literature. We demonstrate that it is possible to infer the liveness of hosts and some information about what operating system they are running on a subnetwork without the ability to route packets to that network. We also demonstrate that it is possible to port scan a network on a port that the firewall protecting the network blocks. This means that if, *e.g.*, a particular port is blocked by a firewall for an entire subnetwork, an attacker can scan the hosts on that subnetwork on that port from outside the firewall using idle scans. Finally, we demonstrate that if a distinction between trusted and untrusted hosts were made explicit in the lower layers of the network protocol stack, then separate RST rate limitations and a split SYN cache structure eliminates these attacks in our model of network stacks.

The two new forms of idle scan that have resulted from the model checking effort presented in this paper are based on RST rate limiting and SYN caches, respectively. These were discovered during the process of building the model and manifest as counterexamples to a non-interference property that are produced by the model checker. In the RST rate limiting counterexample, the zombie in this case is a FreeBSD machine that limits the number of RST packets that it will send in a given time period. The attacker can infer the port status of the victim by testing the rate at which the zombie will reply with RST packets, given that if the victim port is open then the zombie is sending as many RSTs to the victim as the attacker sends forged SYNs to the victim (that appear to be from the zombie).

The SYN cache counterexample is different from the existing IPID-based idle scan, which is described in Section 2, in that the attacker never sends packets to the victim, not even forged packets. Instead the attacker forges SYN packets from the victim to the zombie, and the zombie sends a SYN/ACK to the victim and places these SYN packets in its SYN cache (a data structure for holding half-open TCP connections for which a SYN/ACK has been sent but an ACK response has yet to arrive). Because RSTs and ICMP errors from the victim will cause this SYN cache entry to be removed, the attacker can effectively perform a SYN/ACK scan of the victim without needing the ability to route packets to the victim. The attacker does this by testing the state of the SYN cache by sending SYNs with its own return IP address and viewing the SYN/ACK responses. The replies of the victim probes can be inferred from the attacker's ability to get SYN cache entries for its own SYNs. This makes possible testing for the liveness of IP addresses on protected networks with a rudimentary form of OS detection, and even port scanning on certain types of hosts on a port that is entirely blocked by a firewall. More details on these counterexamples are given in Section 4.

Like virtually all side-channel attacks, idle scans are associated with shared, limited resources. Because these resources generally cannot be made unlimited, we recommend in light of our results that trust relationships between hosts be made explicit to those hosts all the way down to the IP layer. Currently the only distinction at the TCP and IP layers is subnetworks, which do not necessarily correspond to the IP-based trust relationships between hosts that are enforced by firewall rules and routing tables. Trusted hosts can be hosts protected by the same firewall or that have special trust relationships in the packets they can route to each other. By making a distinction between trusted and untrusted hosts non-interference can be achieved by statically dividing shared resources, effectively eliminating idle scans. We verify non-interference for our model with separate RST rate limitations using symbolic model checking. Then we demonstrate that our split SYN cache structure using bounded model checking to a depth of 1000 transitions has no violations of non-interference. This means that no practical attack exists.

This paper is organized as follows. Section 2 gives more background and related works. Our model is described in Section 3. This is followed by a description of the counterexamples discovered during the process of building the model and some experimental results from their implementation in Section 4. We demonstrate that non-interference is achievable by distinguishing between trusted and untrusted hosts in Section 4.3. This is followed by discussion and future work in Section 5 and the conclusion.

2 Background and Related Work

Because network reconnaissance is an important first step in most network attacks, a fair amount of previous work has focused on detecting port scans. Staniford *et al.* [31] use simulated annealing to detect stealthy scans. Leckie and Kotagiri [18] present a probabilistic approach for detecting port scans, and Muelder *et al.*

propose a visualization approach. The scan behavior of Internet worms has been studied [28, 35, 12, 34], as has the scan detection problem at the backbone level [29, 30] and measurements of port scans and their side effects at Internet telescopes [23]. Jung *et al.* [14] describe an approach based on sequential hypothesis testing. Gates [9, 10] and Kang *et al.* [15] consider the problem of stealth port scans based on using many distributed hosts (*e.g.*, a botnet) to perform the scan. To our knowledge, ours is the first study to model idle scans, which are a distinct stealth technique that, in addition to being used for stealth, can also can be used for inferring IP-based trust relationships. Passively identifying hosts that have no routable IP address and are hidden by network address translation [2, 17] is a related problem to idle scans, but assumes a very different threat model where some amount of traffic can be viewed passively by the attacker.

Idle scans were introduced by Antirez [1] in a 1998 posting to the bugtraq mailing list. The one currently known form of idle scan, based on non-random, sequential IPIDs of older network stacks, was also described in this posting and is described in more detail by Lyon [20]. An IPID is a unique identifier for each IP packet, used primarily for IP fragmentation. In early implementations of the IP protocol, the IPID was chosen sequentially by simply incrementing the IPID value for each packet. Antirez showed that this made it possible to perform an idle scan of a victim by using a third host, the zombie, which the attacker need not have control of, in a form of side-channel attack.

In this form of idle scan, the attacker queries the zombie for IP packet responses and observes the sequence of IPIDs in the zombie's responses. The attacker then sends one or more SYN packets to the victim on the target port to be scanned with the return IP address of the zombie and return port of a closed port on the zombie. If the victim replies to the SYN with a SYN/ACK, meaning the victim's port is open, then the zombie will reply to the victim with a TCP reset (RST) and the attacker will observe a discontinuity in the sequence of IPIDs that it receives from the zombie. If the victim port is closed, the SYN is dropped or replied to by the victim with a RST, which the zombie simply drops and no discontinuity is observed by the attacker. Thus, the attacker is able to infer the port status of the victim without revealing their return IP address to the victim. Furthermore, the attacker is able to infer trust relationships between the victim and zombie. For example, the attacker might infer that the victim only accepts connections from a particular trusted subnetwork by using a zombie on that subnetwork. This is the one known form of idle scan in the current literature. Modern network stacks randomize the IPID for security reasons not related to idle scans, so the zombie must be an older system for this known type of idle scan to work.

IPID-based idle scans have been implemented in nmap [20] using an algorithm that accounts for interference from other hosts and packet loss. Lyon [20] is a good resource for how this type of idle scan works in detail and the different uses of idle scans. FTP bounce scans [20] are currently the only known way to port scan a victim host or network without routing forged packets to that host or network from the attacker. These scans use a feature that has been largely discontinued in FTP server implementations because of the many potential abuses of it. FTP bounce scans require that the attacker be able to log into an FTP session on the zombie, and operate at layer 7 (the application layer) of the OSI network model, whereas the SYN cache idle scan that we describe operates at layers 3 and 4 (TCP/IP) and requires only that the attacker be able to route SYN packets to an open port on the zombie.

Non-interference [11] is a widely used concept of information flow security that has seen wide application for proving security properties of programs. The works that are most related to ours in this space are those that treat non-interference as two or more separate scenarios that must produce the same result from the attacker's view for non-interference to be demonstrated, *e.g.*, TightLip [37] or the work of McCamant and Ernst [21, 22]. We apply non-interference to network stacks in this paper. Non-interference proved to be a very fruitful model of information flow in this context, but for future work that might consider packet loss, packet delay, and other such factors, alternatives such as non-deducibility [32] or treating the problem as a covert channel problem and studying object storage [16] and timing channels [36] may be desirable. For the modeling effort presented in this paper, which is based on an abstracted model of real networks that does not include packet loss and delay, non-interference proved to be a very useful property because it can be specified with Linear Temporal Logic (LTL).

The model checker that we chose for our study is the Symbolic Analysis Laboratory (SAL) [3]. SAL provides a SAT-based bounded model checker that allows for counterexamples to be easily interpreted as a trace through the states of the model, or, in our case specifically, a sequence of packets. SAL also provides a BDD-based symbolic model checker. Model checking has been applied to many properties of network protocols and their implementations where specific bugs lead to security vulnerabilities or availability issues, (*e.g.*, [8, 25, 13]). We have particularly patterned our analysis following Rushby's tutorials for modeling the Needham-Schroeder protocol [26] to identify Lowe's bug and the fault-tolerant algorithm for maintaining interactive consistency (Byzantine agreement) [27] as the transition systems for these problems seem similar to the ones for modeling port scanning and side-channel attacks in a protocol stack. Our results demonstrate that model checking is also useful for studying information flow on networks, particularly in this paper within the context of idle port scans.

3 Formalizing non-interference analysis of idle scans

In this section, we first describe the basics of our network stack model, then we describe more details and its implementation in SAL, and finally we list simplifying assumptions of the model.

3.1 Modeling the network stack

A host is viewed to be at the end of the network, *i.e.*, an end host. Hosts have internal state, such as a SYN cache, RST rate limit variables, and receive buffers. Hosts also have ports, which can be open, closed or filtered and their status does not change. An open port is one that the host will accept an incoming TCP connection on. For UDP, open ports simply drop packets and closed ports send ICMP errors. Filtered ports behave as would a typical host, but for the results presented in this paper all ports are either open or closed and never filtered. Hosts reply to packets based on rules that model a typical Linux or FreeBSD network stack. Our model is based on the IP protocol and includes TCP (but only up to the point of half-open connections), ICMP, and UDP.

The SYN cache on a host is a cache for pending SYN packets for which a SYN/ACK has been sent and the host is waiting for an ACK to complete the TCP three-way handshake. The SYN cache drops duplicate SYNs for the same IP address and port pairs. In our model packets are only removed from the SYN cache when a TCP RST is received from the source IP address and port of the original SYN packet (because we only model half-open TCP/IP connections, so there is no ACK for the third part of a three-way TCP handshake). When the SYN cache is full, the host replies with a SYN cookie and drops the SYN. A SYN cookie is a method for sending an initial sequence number in the SYN/ACK that, when ACKed by the remote host, contains enough information to complete the connection so that no state about the half-open connection needs to be kept in memory [4]. TCP RST rate limiting, where the number of resets sent by a host is limited, is based on the FreeBSD implementation where separate rate limits are maintained for open ports and closed ports.

Figure 1 shows the basic definition of an idle scan that we use for our model. There are four IP addresses, three for the attacker, zombie, and victim hosts, and one where there is no host so all packets are dropped. A solid arrow denotes that the source host can send packets to the destination using its own return IP address. A dashed arrow indicates that the source can send a packet to the destination using any return IP address other than its own. The salient feature of this definition of an idle scan is that the attacker cannot send packets to the victim using its own return IP address. This entails that the victim never sends any packets to the attacker, and that the attacker therefore only ever receives packets from the zombie, since the victim and zombie only ever reply to packets using their real IP address as the return address.

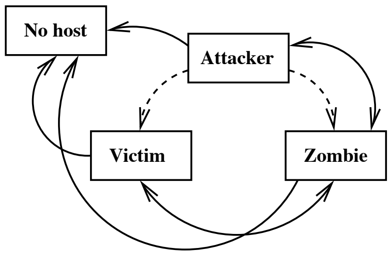

Figure 1: **Basic definition of an idle scan.**

Our goal is to ensure that the network satisfies the *non-interference property*, which is specified as: for any possible sequence of packets that the attacker can send to the victim and zombie, the sequence of packets the attacker receives in response is identical regardless of whether the target victim port is open or closed. This models the desired behavior that the attacker cannot gain any information about the target victim's port.

We do this by modeling two possible scenarios faced by the attacker which the attacker is attempting to distinguish and thus gain information about the victim. In each scenario, there is a victim and a zombie whose behavior and initial state are identical (except for the status of the target port on the victim, of course), but whose behavior and internal state over time can differ between scenarios through certain sequences of events due to the port status of the target port. In one scenario, the target port of the victim is open whereas in the second scenario, the target port is closed. The attacker sends identical packets in both scenarios.

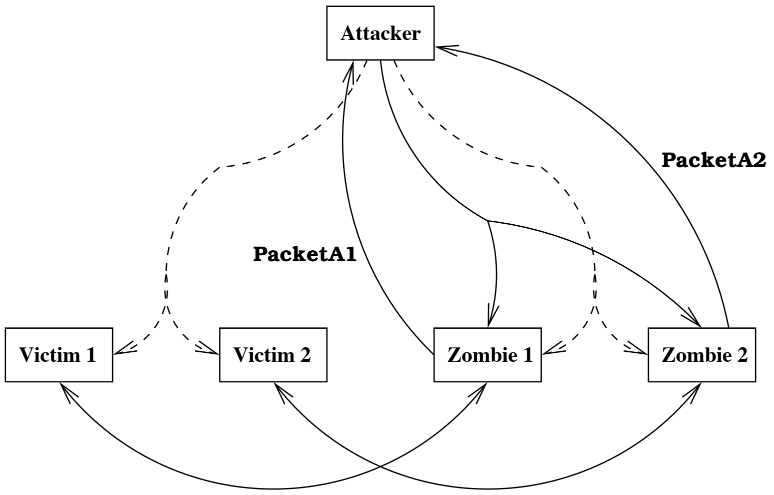

Figure 2: **Overview of our model (the IP address with no host that drops all packets is excluded from this figure for clarity).**

Figure 2 gives an overview of our model for testing non-interference properties of network stacks for idle scans. The status of the target port being open or closed is modeled as two different scenarios. Victim 1 and Zombie 1, for example, exist in scenario 1 where the target port on Victim 1 is open. Victim 2 and Zombie 2 exist in scenario 2 where the target port on Victim 2 is closed. The attacker can forge any arbitrary sequence of packets, but it must forge identical packets in both scenarios. The hosts in the different scenarios can respond differently and contain different internal state. `PacketA1` and `PacketA2` are the sequence of packets the attacker receives in scenario 1 and scenario 2, respectively.

In our model, the attacker can nondeterministically choose any arbitrary sequence of packets that do not violate the definition of an idle scan. Furthermore, the attacker need not reply to packets; the fact that the model allows the attacker to send any arbitrary packet covers all possibilities for reply. For the destination and return IP addresses of a packet, the attacker can choose among its own IP address, that of the victim or the zombie, or an IP address with no live host (that simply drops all packets). The only constraint is that the attacker cannot send a packet to the victim with its own IP address as the return IP address, as this violates the definition of an idle scan.

The attacker can distinguish between SYN cookies and regular SYN/ACKs that it receives in our model. This is true in reality due to the statistical properties of the initial sequence numbers of SYN cookies and the fact that they are never retransmitted whereas regular SYN/ACKs are.

The attacker can choose any values for the IP protocol (TCP, UDP, or ICMP), TCP flags, source and destination ports, validity of checksums, and so on. Every packet that the attacker forges is forwarded to the appropriate host in both scenarios.

3.2 SAL for modeling, generating counterexamples, and verifying properties

We model the network stack as a transition system. At an informal, high level, a transition system specifies computation as a sequence of transitions in a state machine. A state is given by the values of the local variables used to describe transitions. A transition system has an initial state. For every transition, there is an optional guard, which when true in the current state, leads the computation from the current state to the next state. For a nondeterministic transition system, multiple transitions may be triggered and one transition is randomly selected. This is repeated and the computation terminates if no guard is true in the current state. Dijkstra's guarded command language [7] is an example of a formalism for specifying transitions.

We used SAL (Symbolic Analysis Laboratory) for specifying the transition system and analyzing its properties. SAL is a language and a tool kit for specifying transition systems and analyzing them using model checking. SAL provides support for a suite of tools which have been successfully used for analyzing protocols and distributed algorithms (see [24]).

Figure 3 shows the outline of our SAL code for the model. Elipses indicate where detailed code has been omitted, the full model is 895 lines of SAL code.

For a transition step, a nondeterministic choice is made between the attacker, victim, or zombie. If the attacker is chosen, it forges a nondeterministic packet, which can be a "drop" packet that has no effect. This packet is placed in the receive queue of the destination IP address. If the victim or zombie is chosen, it removes the next packet from its FIFO receive queue and replies based on its internal state and configuration. The functions `ProperReply` and `UpdateSyncache` are responsible for choosing the packet to reply with, if any, and any updates to the host's internal state (specifically the RST counter and SYN cache). Note that these are pure functions and do not update any state themselves.

Figures 4 and 5 show how the `ProperReply` and `UpdateSyncache` functions are used. A transition has a guard, *e.g.*, "`(z1.fullness /= 0 AND z2.fullness /= 0 ) -->`", which is a quantifier-free formula specifying a condition on the current state and must hold before the transition is executed, and then a formula relating the current state with the next state. An example of such a formula is "`z1'.fullness = z1.fullness - 1`", where `z1'` is the variable in the next state and `z1` is the variable in the current state. In this example the variable will be decremented by 1 in the next state.

When the guard on a transition for the zombie or victim fires, that host must remove a packet from its queue and then reply and update its state in both scenarios. `UpdateSyncache` returns not only the new state of the SYN cache, but also a variable called `.synPIsThere` which can take on the values `put`, `notexist`, or `exist`. This return value is passed to `ProperReply`, which needs to know if a SYN packet was put in an entry in the SYN cache, no entry was found for it because the SYN cache is full, or already existed in the SYN cache. In this way `ProperReply` knows whether to send a SYN/ACK, send a SYN cookie, or drop the packet if the packet is a SYN, respectively.

For example, if the zombie in one of the scenarios receives a SYN packet, it calls `UpdateSyncache` to determine the new state of the SYN cache and what will happen to the packet. If the internal state of

```
Network_Protocol_Stack : CONTEXT =
BEGIN
   % Type Declarations
       Protocol : TYPE = {tcp, icmp, udp, invalidProtocol};
       IP: TYPE = [1..5]; % 1 is Attacker, 2 is Victim, 3 is Zombie
       PortStatus: TYPE = {open, closed, filtered};
       TCP_Flag : TYPE = {syn, fin, synack, rst, drop};
       Port: TYPE = [1..3]; % each host has 3 ports
       Type_ICMP: TYPE = [1..5];
       Packet: TYPE = [# sip_IP : IP, dip_IP : IP,  ihl_IP : ValidOrInvalid, ...
       Host : TYPE = [# ip : IP , portstatus : array Port of PortStatus, ...
...
   % Functions
       ProperReply(Packet, PortStatus, Rst_counter): Packet, Rst_counter;
         % This function returns the proper response to a given packet
         % and a reduced Rst_counter if a RST is sent.
...
       UpdateSyncache (Packet, Syncache): Syncache, SynPIsthereOrNot;
         % This function returns the new state of the SYN cache, which can change
         % for SYNs or RSTs, and a value indicating if the SYN already exists, was
         % dropped, or was placed in the SYN cache
...
     main : MODULE =
       BEGIN
         % Variable Declarations
           GLOBAL v1 : Host % victim 1
           GLOBAL v2 : Host % victim 2
           GLOBAL z1 : Host % zombie 1
           GLOBAL z2 : Host % zombie 2
...
           LOCAL PacketA1 : Packet % Packets sent to attacker in scenario 1
           LOCAL PacketA2 : Packet % Packets sent to attacker in scenario 2
...
         % Initialization Section
...
         % Transition Section
           TRANSITION
             [
               (v1.fullness /= QueueSize OR z1.fullness /= QueueSize )-->
               % Attacker creates a packet and puts it in queues of either
               % v1 and v2 or z1 and z2
...
             []
               (v1.fullness /= 0 AND v2.fullness /= 0 ) -->
               % v1 and v2 pop a packet and call ProperReply and UpdateSyncache to
               % choose a proper reply and update their internal state.
...
             []
               (z1.fullness /= 0 AND z2.fullness /= 0 ) -->
               % z1 and z2 reply and update their state
...
             ]
       END;
      theorem1: THEOREM main |- G( PacketA1 = PacketA2 );
END
```

Figure 3: **Outline of SAL code for the network protocol stack model.**

```
...
[]
               (z1.fullness /= 0 AND z2.fullness /= 0 ) -->
               % z1 and z2 reply and update their state
...
               collector_1'.packet = z1.queueOfHost[1] ;
               (FORALL (j: FullnessOfQueue ): z1'.queueOfHost[j-1] = z1.queueOfHost[j]);
               z1'.fullness = z1.fullness - 1;
...
               temp'= UpdateSyncache(...);
               z1'.syncache= temp'.host.syncache;
...
               response' = ProperReply(collector_1'.packet, z1.portsstatus[collector_1'.packet.dport], temp' ) ;
...
               % Do the same for z2, which may have a different packet in its queue
               collector_2'.packet = z2.queueOfHost[1] ;
               (FORALL (j: FullnessOfQueue ): z2'.queueOfHost[j-1] = z2.queueOfHost[j]);
               z2'.fullness = z2.fullness - 1;
...
```

Figure 4: **Overview of what happens when a transition fires.**

```
...
   % Functions
       ProperReply(Packet, PortStatus, Rst_counter): Packet, Rst_counter;
         % This function returns the proper response to a given packet
         % and a reduced Rst_counter if a RST is sent.
IF ...
...
        ELSIF ( Packet.protocol_IP = tcp AND  (Packet.seqNum_TCP = known AND Packet.ack_TCP = known )
                AND Packet.flag_TCP = syn AND ps = open AND Packet.synPIsThere2 = put  )
        THEN
              packetOut with .packet.sip_IP := Packet.dip_IP
                        with .packet.sport := Packet.dport
                        with .packet.dip_IP := Packet.sip_IP
                        with .packet.dport := Packet.sport
                        with .packet.flag_TCP := synack
                        with ...
...
       UpdateSyncache (Packet, Syncache): Syncache, SynPIsthereOrNot;
         % This function returns the new state of the SYN cache, which can change
         % for SYNs or RSTs, and a value indicating if the SYN already exists, was
         % dropped, or was placed in the SYN cache
        IF ( ... AND Packet.flag_TCP = syn AND Host.synQeueEntries[1]=valid
                        ... AND Packet.sip_IP = host.syncache[1].sip_IP ...)
        THEN  # Ignore duplicate SYNs, i.e., that already exist in the SYN cache
              synCollOut with .synPIsThere := exist
        ELSIF ( ... )
        THEN synCollOut
                with .synPIsThere := put
                with .host.syncache[1] := synCollIn.packet
...
```

Figure 5: **Structure of the** `UpdateSyncache` **and** `ProperReply` **functions.**

the zombie indicates that there is a free entry in the SYN cache, the fact that the SYN will be placed in the SYN cache and the new status of the SYN cache are returned by this function. Then `ProperReply` is called with this information as an argument, and this function will determine that the proper reply is a normal SYN/ACK, with the destination IP address as the source of the SYN, valid checksums, *etc.*

Another example is that a host (a victim or zombie) receives a SYN/ACK. `UpdateSyncache` returns the current state (*i.e.*, no changes will be made to the SYN cache state) and then `ProperReply` will be called and will ignore `.synPIsThere` because the packet is not a SYN. The return value of `ProperReply` depends on the RST counter. If the RST counter is non-zero the return value of `ProperReply` will be a RST packet that the zombie will use for a reply and a reduced RST counter. If the RST counter is already zero, `ProperReply` will return a drop packet and zero still for the RST counter. All possible TCP, UDP, or ICMP packets and their corresponding replies are enumerated in `ProperReply` based on the reply that a typical network stack would send.

By forcing the zombies or the victims in both scenarios to reply at the same time step, the model stays in sequence. If a host in one scenario (*e.g.*, the victim in the closed port scenario) replies to a packet whereas the corresponding host in the other scenario (*e.g.*, the victim in the open port scenario) drops the packet, a "drop" entry is inserted in the destination host's queue as a filler and eventually ignored. This ensures that the packets that are received by the attacker vary only when non-interference is violated, *i.e.*, only when the sequence diverges.

SAL supports a suite of tools; the ones most relevant for the analysis discussed in this paper include a dead-lock checker, a symbolic model checker for finite state systems based on the CUDD BDD package, and a bounded model checker based on the Yices SAT solver. Properties of a transition system are specified in Linear Temporal Logic (LTL). Our analysis involved using properties of the form $G(\alpha)$, where α is a quantifier-free, modality free formula expressed using state variables, to mean that α holds in every state of the transition system. The noninterference property is specified as:

$$\vdash \text{G(PacketA1 = PacketA2)}$$

This means that the sequence of packets the attacker receives in response to its probes from the zombie in the first scenario is always identical to the response from the zombie in the second scenario.

We have used SAL's bounded model checker for finding counter-examples as it is depth-first and explicitly enumerates states. SAL's symbolic model checker, which is exhaustive, is useful for finding smaller counter-examples as well as for proving properties of interest, which are often difficult to do by explicit state enumeration model checkers. A useful comparative study of exhaustive symbolic model checkers and explicit state enumeration model checkers is in [5] for protocol analysis and controllers.

3.3 Assumptions to reduce the number of states in the model

A number of assumptions were made to keep our model simple. Our strategy was to start with a simple model and introduce additional complexity into the model if no counterexamples are generated, and ensure that the abstractions we made caused no loss of generality that would exclude potential counterexamples.

A major abstraction in the model is that we consider the proper reply to SYN/ACK packets to be "drop" for open ports and RST for closed ports. In reality, network stacks that respond differently to SYN/ACKs on open *vs.* closed/filtered ports typically respond with RSTs or ICMP and have different rate limits per port. Since the lower rate limit (typically ICMP) will cause drops before the higher rate limit, without loss of generality, we can consider open ports to simply drop SYN/ACK packets from the initial state. This is equivalent to assuming that the attacker immediately exhausts the lower rate limit.

We also exclude ICMP and UDP from the split SYN cache version of our model. Since ICMP host error packets have the same effect on the SYN cache as RSTs, and other ICMP and UDP packets make no relevant changes to the destination host's TCP state, ICMP and UDP do not affect the non-interference property for the SYN cache structure. Invalid checksums in packet headers are also excluded, because they are dropped without affecting the state of the destination host in all cases.

Another major abstraction is that each of the two buffers in our split SYN cache has only a single entry. There are three reasons why only a single entry in the SYN cache is necessary in the model:

- Pending entries in the SYN cache with source IP addresses and ports (possibly forged by the attacker) that correspond to invariant ports (that have the same status in both scenarios) cannot cause divergence in the internal state of any of the hosts in the two scenarios. Thus, no more than one such SYN cache entry at a time can be useful for creating a counterexample.

- Even though RST rate limiting is performed separately for open and closed ports, the rate limit value stored by any host cannot be caused to diverge on invariant ports. Only the target port on the victim can cause divergence. Since only one such port exists, only one SYN cache entry at a time can be useful for creating a counterexample. If we had not received a counterexample under this assumption, we would have incrementally allowed more entries in the SYN cache.

- While the single entry is full, the SYN cookies generated in response to dropped SYN packets can only cause internal state differences if sent to the target port on the victim. If this is the case then the entry in the SYN cache cannot also be a SYN packet with the victim IP address and target port, since duplicate SYNs are ignored by the SYN cache. Thus, one SYN cache entry per trust level (trusted and untrusted) is general enough to handle all of the cases of any number of SYN cache entries.

Because of the above simplification of making the SYN cache have a single entry for each trust level, we modeled only three ports without loss of generality. Port 1 is prohibited (*e.g.* by a firewall rule) to the attacker for the split SYN cache implementation, meaning that the attacker cannot send packets to port 1. This was done so that we could include the RST rate limitation, which has important interactions with the SYN cache, without recieving the RST rate limit counterexample. Port 1 is closed in both scenarios for the zombie; however, for the victim, it is open in one scenario and closed in the other. In other words, port 1 is the target port for the attacker to get information. Port 2 is closed and port 3 is open on both hosts in both scenarios. Because closed ports are equivalent in terms of their responses, a single closed port per host is equivalent to any number of closed ports. Because the SYN cache has a single entry and open ports only have different behaviors based on the status of the SYN cache, a single open port per host is also equivalent to any number of open ports.

In real SYN cache implementations, there is a timeout after which SYNs that have not become fully open TCP connections are dropped. Because our model allows the attacker to remove any entry from the SYN cache at any time via a RST packet (which is also possible in reality for Linux SYN cache implementations), our model need not incorporate this timeout. Also, RST rate limiting is done per a time period in reality. A fixed limit of RSTs for an unbounded amount of time is a generalization of this that does not exclude any counterexamples because for any violation of non-interference based on a rate limit a single time period is enough to create a counterexample.

4 Finding and ameliorating idle scans

In this section, we describe the counterexamples that our modeling effort produced and give experimental results of an implementation of these counterexamples to demonstrate that they can indeed be used to do idle port scans.

Port	Zombie status	Victim Status
1	Open	Open in scenario 1, closed in scenario 2
2	Closed	Closed
3	Open	Open

Table 1: Ports and their status in our model.

4.1 Discovering counterexamples

We now describe the two counterexamples that were discovered.

RST count (open scenario)	**Victim Port 1 Open**	**Victim Port 1 Closed**	**RST count (closed scenario)**
1	**Zombie(actually Attacker):2 → Victim:1 SYN**		1
1	Victim:1 → Zombie:2 SYN/ACK	Victim:1 → Zombie:2 RST	1
1	Zombie:2 → Victim:1 RST		1
0	**Attacker:2 → Zombie:2 SYN/ACK**		1
0		*Zombie:2→Attacker:2 RST*	0

Table 2: RST rate limiting counterexample to the non-interference property.

SYN cache (open scenario)	**Victim Port 1 Open**	**Victim Port 1 Closed**	**SYN cache (closed scenario)**
0	**Victim(actually attacker):1 → Zombie:3 SYN**		0
1	Zombie:3 → Victim:1 SYN/ACK	Zombie:3 → Victim:1 SYN/ACK	1
1		Victim:1 → Zombie:3 RST	1
1	**Attacker:2 → Zombie:3 SYN**		0
1	*Zombie:3 →Attacker:2 SYN cookie*	*Zombie:3 →Attacker:2 SYN/ACK*	1

Table 3: SYN cache counterexample to the non-interference property.

4.1.1 RST rate limiting counterexample

When we applied SAL's bounded model checker to a simpler version of the model, in which the SYN cache did not play any role, for the property "⊢ G(PacketA1 = PacketA2)" SAL identified a counterexample with `RST_counter` set to 3 in the initial state. We simplified the model further by reducing the initial value of `RST_counter` to 1 still received a counterexample. The counterexample in this case was found much more quickly, at depth 5 in the transition system.

This counterexample is illustrated in Table 2. It is a trace through the state machine for our model that roughly corresponds to the sequence of packets involved in the idle scan. The left side of the table shows the relevant state on the victim (the RST limitation counter) and sequence of packets for the scenario where the victim target port is open. The right side is the scenario where the target port is closed. The state is the value of that variable before the packet is sent, *i.e.*, when a RST is sent the RST counter will be 0 in the next state. Everything shown in gray, including the state, is unknown to the attacker and can only be inferred.

In this example the attacker wants to discern the port status (open or closed) of port 1 on the victim. Port 2 on the zombie is closed. The packets the attacker sends are identical in both scenarios and are shown in bold. The packets that the attacker cannot see (because it is not the destination) are shown in gray. Packets that the attacker can see, and infer information about the port status of the victim from if they differ, are shown in italics. First, the attacker forges a SYN to the victim on the target port that appears to be from the zombie with return port 2. If the target port on the victim is open, it will respond to the zombie with a SYN/ACK on the zombie's closed port 2, causing the zombie to send the victim a RST and decrement its RST count. If the port is closed, the victim will respond to the zombie with a RST which the zombie ignores. Next, the attacker sends a SYN/ACK packet, using its own return IP address, to the closed port on the victim. If the attacker receives a RST in response, then it can infer the the victim target port status is closed since an open port would have caused the zombie to have already reached its RST rate limit.

4.1.2 SYN cache counterexample

For the second case, we tried a more complex model that included a SYN cache. We started with a SYN cache of size 2, then simplified it further to size 1, and SAL's bounded model checker still identified the counterexample to the noninterference property as illustrated in Table 3.

The relevant state in this case is the number of entries in the SYN cache, with a maximum value of 1 in our model. The notable thing about this form of idle scan is that the attacker never sends any packets to the victim, not even packets with forged return IP addresses. Instead, the attacker sends a SYN to the zombie on an open port, with the return IP address of the victim and the return port as the target port. The zombie places this SYN packet in the SYN cache, which in our model has only a single entry, and sends a SYN/ACK response to the victim. If the victim target port is closed it will send a RST in response, which causes the zombie to remove the relevant SYN cache entry so that there is now a free entry in the SYN cache. An open target port on the victim will simply drop the SYN/ACK from the zombie, so that the SYN cache of the zombie remains full since the zombie is still waiting for a response to the SYN/ACK. The attacker can then infer the status of the zombie's SYN cache, and therefore the victim port status, by sending a SYN to the zombie with the attacker's own return IP address. A regular SYN/ACK means the SYN cache entry was free, a SYN cookie indicates that it was empty.

Note that responses to SYN/ACKs on open, closed, or filtered ports vary for different operating system, but all that matters is that for open *vs.* closed or open *vs.* filtered the response differs in some way under certain conditions. More discussion of the possibilities for this is in Section 4.2. The SYN cache counterexample makes it possible to, *e.g.*, port scan a network on a port that is blocked for the entire network from outside the firewall. Imagine in Table 3 that the zombie and victim are behind a firewall and the attacker is outside the firewall. Even if the firewall drops all incoming packets with destination port, *e.g.*, 22 for Secure Shell (SSH), the attacker can scan port 22 on the network by using other open ports. Also, there may be firewall rules that enforce that only trusted machines (*e.g.*, the zombie) can route packets to the victim. In this case the victim might be an internal database server and the zombie is the web server interface to the database, for example. Information about what ports the victim has open might give the attacker an idea of whether compromising the zombie to subsequently get access to the victim is worth the effort and risk. It might also be that the attacker can route packets to the victim, but not on the target port. For example, many machines leave certain ports open only for their backup servers that contact them nightly. Or, the system administrator might only allow incoming SSH sessions on their critical servers from their own office machines and not from other IP addresses. Knowing these kinds of trust relationships and exploiting them to find out more about the victim machines can be very valuable to an attacker.

For each host, both the SYN cache and the reset rate limiting variables constitute shared, limited resources, which are the sources of violations of non-interference in our two counterexamples.

4.2 Experimental confirmation of counterexamples

We implemented both counterexamples to verify that these two new forms of idle scan that resulted from the modeling effort were possible for real hosts. Our results presented in this section demonstrate that the differences in the sequence of packets the attacker sees translate from the abstract notion of noninterference in our model to differences that can be seen in real network packet traces. Our implementations of the two idle scans are not optimized for speed or stealth, nor do they account for packet loss or packet delay, but in this section we discuss the practicality of these two forms of idle scan and conclude that they are both practical.

4.2.1 Experimental setup

For our experiments, we set up VirtualBox [33] virtual machines connected using IPv4 on two different subnetworks with TUN/TAP interfaces. The attacker machine was the host, and one subnet contained a Linux kernel 2.4 host (Fedora Core 1) which served as the zombie for the SYN cache idle scan implementation. The other subnet contained a Windows XP host with no service packs, a Linux kernel 2.6 host (CentOS 5.2), and a FreeBSD 7.1.1 host. The latter served as the zombie for RST rate limiting idle scan implementation. IP forwarding between these two subnetworks was performed by the host. Packets were generated and captured by separate threads using the Perl Net::RawIP and Net::Pcap libraries, respectively.

4.2.2 RST rate limiting idle scan implementation

In our transition system model, RSTs are limited to a finite number for infinite time. For a real FreeBSD system, RSTs are limited to a default of 200 per second, with separate limitations for open and closed ports. Our implementation sends 2000 each of two different types of packets, each at a rate of 180 per second, to the victim and FreeBSD zombie, respectively. One type of packet is forged SYNs to the victim on the target port that appear to be from the zombie on a port that is closed on the zombie. The other is SYN/ACKs from the attacker to the zombie, which the zombie should reply to with RSTs. If the zombie is sending RSTs at a rate of 180 per second to the victim in response to the victim's SYN/ACKs (meaning the victim target port is open), this should interfere with the rate at which the zombie sends RSTs to the attacker. Thus the number of RSTs the attacker receives in our experiment can be used to infer the port status of the target port on the victim. We repeated the RST rate limiting idle scan experiment 700 times each for an open and closed port on the victim. The victim was a Linux kernel 2.6 virtual machine. The host-based firewalls on both machines were disabled, although for the victim the idle scan works whether the host-based firewall is enabled or not. For FreeBSD, RST rate limiting does not apply to filtered ports. The pf host-based firewall is disabled by default for FreeBSD installations.

The results from our RST rate limiting idle scan are shown in Table 4, where the results are based on the number of RSTs the attacker receives. When the victim port is closed, the attacker receives all 2000 RST responses from the zombie. When the victim port is open, the attacker receives at most 1634 RSTs. Thus, determining if the target port is open or closed is straightforward for idle scans based on RST rate limiting.

4.2.3 SYN cache idle scan implementation

SYN cache implementations vary for different operating systems. While the SYN cache idle scan is possible on virtually any host, the simplest network stack to use for a zombie is Linux kernel 2.4. Linux kernel 2.4

Port status	Mean	Std. dev.	Min.	Max
Open	1552.1	47.0	1429	1634
Closed	2000	0	2000	2000

Table 4: Results from RST rate limiting idle scan implementation.

Host	Mean	Std. dev.	Min.	Max
FreeBSD	300	0	300	300
Linux 2.6	126.9	6.3	111	138
Windows XP	460.3	11.9	435	477
Not live	109.1	7.4	96	123

Table 5: Results from SYN cache idle scan implementation for liveness and operating system.

uses a simple buffer for the SYN cache, with between 128 and 1024 entries depending on the memory available on the system. Our Linux kernel 2.4 virtual machine zombie had a SYN cache size of 128, but Linux enforces a rule that only three fourths of the SYN cache can contain SYN packets from hosts that have not demonstrated their liveness in the recent past by completing a fully open TCP connection. This effectively reduces the SYN cache size to 97. We did not enable SYN cookies, which are disabled by default in Linux. The attack works basically the same whether or not SYN cookies are enabled. We ran two separate sets of experiments for the SYN cache idle scan implementation, one to demonstrate that it is possible for the attacker to detect the presence of live machines and perform a rudimentary form of operating system detection, and another to demonstrate that under certain circumstances the attacker can infer the port status of a target port on a particular victim IP address. For all experiments, 100 data points were generated for both open and closed port scenarios.

For checking for liveness, we scanned four different IP addresses. One is a default FreeBSD 7.1.1 machine (with the pf host-based firewall disabled, as is the default), another is a Windows XP machine with no service packs (with Windows firewall disabled, as is the default), and a third is a Linux 2.6 machine (CentOS 5.2, with iptables enabled, as is the default). The fourth IP address has no live host so all packets are simply dropped. Forging packets from random return IP addresses on these victims is very likely to send SYN/ACKs to closed or filtered ports, so we choose random return ports for all forged SYNs where the attacker uses the victim as the return IP address. Varying this return port number is important because if the return port is not different then the forged SYNs will have the same IP addresses and ports for both the destination and source and the SYN cache will drop such duplicates. Both RSTs and ICMP errors cause their corresponding entries to be removed from the SYN cache when received by the zombie.

Because Linux responds to SYN/ACKs on filtered ports at a very low rate (about 10 per second) with ICMP host prohibited packets, FreeBSD responds to SYN/ACKs on closed ports at a rate of at most 200 per second, Windows responds on closed ports with RSTs at an unlimited rate—and IP addresses without live hosts simply cause SYN/ACKs to be dropped—it is possible for the attacker to idle scan a subnetwork and infer something about the operating systems that the live hosts discovered have installed. To scan a single IP address, our implementation sends 50 forged SYNs (that appear to be from the victim), then 50 each of forged SYNs and SYNs where the attacker uses their own return IP address, and then 200 more forged SYNs, all at a rate of 1000 per second. It then sends 200 each of forged SYNs and SYNs where the attacker uses their own return IP address at a rate of 400 per second. The number of SYNs where the attacker uses their own return IP address and receives a SYN/ACK response can then be used to infer the liveness and operating system of the IP address. The results from this experiment are shown in Table 5, where the results are based on the number of SYN/ACKs the attacker receives (note that for Linux 2.4 network stacks SYN/ACKs are retransmitted five times until they time out after 190 seconds).

Under certain circumstances, it is also possible to port scan specific ports on a particular IP address using

Port status	Mean	Std. dev.	Min.	Max
Open	262.1	41.8	120	447
Closed	218.0	39.3	68	318

Table 6: Results from SYN cache idle scan implementation for port scanning FreeBSD.

Port status	Mean	Std. dev.	Min.	Max
Open	482.4	3.3	474	489
Closed	427.8	3.4	417	435

Table 7: Results from SYN cache idle scan implementation for port scanning Linux.

a SYN cache-based idle scan. Specifically, if the response or rates differ for open *vs.* closed or filtered ports on the victim then scanning a target port is possible. Examples of this are FreeBSD with the pf host-based firewall disabled, where open ports and closed ports are rate-limited separately, or Linux hosts with the iptables host-based firewall enabled and an open port that does not use the stateful module of iptables.

To test the FreeBSD example, we developed a SYN cache-based idle scan that simultaneously sends 20000 forged SYN packets (with random return ports that are closed on the zombie) as quickly as possible while sending, at half the rate, alternating forged SYNs with the target port on the victim as the source port and valid SYNs with the return address of the attacker. Because closed ports on the victim are rate limited due to the forged SYNs with random return ports coming from the zombie, the forged SYNs with the target port on the victim as their return port will quickly fill the SYN cache if the target port is also closed and cause fewer entries to be free for non-forged attacker SYNs, therefore causing the attacker to see fewer SYN/ACKs in response. If the target port is open, the open port sends more RSTs before rate limiting begins meaning that more SYN cache entries remain free and the attacker sees more SYN/ACKs. The results of this experiment are shown in Table 6, where the results are based on the number of SYN/ACKs the attacker receives. Some data points for both closed and open ports were thrown out due to failures of the Python pcap library at high packet rates. Packet loss due to the high rates could only make the distributions more similar, not less, because more packets are sent over the TUN/TAP interface for the open port scenario. Thus, the distributions for open and closed ports are clearly different. A two-sampled, unpooled t-test (which assumes neither known variances nor equal variances) for these two sets of data gives a t score of 7.71 with 197 degrees of freedom, which corresponds with a p-score of 0.999999999999696 meaning that a null hypothesis that the two distributions have an equal mean is rejected with very high confidence.

For port scanning Linux-based victims, the idle scan first sends 96 filler SYNs to fill all but one entry in the SYN cache. SYN/ACK replies to filler SYNs are not counted in the results. Then it alternates, at an overall rate of 100 packets per second, forged SYNs with the return IP address of the victim and return port of the target port, filler SYNs, and probe SYNs. Table 7 shows the results of these experiments, where the results reflect the number of SYN/ACK responses to probe SYNs.

4.2.4 Stealth and efficiency

Our idle scan implementations in this section are intended to show that the abstract counterexamples that resulted from our modeling effort were real divergences in real network stacks that could be exploited by the attacker for idle scans. Since the divergences are based on rates in real network stacks we used hypothesis testing to show this. We only report a t-score and p-score for one set of experiments (the SYN cache idle scan implementation for port scanning FreeBSD) because the distributions of the results for other experiments were so different that their high t-scores led to p-scores that were within floating point rounding error of 1.0. Our implementations of these idle scans were designed for this hypothesis testing and therefore are not optimized for attacker stealth or efficiency in carrying out the scan. For assessing the practicality of these idle scan techniques, we will now comment on stealth and efficiency.

For the RST rate limiting idle scan, the attacker cannot perform the idle scan without sending more than 200 SYN/ACKs to the zombie either directly or indirectly. However, the attacker need not send SYNs to the victim (forged from the zombie) at half this rate. It is possible to, *e.g.*, send SYNs to the victim at a rate of 20 per second and send SYN/ACKs (or any packet that will elicit a RST) to the zombie at 195 per second. Theoretically, the mutual information between the victim port status and the sequence of packets the attacker sees is non-zero even if the attacker sends only a single forged SYN to the victim, and even when packet loss is accounted for. Thus the attacker has a fair amount of flexibility in terms of trading off speed of the scan *vs.* stealth for packets it sends to the victim. Furthermore, sending SYNs simultaneously to multiple victims and multiple ports and measuring the zombie responses in the aggregate can increase the efficiency of the scan if the distribution of expected closed *vs.* open victim ports diverges from an equal distribution. To see this, consider the extreme case where a large subnetwork has only a single host with an open port, something similar to a binary search could greatly reduce the amount of time necessary for the scan in this case.

For the SYN cache idle scans, which are much more powerful in terms of the new capabilities they offer attackers beyond the currently known idle scan technique, there is a wider range of efficiency and stealth tradeoffs that the attacker can make. Furthermore, unlike ICMP IPID- or RST rate limit-based idle scans, virtually any modern network stack that offers any type of protection against SYN flooding can be used as a zombie. We chose to use a low-memory Linux kernel 2.4-based zombie for our experiments due to its simplicity and small SYN cache size, but larger SYN cache sizes or more complex SYN cache implementations are also easily exploited for SYN cache idle scans. The SYN cache only needs to be almost full for SYN cache idle scans to work, and SYNs for half-open connections take 190 seconds to timeout in Linux by default. So even for high-memory Linux 2.4 machines with 1024 SYN cache entries (of which 769 are used, compared to 97 for 128-entry SYN caches), the rate necessary to create the conditions for an idle scan only increases from 0.5 SYNs per second from the attacker to the zombie to about 4.1 SYNs per second (these rates keep the buffer almost full despite the timeouts). Once these conditions are created, the attacker effectively can do a SYN/ACK scan of the victim host or network at the cost of two packets sent per SYN/ACK query and three more generated as responses. It also does not matter whether or not the zombie implements SYN cookies, since SYN cookies are never retransmitted (compared to typically three to five retransmissions of regular SYN/ACKs for various zombie configurations) and also have easily identifiable statistical anomalies in their initial sequence numbers.

Some SYN cache implementations that are not simple buffers like Linux 2.4 may make SYN-cache idle scans slightly more difficult, but still possible and relatively efficient. For example, the FreeBSD SYN cache implementation [19] uses a SYN cache with 512 buckets that each have 30 entries and are chosen uniformly at random using a hash of the IP address/port pairs and a host-generated secret. This mechanism is designed to stop denial-of-service, not idle scans. It creates some equivocations that can reduce the amount of information flow the attacker can exploit for idle scans but the attacker can still perform the scan relatively efficiently even with FreeBSD zombies. We have not explored the SYN cache implementations of Linux kernel 2.6 or Windows hosts, but all modern network stacks must have some form of SYN cache for reliability purposes and a limit on this resource to prevent denial-of-service. Thus, only by making this resource non-shared is non-interference to prevent idle scans possible, and the current OSI network stack model with TCP/IP does not make the necessary trust distinctions to split the SYN cache. Thus, virtually every end host machine that the attacker can route to at least one open port on is a potential zombie.

The rate at which the attacker must send packets to the zombie for a SYN cache idle scan, and therefore the stealth of the scan, depends on the attacker's goals. If the zombie is a Linux kernel 2.4 machine and the attacker wants only to check the liveness of a range of IP addresses on the victim network, then between 0.5 and 4.1 packets per second plus the probes themselves is sufficient. Note that, in terms of stealth, it is also relevant that the attacker need not send any packets to the victim for this form of idle scan, not even packets with forged return addresses.

For detecting the operating system of a victim host or scanning individual ports on the victim, higher rates are necessary. Detecting a Linux machine on the victim network and port scanning it can easily be done at between 10 and 20 packets per second. We also discovered during our experiments that, at least for Linux kernel 2.4 hosts, it is easy for the attacker to not only remove their own packets from the SYN cache manually, but any packet that they have fogred, using forged RSTs. This is because only the IP address and port pairs are checked, the sequence and acknowledgment numbers for RSTs are ignored when deciding whether to drop an entry from the SYN cache on the zombie. Thus, the attacker has a high degree of control over the SYN cache status of the victim. Packet delay, packet loss, and interference from other machines that contact the zombie can easily be accounted for in this way, and the aggregate effect of scanning multiple victims at a time mentioned above also applies to SYN cache idle scans. Thus, while it is not impossible for certain network configurations to defend against SYN cache idle scans with, *e.g.*, ingress filtering rules that prevent packets with return addresses forged for hosts inside the firewall domain, special attention must be paid to how network intrusion detection systems can deal with this new form of idle scan. In future work we intend to model the capabilities of this attack as a Markov Decision Process and discern tight bounds on numbers of packets and rates needed for different attacker goals.

4.3 Ensuring non-interference using the SAL model checker

Based on our experimental results from implementing the two counterexamples as idle scan attacks, it is apparent that RST rate limiting and the SYN cache interact in complex ways and cannot be considered separately. Thus we chose to leave RST rate limiting in the model for verifying non-interference of the split SYN cache.

It is well-known that verifying properties using a model checker is much more difficult than finding a counterexample. We abstracted the model down to the simplest form that produces both counterexamples, and attempted to prove the noninterference property for cases where the shared, limited resources were split based on trust relationships and therefore no longer shared. The zombie and victim consider each other trusted and the attacker untrusted. For the RST rate limiting counterexample, the hosts have separate `RST_count` RST counters for trusted *vs.* untrusted hosts, and the SYN cache is removed. For the SYN cache counterexample, we implemented a split SYN cache structure with separate SYN cache buffers for trusted *vs.* untrusted hosts.

In the first case, we removed the SYN cache and focused only on the RST rate limitation counter example. Since symbolic model checkers are known to be better for verifying properties in contrast to explicit state enumeration based bounded-model checkers, we used SAL's symbolic model checker. It verified the property that:

$$\vdash \mathrm{G}(\mathrm{PacketA1} = \mathrm{PacketA2})$$

This verification completed in a little over 5 minutes.

Encouraged by this result, we introduced the SYN cache back into the model. The symbolic model checker ran out of memory on a machine with 16GB of memory after three days. We then ran the bounded model checker up to depth 1000 (to mean that all sequences of transitions of length ≤ 1000 are checked for counterexample), and the model checker did not report any counterexample, which is very encouraging. This means that the attacker cannot violate non-interference with any idle scan where less than 1000 transitions occur. The SYN cache counterexample to our shared SYN cache implementation required only 5 transitions. Informally, this result means that there exists no attack, even with only a single entry in the SYN cache, where the attacker can violate non-interference with 1000 or fewer packets being generated by the attacker or by the zombie and victim's responses. Currently, we are exploring alternatives to symbolic model checking for the split SYN cache, including verifying the property through k-induction [6]. We are also considering attempting a proof by induction on an induction-based theorem prover such as ACL2 or RRL.

5 Concluding remarks and future work

We modeled idle scans for modern network stacks using transition systems and analyzed them using model checking. This modeling effort led to the discovery of two new forms of idle scan, each of which was associated with a shared, limited resource. Our results demonstrate that non-interference for network protocol stacks warrants further study. We discovered two new forms of idle scan, one of which gives the attacker capabilities that no current attacker port scanning capabilities below layer 7 (the application layer) provide. We demonstrated in this paper that it is possible for an attacker to port scan a network from outside the firewall on a port that the firewall blocks, for example. We also showed that this form of idle scan, based on SYN caches, can be used for a rudimentary form of operating system detection. In light of these results, a more formal treatment of information flow in networks is needed so that we can develop a variety of defenses, both for existing networks and in future protocol designs.

We discussed the stealth and efficiency of the idle scans in Section 4.2.4. While it is clear both that the attacks are practical and that certain defenses exist in some situations, a more thorough treatment of possible scans and defenses to detect or eliminate them is needed. By modeling idle scans as a Markov Decision Process, it will be possible to explore this space more thoroughly and find boundaries in terms of packet rates.

Using SAL's model checkers, we were able to identify counterexamples to non-interference, in the form of idle scans, from our formal model of a network stack as a transition system. After fixing the model by splitting limited resources and separating them for trusted *vs.* untrusted hosts we are able to verify the noninterference property for the RST rate limit case. However, we were only able to show the noninterference property with RST rate limiting and a SYN cache up to 1000 transitions. Verifying the noninterference property for this more general fix using a model checker remains a challenge. We plan to investigate induction methods for this.

Our model of network stacks was at the level of abstraction of sequences of packets. A richer model that includes memory usage, packet loss, and packet delay would likely produce more counterexamples to the non-interference property for idle scans. Thus we propose that trust relationships be made explicit all the way down to the IP layer in future protocol designs. Because all resources are inherently limited, giving protocol implementations a mechanism that can help divide these resources and remove sharedness is the only way to address the advanced network reconnaissance attacks of the future. Our results in Section 4.3 demonstrated that non-interference, which effectively eliminates idle scans, is achievable by statically dividing resources based on trust relationships.

References

[1] Antirez. new tcp scan method. Posted to the bugtraq mailing list, 18 December 1998.

[2] Steven M. Bellovin. A technique for counting natted hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 267–272, New York, NY, USA, 2002. ACM.

[3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[4] D. J. Bernstien. SYN Cookies. `http://cr.yp.to/syncookies.html`.

[5] Yunja Choi. From nusmv to spin: Experiences with model checking flight guidance systems. *Form. Methods Syst. Des.*, 30(3):199–216, 2007.

[6] Leonardo de Moura. SAL tutorial. Csl technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2004. Available at `http://sal.csl.sri.com/doc/salenv_tutorial.pdf`.

[7] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[8] Sagar Chaki Edmund, Edmund M. Clarke, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag, 2004.

[9] Carrie Gates. *Co-ordinated port scans: a model, a detector and an evaluation methodology*. PhD thesis, Dalhousie Univ., Halifax, Canada, Canada, 2006.

[10] Carrie Gates. Coordinated scan detection. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 09)*, February 2009.

[11] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[12] Guofei Gu, Monirul Sharif, Xinzhou Qin, David Dagon, Wenke Lee, and George Riley. Worm detection, early warning and response based on local victim information. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Hazem Hamed, Ehab Al-Shaer, and Will Marrero. Modeling and verification of ipsec and vpn security policies. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pages 259–278, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[15] Min Gyung Kang, Juan Caballero, and Dawn Song. Distributed evasive scan techniques and countermeasures. In *DIMVA '07: Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 157–174, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*, 1(3):256–277, 1983.

[17] Tadayoshi Kohno, Andre Broido, and K. C. Claffy. Remote Physical Device Fingerprinting. *IEEE Symposium on Security and Privacy*, May 2005.

[18] C. Leckie and R. Kotagiri. A probabilistic approach to detecting network scans. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 359–372, 2002.

[19] Jonathan Lemon. Resisting SYN flood DoS attacks with a SYN cache. `http://people.freebsd.org/~jlemon/papers/syncache.pdf`.

[20] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[21] Stephen McCamant and Michael D. Ernst. A simulation-based proof technique for dynamic information flow. In *PLAS 2007: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, USA, June 14, 2007.

[22] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[23] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of internet background radiation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 27–40, New York, NY, USA, 2004. ACM.

[24] SAL project. Examples. `http://sal.csl.sri.com/examples.shtml`.

[25] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2000. IEEE Computer Society.

[26] John Rushby. SAL tutorial: The Needham-Schroeder protocol in SAL. Csl technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003. Available at `http://www.csl.sri.com/users/rushby/abstracts/needham03`.

[27] John Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Csl technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2004. Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`.

[28] Stuart Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast Detection of Scanning Worm Infections. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, French Riviera, France, September 2004.

[29] Avinash Sridharan, T. Ye, and Supratik Bhattacharyya. Connectionless port scan detection on the backbone. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 10 pp.–576, April 2006.

[30] Avinash Sridharan and Tao Ye. Tracking port scanners on the ip backbone. In *LSAD '07: Proceedings of the 2007 workshop on Large scale attack defense*, pages 137–144, New York, NY, USA, 2007. ACM.

[31] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *J. Comput. Secur.*, 10(1-2):105–136, 2002.

[32] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, 1986.

[33] VirtualBox. `http://www.virtualbox.org/`.

[34] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[35] David Whyte, Evangelos Kranakis, and P. C. van Oorschot. Dns-based detection of scanning worms in an enterprise network. In *In proc. of the 12th annual Network and Distributed System Security symposium*, pages 181–195, 2005.

[36] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[37] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Networked Systems Design and Implementation (NSDI)*, 2007.