Blowtorch: a Framework for Firewall Test Automation

Daniel Hoffman
University of Victoria
Department of Computer Science
PO Box 3055 STN CSC
Victoria, BC Canada V8N 4C9
dhoffman@cs.uvic.ca

Kevin Yoo
University of Victoria
Department of Computer Science
PO Box 3055 STN CSC
Victoria, BC Canada V8N 4C9
kyoo@uvic.ca

ABSTRACT

Firewalls play a crucial role in network security. Experience has shown that the development of firewall rule sets is complex and error prone. Rule set errors can be costly, by allowing damaging traffic in or by blocking legitimate traffic and causing essential applications to fail. Consequently, firewall testing is extremely important. Unfortunately, it is also hard and there is little tool support available.

Blowtorch is a C++ framework for firewall test generation. The central construct is the packet iterator: an event-driven generator of timestamped packet streams. Blowtorch supports the development of packet iterators with a library for packet header creation and parsing, a transmit scheduler for multiplexing of multiple packet streams, and a receive monitor for demultiplexing of arriving packet streams. The framework provides iterators which generate packet streams using covering arrays, production grammars, and replay of captured TCP traffic. Blowtorch has been used to develop tests for industrial firewalls, placed between an IT network and a process control network.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Security,Verification

Keywords

automated testing, production grammar, covering array, capture/replay, network firewall

1. INTRODUCTION

The rule sets used to configure firewalls are based primarily on a single language construct: condition/action lists evaluated like a C switch statement. The rule set is applied to each packet arriving at a firewall port. The conditions are boolean expressions on the packet header field values or on firewall state, which is usually a table with an entry for each active network connection. The actions are typically "accept," "drop," or "log." There are no variables, loops, or function calls. Despite the simplicity of the language, commercial firewall rule sets are complex and extremely error prone [14]. Because firewall rule sets are complex programs, they need to be thoroughly tested. Unfortunately, developing firewall tests is difficult and there is little tool support available.

Blowtorch is a C++ framework designed for testing firewall rule sets in the process control environment [3], where the cost of failure is high and extensive testing is justified. Most firewall testing today is based on vulnerability assessments, e.g., port scans, intended to determine the vulnerability of a protected network to attacks through the firewall. In contrast, Blowtorch tests the firewall in isolation, connected only to test equipment. The tests are fully automated, including output checking, and are repeatable at negligible cost. Both kinds of tests are valuable; at present, there is much better tool support for vulnerability assessments.

While the target domain of this paper is firewall testing, the techniques are drawn from classic approaches in test tools. A framework is supplied to factor out the common aspects, significantly reducing the cost of test development. Tests are generated using production grammars, covering arrays, and capture/replay. Considerable adaptation was required, however, to apply these techniques to firewall testing. For example, packet capture/replay is fundamentally different from GUI capture/replay.

Section 2 presents an architecture and security policy for a firewall configured for use in a process control network. Section 3 presents the Blowtorch framework and Section 4 describes tests developed using the framework. Section 5 presents related work.

2. FIREWALL ARCHITECTURE

This section presents an architecture and security policy designed to separate hosts on the Business Network (BN) from those on the Process Control Network (PCN). While the architecture and policy are consistent with current best practices for PCNs [3], they have been simplified for the purposes of this paper. In practice, there will be more servers and more applications than shown.

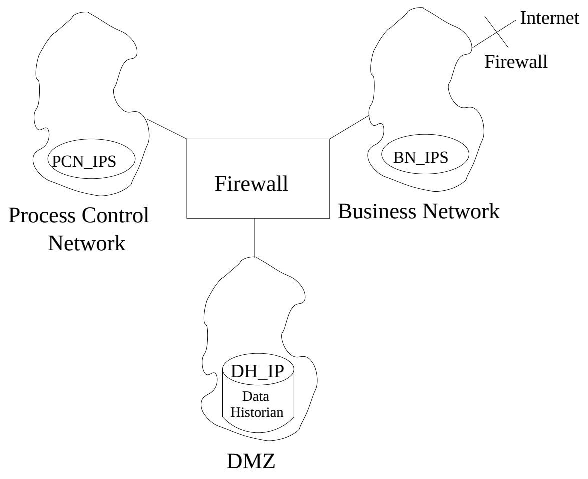

Figure 1: A Process Control Network firewall architecture

2.1 Firewall Architecture

Figure 1 shows a 3-zone firewall, similar to those used in IT networks. The demilitarized zone (DMZ) contains servers to be shared by the PCN and the BN. The DMZ contains a Data Historian: a specialized database server used to store sensor values from the PCN.

The PCN typically contains thousands of sensors and actuators, controlled by tens or hundreds of programmable logic controllers (PLCs). The PLCs, in turn, are monitored and controlled by human-machine interfaces (HMIs), typically PCs running specialized GUIs displaying virtual gauges and switches. The HMIs log sensor values to the Data Historian.

The BN contains tens or hundreds of hosts, mostly office PCs. These PCs read sensor values from the Data Historian. The BN is connected to the PCN/DMZ firewall, and also to the Internet through another firewall, which is not the topic of this paper.

2.2 Firewall Security Policy

While the firewall architecture in Figure 1 is similar to typical IT architectures, the traffic patterns and security concerns are quite different. On the PCN/DMZ path, the traffic rate is modest but delay or loss can be costly. Because the sensor data must be timestamped by the Data Historian, packet delays will cause timestamp inaccuracies. On some PCNs, e.g., those used for drug manufacture, complete and accurate sensor data is required by law. Even when not legally required, loss of sensor data is considered serious. On the BN/DMZ path, the traffic rates may be higher but delays or temporary outages are problematic but much less serious.

Because the BN is connected to the Internet, the PCN is as well, albeit indirectly. Attacks on the BN can be costly, like attacks on other IT networks. A successful attack on the PCN, however, can be far more costly, resulting in loss of product, damage to equipment, or injury or loss of life. Modern PLCs often use Ethernet and TCP/IP but usually do not have hardened TCP/IP stacks, and are known to be vulnerable to standard Internet attacks [6].

A security policy to (partially) address these concerns is as follows:

- On the PCN/DMZ path, allow only hosts in PCN_IPS (see Figure 1) to access the Data Historian. Allow only inbound TCP connections to DH_IP.
- Similarly, on the BN/DMZ path, allow only hosts in BN_IPS to access the Data Historian. Allow only inbound TCP connections to DH_IP.
- Drop all traffic on the PCN/BN path.
- Provide SYN flood protection using rate limiting on the BN firewall port, to protect the PCN from this DoS attack.
- On all TCP connections, drop all packets with bad TCP flag combinations, e.g., SYN plus FIN.

2.3 Four Firewall Tests

In the sections to follow, we will focus on four tests for this firewall security policy:

1. *SYN flood on BN firewall port.* Check that, if TCP SYN packets are sent at a higher rate than permitted, the accepted packets do not exceed the maximum rate specified by the policy.
2. *Data Historian Connections.* Let a *connection identifier* be the four-tuple:

   $\langle$ *srcIPaddr*, *srcTCPport*, *dstIPaddr*, *dstTCPport* $\rangle$

   and let SRC_PORTS be a set of legal source ports. Check that PCN/DMZ connections are permitted with every connection identifier in

   $\langle$ PCN_IPS $\times$ SRC_PORTS $\times$ DH_IP $\times$ DH_PORT $\rangle$

   Let PCN_IPS_XC be set of IP addresses disjoint with PCN_IPS. Check that PCN/DMZ connections are blocked with every connection identifier in

   $\langle$ PCN_IP_XC $\times$ SRC_PORTS $\times$ DH_IP $\times$ DH_PORT $\rangle$

   Perform the analogous tests for the BN/DMZ path.
3. *Bad TCP flags.* Drop all packets with illegal TCP flag combinations.
4. *Replay of production traffic.* Capture traffic from actual PCN/DMZ and BN/DMZ TCP connections. Check that the firewall allows this traffic to be replayed.

3. THE BLOWTORCH TEST FRAMEWORK

Stream (TCP) and datagram (UDP) sockets are in common use in distributed computing applications. Unfortunately, they are inadequate for firewall testing. For many tests, raw sockets are required, which provide complete control over header fields and packet timing. While raw sockets provide excellent control, they require far more coding effort to use. Tool support is required, offering at least:

- *Control over all header fields.* To generate tests for bad TCP flag combinations, for example, you must create TCP headers with flag combinations which would never appear in stream or datagram socket packets.
- *Control of packet timing.* To test a firewall for SYN flood protection, you might need to generate 500 TCP SYN packets at 50 packets per second. Attempting to open 50 stream sockets per second would not produce the desired result. The first TCP packet in each connection might be sent immediately after the socket is opened, or some time later. Further, most hosts are limited to 100 or so open stream sockets.
- *Support for handshaking.* So called "stateful filtering rules" track request/response exchanges. A typical rule will drop a response packet from a server unless it is preceded by a request packet from the client. Consider a TCP SYN (request) from a PCN client to a DMZ server and the TCP SYN-ACK (response) packet from the server. For testing, a SYN packet must be sent to the firewall PCN port. The test code must wait until the firewall forwards that packet, and it is seen on the firewall DMZ port, before sending the SYN-ACK packet to the firewall DMZ port. If the SYN-ACK is sent before the firewall has handled the SYN, the firewall will (correctly) drop the SYN-ACK, but the test code will report an error.
- *Support for traffic capture and replay.* Production traffic streams often contain complex packet exchanges which are difficult to generate synthetically. There are huge benefits to tools which can capture the traffic—relatively easy—and replay it—much more difficult.
- *Easy test creation, execution and maintenance.* Network administrators are typically smart but rarely get the large blocks of uninterrupted time required for serious software development. Tools are needed which factor out the repetitive tasks and allow development of useful tests in a page or two of code.

3.1 Framework Overview

The central construct in the Blowtorch framework is the packet iterator. A `PacketIter` object provides a stream of packets. As shown in Figure 2, iterators $I_0, I_1, \ldots, I_{N-1}$ provide packets to the `PacketScheduler` which enqueues the streams. `PacketTransmit` dequeues the packets, passing them to the appropriate network interface. Each network interface is encapsulated as a `PacketIO` object.

Blowtorch monitors the `PacketIO` objects for packet arrivals. When a packet arrives, it is passed to the `PacketIter` object that generated it, providing support for handshake tests.

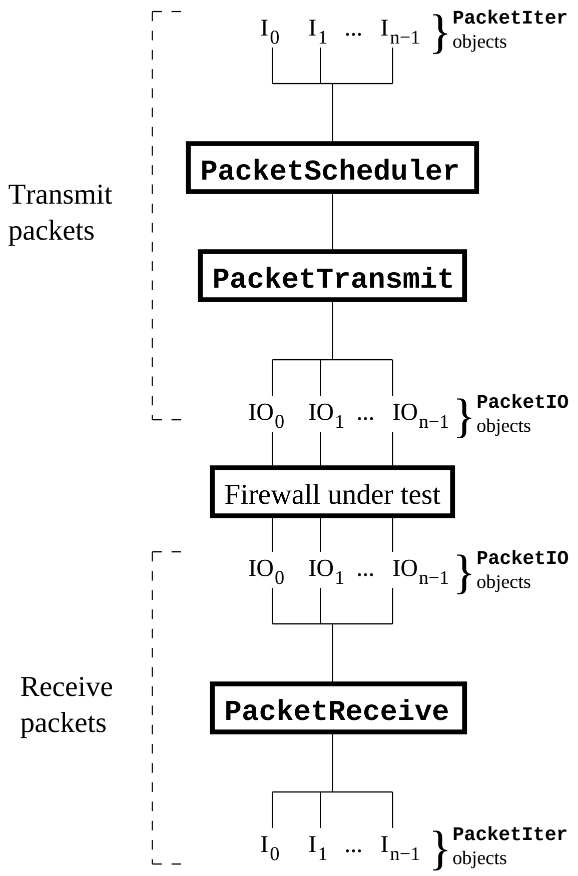

Figure 2: Blowtorch packet flow

3.2 Packet Generation and Parsing

Each `Packet` object contains a buffer, an I/O identifier, and a timestamp. For a packet to be transmitted, the buffer contains the raw packet, the I/O identifier specifies the `PacketIO` object to be used for transmission, and the timestamp is the requested transmission time. For a received packet, the buffer contains the raw packet, the I/O identifier specifies the `PacketIO` object which handled the arriving packet, and the timestamp is the time at which the packet was received.

By using raw sockets, Blowtorch provides direct control over all packet header fields. Unfortunately, raw sockets are much harder to use than stream or datagram sockets: the programmer must explicitly assign every header field, and deal with header layouts, byte swapping, and checksum calculations. Blowtorch provides a library, the Raw Socket Toolkit, which hides these details and makes it easy to generate and parse packets with raw sockets.

3.3 Packet Iterators

`PacketIter` is an abstract class containing three methods:

1. `next` returns the next packet for transmission, if available.
2. `hasNext` returns `READY` if the iterator has a packet ready for transmission, `SUSPENDED` if the iterator does not currently have a packet ready for transmission, and `TERMINATED` if the iterator has already provided all of its packets.
3. `notify` is a callback method, invoked by the framework to pass an arriving packet to the iterator.

`PacketIterId` is an abstract class containing two methods:

1. `getIterId` takes a `Packet` object and extracts an identifier specifying one of the iterators. Usually, `getIterId` is implemented to specify the iterator which originally provided the packet for transmission, or -1 if the packet was not provided by any of the iterators.
2. `notify` is a callback method invoked when `getIterId` returns -1. In some tests `notify` is used to display error messages for unexpected packets. In other tests, it is used for housekeeping tasks, e.g., detecting an ARP request and generating an ARP reply.

3.4 Packet I/O

`PacketIO` is an abstract class providing two methods: `send` and `receive`. Four `PacketIO` objects are currently provided by Blowtorch:

1. `RawSocketIO` uses the raw socket service.
2. `TapSocketIO` uses Ethernet taps, an Ethernet driver without an associated Ethernet card. A tap sends packets to a buffer which can be read by a test program, and receives packets from a second buffer which can be written by a test program. Most Blowtorch tests require two or three network interfaces; a three interface test can be developed on a machine with no Ethernet hardware by using `TapSocketIO`.
3. `NullSocketIO` is the Blowtorch equivalent of `/dev/null`: calls to `send` and `receive` are always legal but have no effect.
4. `GatewaySocketIO` provides more accurate transmission timing using an off-the-shelf home gateway connected to the PC by Ethernet. Packets are sent from the PC immediately, regardless of the timestamp value. The gateway, running custom software developed with MicroC/OS [8], buffers the packets, sending them according to the timestamp value. Our experiments have shown that `GatewaySocketIO` can achieve transmission accuracy within about 50 microseconds, where `RawSocketIO` accuracy is typically about 15 milliseconds.

With network I/O encapsulated in this way, it is easy to switch between network interfaces during test development.

3.5 Timestamp Generation

The `TimeIter` abstract class supports the generation of packet timestamps. Two methods are provided: `next` returns the next timestamp, which must be larger than the previous one, and `hasNext` returns true until the last timestamp has been retrieved. `TimeIter` objects are usually passed to `PacketIter` constructors. Blowtorch provides three `TimeIter` objects:

- `FixedTimeIter` returns timestamps with specified start value and delta, e.g., $1.0, 1.1, 1.2, \ldots$.
- `BurstyTimeIter` returns timestamps with specified start time, burst length, inter-packet gap and inter-burst gap, e.g., $0.0, 0.1, 0.2, 1.0, 1.1, 1.2, \ldots$.
- `FileTimeIter` retrieves timestamps from a file. Typically, the file contents are either generated from a statistical distribution or extracted from a captured packet trace.

3.6 All Together Now

With the main framework classes in hand, it is helpful to revisit Figure 2, focusing on the tester's view. The tester must provide three entities:

- `packetIOs`: a vector of `PacketIO` objects.
- `packetIters`: a vector of `PacketIter` objects. The I/O identifier contained in each returned packet is interpreted as an index into `packetIOs`.
- `packetIterId`: a `PacketIterId` object. The iterator identifier returned by `getIterId` is interpreted as an index into `packetIters`.

Blowtorch then repeatedly invokes the main Blowtorch objects: `PacketScheduler`, `PacketTransmit`, and `PacketReceive`:

- `PacketScheduler` polls the iterators, calling `next` whenever `hasNext` returns `READY`. Packets from the iterators are merged in an internal queue. Based on a constructor parameter, either earliest-deadline-first or earliest completion-time-first scheduling is used. The goal is to minimize the total difference between the timestamps returned by the iterators and the actual transmission times.
- `PacketTransmit` dequeues the scheduled packets, sending each packet to the `PacketIO` object indicated by the I/O identifier at the time indicated by the timestamp.
- `PacketReceive` polls the `PacketIO` objects in `packetIOs` calling `packetIterId.getIterId` to compute the iterator identifier, i. If $i \in [0, \texttt{packetIters.size}() - 1]$ then `packetIters[`i`].notify` is invoked; otherwise `packetIterId.notify` is invoked.

4. FOUR TESTS REVISITED

This section presents Blowtorch implementations of the four tests described in Section 2. For simplicity, each test is presented below "standalone," i.e., the `packetIters` vector has a single element.

4.1 A SYN Flood Iterator

In this test, the firewall is configured with a rate limiting rule, intended to prevent more than 10 TCP SYN packets per second from being forwarded. The test will send 1000 TCP SYN packets at 100 packets per second. While only 100 of the packets should be forwarded, to allow for timing variations the test will check that no more than 110 get thorough.

The test is implemented with the `SynFloodPacketIter` and `SynFloodPacketIterId` classes. In `SynFloodPacketIter`:

- The constructor takes three parameters: (1) a starting connection id, (2) the I/O identifier of the client `PacketIO` object, and (3) a `FixedTimeIter` object, with start timestamp = 0 and delta = 10 milliseconds. The constructor uses the Raw Socket Toolkit to build a TCP packet based on the constructor parameters.
- `hasNext` returns `READY` until 1000 packets have been retrieved and returns `TERMINATED` thereafter.
- `next` increments the source TCP port to avoid duplicate connection identifiers, fetches the next timestamp from the `FixedTimeIter` object, and invokes a Raw Socket Toolkit function to do byte swaps and to compute the IP and TCP checksums.
- `notify` records the total number of SYN packets received in `synTotal`.
- The destructor reports an error if `synTotal` < 110.

In `SynFloodPacketIterId`, the `getIterId` method returns 0 if the packet is a TCP SYN packet and returns -1 otherwise. If the `notify` method receives an ARP request, it generates an ARP reply; otherwise it generates an "unexpected packet" message.

4.2 A Generic TCP Iterator

The `TCPIter` class helps the tester to create legitimate TCP sessions. By legitimate, we mean that `TCPIter` is sending TCP segments that perform correct handshaking and use proper header field values. For example, the sequence and acknowledgement numbers in each TCP segment are properly set.

The sequence of TCP segments are specified abstractly in an input file containing one or more abstract frames. An abstract frame is a nine-tuple:

host action urg ack psh rst syn fin len

Each TCP session involves two hosts: a client, the host that initiates the session, and a server, the host that receives the initiation request; *host* indicates which of these two hosts the abstract frame belongs to. *Action* enables `TCPIter` to enforce proper handshaking; it can take one of two values: *tx*, specifying that `TCPIter` will send this frame onto the network, and `rx`, indicating that `TCPIter` will be suspended until the specified frame is received. The last seven parameters of the abstract frame represent fields in the TCP header. *Urg, ack, psh, rst, syn,* and *fin* are the TCP control flags, while *len* specifies the length of the TCP payload.

The 3-way handshake is executed at the beginning of every TCP session to perform connection initialization [13]. Here are the abstract frames that constitute the 3-way handshake:

host	*action*	*urg*	*ack*	*psh*	*rst*	*syn*	*fin*	*len*
`client`	`tx`	0	0	0	0	1	0	0
`server`	`rx`	0	0	0	0	1	0	0
`server`	`tx`	0	1	0	0	1	0	0
`client`	`rx`	0	1	0	0	1	0	0
`client`	`tx`	0	1	0	0	0	0	1
`server`	`rx`	0	1	0	0	0	0	1

The first abstract frame represents the transmission of the SYN packet on the client side. The next abstract frame is the reception of this SYN packet on the server side; the subsequent abstract frames will not be evaluated until this SYN packet is received at the server, ensuring that proper handshaking must occur during the creation of this session.

4.3 Data Historian Connections

In this test, we need to create full TCP connections for two sets of connection identifiers. In NC_IDS, the source IP address is taken from PCN_IPS or BN_IPS and the source ports are selected from the temporary ports, i.e., 1024 or larger. The destination IP address is DH_IP and the destination port is the Data Historian server port. The XC_IDS connection identifiers are like those in NC_IDS except that one or more of the addresses and ports is prohibited. All TCP packets in the NC_IDS connections should be forwarded by the firewall while all packets in the XC_IDS connections should be dropped. If either NC_IDS or XC_IDS is large, e.g., uses all the addresses on the subnet, then cost dictates that only a subset of the connection identifiers will be tested. The subset should be selected systematically.

Using `TCPIter`, it is easy to develop the tests by overriding `notify` for `TCPiter` so that it is silent if a received packet has a connection identifier in NC_IDS and issues an error message otherwise. If either NC_IDS or XC_IDS is too large, we produce smaller subsets using covering arrays. Our implementation of Tai's algorithm [12] extracts all pairwise combinations from the full sets, producing substantially smaller subsets.

4.4 Bad TCP Flags

There are six control flags in the TCP header. While a network application can set these flags in any way that it wants, not all flag combinations are meaningful. For example, a TCP packet with both the *syn* and *fin* flags set is not understood by any TCP stack. Of the sixty-four possible combinations of flags, only eighteen are considered meaningful [10, 282]. For this test, we wanted to determine how a firewall handles the forty-six bad flag combinations. Since these flags should never occur in normal network traffic, the desired behavior of a firewall would be to drop all packets containing these bad flag combinations.

For these tests, we wanted to see which bad flag packets are filtered by a firewall, and to determine if the state of the TCP session has any bearing on which packets are forwarded. The state of the TCP session should play a role, because firewalls maintain a connection tracking table that monitors each TCP connection. Therefore, the firewall may handle a bad flag packet differently if it has just seen, for example, a SYN-ACK packet as opposed to a FIN-ACK packet. In order to determine if state makes a difference, we will send the bad flag packet at different times during a TCP session. Each TCP session consists of six packets that are sent between the server and client. Therefore, there are five different places where we can send a bad flag packet, as depicted in Figure 3. We have five places to insert forty-six bad flag packets; this results in two hundred and thirty test cases, where a test case consists of the packets required to build a TCP session along with the one bad flag packet. To create these test cases, a production grammar was used.

A production grammar is defined in the same manner as a traditional parsing grammar; it is composed of a set of rules: non-terminal to terminal mappings. However, the difference between a production grammar and a parsing grammar is that the former is used in reverse; it is used to generate strings in the language that it defines, rather than to parse a string to determine its membership in the language. The power of production grammars lies in their ability to generate a very large set of strings from a relatively small set of rules. For our tests, the strings that are generated are abstract frames, which are then passed to `TCPIter`. The grammar that defines the TCP segments for our tests is depicted in Figure 4.

Our production grammar creates the entire language that it defines; that is, a test case is created for every permutation of every bad flag packet in each of the five locations of the TCP session. For a relatively small test set, this behavior is desirable. However, if the language is much bigger, then the tester may not want all of the different test cases. If the tester wants only a subset of the test cases, then she likely wants some control over how the test cases are being generated. This control can be facilitated through the use of annotations in the grammar.

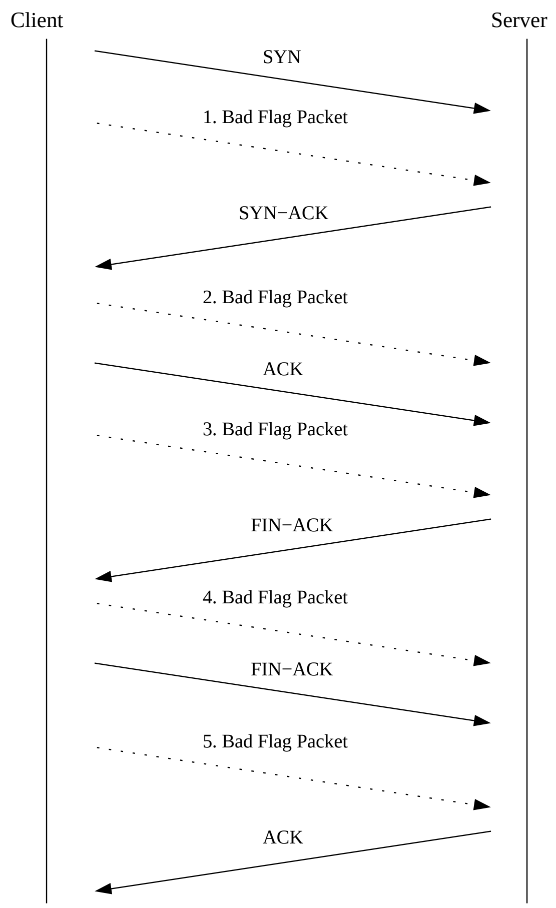

Figure 3: The five insertion points for a bad flag packet

⟨start⟩ ::=
⟨syn⟩ ⟨bfp⟩ ⟨syn-ack⟩ ⟨ack⟩ ⟨fin-ack⟩ ⟨fin-ack⟩ ⟨ack⟩ |
⟨syn⟩ ⟨syn-ack⟩ ⟨bfp⟩ ⟨ack⟩ ⟨fin-ack⟩ ⟨fin-ack⟩ ⟨ack⟩ |
⟨syn⟩ ⟨syn-ack⟩ ⟨ack⟩ ⟨bfp⟩ ⟨fin-ack⟩ ⟨fin-ack⟩ ⟨ack⟩ |
⟨syn⟩ ⟨syn-ack⟩ ⟨ack⟩ ⟨fin-ack⟩ ⟨bfp⟩ ⟨fin-ack⟩ ⟨ack⟩ |
⟨syn⟩ ⟨syn-ack⟩ ⟨ack⟩ ⟨fin-ack⟩ ⟨fin-ack⟩ ⟨bfp⟩ ⟨ack⟩

⟨bfp⟩ ::= ⟨bfp1⟩ | ⟨bfp2⟩ | ... | ⟨bfp46⟩

⟨syn⟩ ::= client tx 0 0 0 0 1 0 0
⟨syn-ack⟩ ::= server tx 0 1 0 0 1 0 0
.
.
.
⟨ack⟩ ::= server tx 0 1 0 0 0 0 0

⟨bfp1⟩ ::= client tx 0 0 0 0 0 0 0
⟨bfp2⟩ ::= client tx 0 0 0 0 0 1 0
.
.
.
⟨bfp46⟩ ::= client tx 1 1 1 1 1 1 0

Figure 4: Bad flag production grammar

With annotations, we provide two ways of giving the tester control over test case generation: a way to limit the total number of test cases created, and a method of assigning probabilistic weights to decisions that are made in the grammar. We created one of these annotated grammars to create TCP sessions that allow multiple bad flag packets within a single TCP session. This new grammar can produce 47^5 distinct test cases, far more than anyone can afford to execute. Therefore, the annotations are not only useful, but often a necessity.

We tested two firewalls: iptables, which is an open source firewall that runs on Linux, and a Cisco Pix, which is the industry standard. Iptables has explicit facilities that deal with bad flag packets, and so none get through inside or outside of a TCP session. The Pix has no such explicit feature, and the results of those tests will now be discussed.

We conducted two tests: one test that simply sent the forty-six bad flag packets through the firewall, and another test that sent bad flag packets within legitimate TCP sessions. For the first test, only two of the forty-six combinations were accepted. For the second test, more packets were indeed forwarded. For example, some bad flag packets with the *ack* flag set now get through, whereas neither of the two bad flag packets that got through in the first test had this flag set. This result makes sense; since legitimate TCP packets cannot have the *ack* flag set outside of a session, we would expect the firewall to easily discard these obviously badly formed packets. Inside a session, however, the process of determining the validity of a packet is not as straightforward. Furthermore, as we suspected, the particular place a packet is sent within a TCP session does influence whether or not a bad flag packet is accepted. For instance, between the two FIN-ACK's, thirty-four of the forty-six combinations are accepted by the firewall, whereas only nineteen are forwarded between the second FIN-ACK and the last ACK.

4.5 Replay of Production Traffic

Capture/replay based testing sounds very appealing; just press "record," execute the software-under-test and press "replay" later to run the test. In the ideal case, the result is realistic tests with negligible development cost. In reality the situation is more complex and there are many kinds of capture/replay. In the networking domain, the following questions arise:

- What network interface(s) do you monitor?
- Which packets do you extract for retransmission?
- During replay, on which interface do you transmit each packet?
- When do you transmit each packet?
- After replay, how do you determine success and failure automatically?

With the `TCPReplayIter` class, packet capture is performed at a single network interface using an off-the-shelf packet sniffer. Each `TCPReplayIter` instance handles one captured TCP stream. The tester provides the connection identifier of the client to the constructor and `TCPReplayIter` extracts TCP packets with either that identifier or the swapped version corresponding to the server.

For replay, the tester provides the I/O identifiers of the client and server `PacketIO` objects. Then each extracted packet with the client connection identifier is sent to the client `PacketIO` object; similarly, each extracted packet with the server connection identifier is sent to the server `PacketIO` object. Correctness checking is straightforward: every transmitted packet should be forwarded.

There remains the subtle issue of *when* to send the packets. `TCPReplayIter` supports three modes:

- *Timestamp-driven.* Each packet is transmitted using the timestamps present in the capture file. This approach provides the most accurate replay but can be problematic, as noted in Section 3 under *Support for handshaking.*
- *Stop and wait.* In this mode each packet is transmitted as soon as the previous packet has been forwarded, eliminating handshaking problems, but ignoring the captured timing information.
- *Hybrid.* This mode combines the previous two, transmitting each packet at the later of (1) the captured timestamp and (2) the arrival of the previous packet.

5. RELATED WORK

The utility of production grammars in software testing has already been demonstrated; there has already been a great deal of work involving production grammars in compiler testing [5]. Within network testing, far less work has been done, but the Protos project did use production grammars to test SNMP [1].

Our pairwise generation algorithm is based on the In-Parameter-Order (IPO) algorithm proposed by Tai and Lei [12]. We have improved the IPO algorithm and our improved algorithm always generates the same or a smaller number of tuples than the original IPO algorithm. In some test scenarios, our improved IPO algorithm also generates better results than the AETG commercial tool [4].

Firewall testing has been conducted in ways much different from our own, including the three techniques that will now be described. Vulnerability testing is one manner of determining the effectiveness of a firewall [2]. The authors use the SATAN security analysis tool to probe two firewalls for vulnerabilities. The firewall analysis tool Fang accepts and answers queries regarding the network security policy using static rule checking [9]. For example, a query might be: "Which services are allowed between the internal network and the Internet." Another approach to firewall testing involves using a CASE tool to model the firewall and the surrounding network [7]. Test cases can then be automatically generated from this model. Their tool does not create live packets; rather, human-readable "message sequence charts" are generated.

The most comparable work to Blowtorch is the MACE framework [11]. Both Blowtorch and MACE are frameworks for low-level packet generation. However, there is a critical difference between the two approaches: Blowtorch is able to generate traffic on both the client and server-side, whereas Blowtorch can only create client-side traffic. For instance, consider the TCPIter class described earlier. For the TCP session, Blowtorch is able to send the initial SYN packet from the client, the SYN-ACK packet from the server, and every other packet on both sides of the TCP connection. MACE can only send the SYN packet from the client side, along with the other client-side packets. In terms of the tests that were presented earlier, MACE cannot perform the Bad TCP Flags test, the Data Historian connections test, nor duplicate the capture/replay functionality.

6. CONCLUSIONS

The effectiveness of a firewall depends directly on its configuration, which is done through rule sets. Despite the simplicity of the rule set language, many firewalls have significant configuration errors. Therefore, tools that test a firewall and its rule set are required. Most of the existing firewall testing is done using vulnerability assessment tools which test generic network vulnerability but not a particular firewall configuration.

We have presented a framework aimed at developing policy-specific firewall tests. We have shown how Blowtorch can test a 3-way firewall in a process control environment. In testing this firewall, we have demonstrated the power and flexibility of the framework with four novel tests. These tests, while particular to firewall testing, employ classic methods in testing: the bad TCP flags test uses production grammars, and the Data Historian connections test utilizes covering arrays. The Blowtorch framework also allows for accurate packet timing, as shown in the SYN flood test, for traffic capture and replay, as evidenced in the capture/replay test, and for control of handshaking, as demonstrated by the TCPIter class. Within this framework, we expect to create many more tests in the immediate future, and to expand the framework itself.

7. ACKNOWLEDGMENTS

We would like to thank B. Chauvin of the British Columbia Institute of Technology for his invaluable assistance in configuring the Cisco Pix firewall for the bad TCP flags tests.

8. REFERENCES

[1] Protos - security testing of protocol implementations, 2000. http://www.ee.oulu.fi/research/ouspg/protos/.

[2] K. Al-Tawil and I. A. Al-Kaltham. Evaluation and testing of internet firewalls. *Int. J. Netw. Manag.*, 9(3):135–149, 1999.

[3] E. Byres and K. Savage. NISCC good practice guide on firewall deployment for SCADA and process control networks. http://www.niscc.gov.uk/niscc/docs/re20050223-00157.pdf, 2005.

[4] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Softw.*, 13(5):83–88, 1996.

[5] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 170–178, Piscataway, NJ, USA, 1981. IEEE Press.

[6] D. Hoffman and E. Byres. Worlds in collision: Ethernet on the plant floor. In *ISA Emerging Technologies Conference*. Instrumentation Systems and Automation Society, Oct. 2002.

[7] J. Jurjens and G. Wimmel. Specification based testing of firewalls. In *4th International Conference Perspectives of System Informatics*, 2001.

[8] J. Labrosse. *MicroC OS II: the Real Time Kernel.* CMP Books, second edition, 2002.

[9] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*, page 177, Washington, DC, USA, 2000. IEEE Computer Society.

[10] B. McCarty. *Red Hat Linux Firewalls.* Wiley Publishing, first edition, 2003.

[11] J. Sommers, V. Yegneswaran, and P. Barford. A framework for malicious workload generation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 82–87, New York, NY, USA, 2004. ACM Press.

[12] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[13] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, fourth edition, 2003.

[14] A. Wool. A quantitative study of firewall configuration errors. *Computer Magazine*, pages 62–67, 2004.