Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits

Helen J. Wang

Chuanxiong Guo

Daniel R. Simon

Alf Zugenmaier

{helenw, t-chuguo, dansimon, alfz} @ microsoft.com

Feburary, 2004

Technical Report
MSR-TR-2003-81

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits

Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier

***Abstract*— Software patching has not been an effective first-line defense preventing large-scale worm attacks, even when patches had long been available for their corresponding vulnerabilities. Generally, people have been reluctant to patch their systems immediately, because patches are perceived to be unreliable and disruptive to apply. To address this problem, we propose a first-line worm defense in the network stack, using *shields* – vulnerability-specific, exploit-generic network filters installed in end systems once a vulnerability is discovered and before the patch is applied. These filters examine the incoming or outgoing traffic of vulnerable applications, and drop traffic that exploits vulnerabilities. Shields are less disruptive to install and uninstall, easier to test for bad side effects, and hence more reliable than traditional software patches.**

In this paper, we show that this concept is feasible by describing a prototype Shield framework implementation that filters traffic at the transport layer. We designed a safe and restrictive language to describe vulnerabilities as partial state machines of the vulnerable application. The expressiveness of the language has been verified by encoding the signatures of a number of known vulnerabilites. Our evaluation provides evidence of Shield's low false positive rate and impact on application throughput. An examination of a sample set of known vulnerabilities suggests that Shield could be used to prevent exploitation of a substantial fraction of the most dangerous ones.

I. Introduction

One of the most urgent security problems facing administrators of networked computer systems today is the threat of remote attacks on their systems over the Internet, based on vulnerabilities in their currently running software. Particularly damaging have been self-propagating attacks, or "worms", which exploit one or more vulnerabilities to take control of a host, then use that host to find and attack other hosts with the same vulnerability.

The obvious defense against such attacks is to prevent the attack by repairing the vulnerability before it can be exploited. Typically, software vendors develop and distribute reparative "patches" to their software as soon as possible after learning of a vulnerability. Customers can then install the patch and prevent attacks that exploit the vulnerability.

Experience has shown, however, that administrators often do not install patches until long after they are made available—if at all [24]. As a result, attacks—including worms, such as the widely publicized CodeRed [5], Slammer [30] and MSBlast [18] worms—that exploit known vulnerabilities, for which patches had been available for quite some time, have nevertheless been quite "successful", causing widespread damage by attacking the large cohort of still-vulnerable hosts.

There are several reasons why administrators may fail to install software patches:

- *Disruption:* Installing a patch typically involves rebooting, at the very least, a particular host service, and possibly an entire host system. An administrator for whom system and service uptime are crucial may therefore be unable to tolerate the required service or system disruption.
- *Unreliability:* Software patches are typically released as quickly as possible after a vulnerability is discovered, and there is therefore insufficient time to do more than cursory testing of the patch. For popular software programs, patch testing is inherently difficult and involves an exponential number of test cases due to their numerous versions, and their dependencies on various versions of libraries—which depend in turn on other libraries, and so on. Hence patches can have serious undetected side effects in particular configurations, causing severe disruption and even damage to the host systems to which they are applied. Rather than risk such damage, administrators may prefer to do their own thorough (and time-consuming) testing, or simply wait—accepting the risks of vulnerability in the meantime—until the patch has been vindicated by widespread uneventful installation [2].
- *Irreversibility:* Most patches are not designed to be easily reversible. Once it is applied, there is often no easy way of uninstalling the patch, short of restoring a backup version of the entire patched application (or even the entire system). This factor exacerbates the risk associated with applying a patch.
- *Accident:* An administrator may simply miss a patch announcement for some reason, and therefore be unaware of it, or have received the announcement but neglected to act on it.

Because of these drawbacks to installing patches, methods are being explored for mitigating vulnerabilities without installing patches, or at least until a patch is determined to be safe to install. The goal is to address the window of vulnerability between vulnerability disclosure and software patching. (This window is illustrated in Figure 1 as the gap between times $T2$ and $T3$.) A firewall, for example, can be configured to prevent traffic originating outside a local network from reaching a vulnerable application, by blocking the appropriate port. By doing so, it can protect an application from attack from "outside". Of course, blocking all traffic to a port is a crude measure, preventing the application from functioning at all across the firewall. Ideally, only traffic that exploits the vulnerability would be blocked, while all other

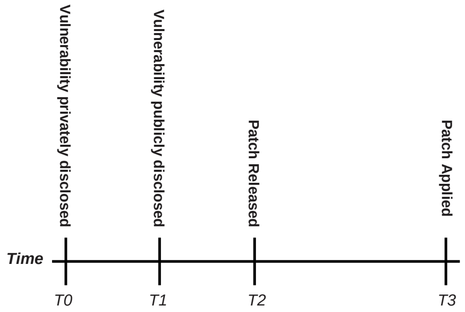
Vulnerability privately disclosed
Vulnerability publicly disclosed
Patch Released
Patch Applied
Time
T0
T1
T2
T3

Fig. 1. Time Windows: the scale of the X-axis does not reflect the actual time window duration. In fact, the time between any adjacent T's could be 0, though it is typically much greater than 0; the time between T2 and T3 are often in weeks or months.

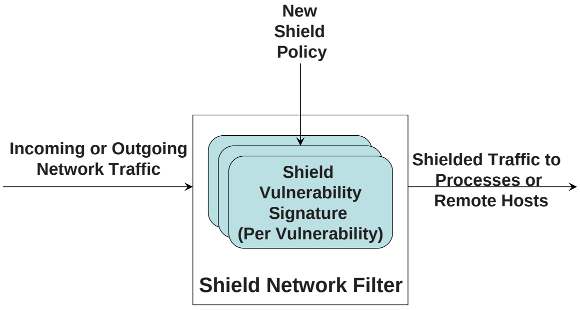
New Shield Policy
Incoming or Outgoing Network Traffic
Shield Vulnerability Signature (Per Vulnerability)
Shielded Traffic to Processes or Remote Hosts
Shield Network Filter

Fig. 2. Shield Usage

traffic would be allowed to pass through to the application.

In this paper, we explore the possibility of applying an intermediate "patch" in the network to perform this filtering function, to delay (or perhaps in some cases even eliminate) the need for installing the software patch that removes the vulnerability. *Shield* is a system of vulnerability-specific, *exploit-generic* network filters (*shields*) installed either at the end host, firewall or edge router, that examines the incoming or outgoing traffic of vulnerable applications and drops or modifies the traffic according to the vulnerability signature. Shield operates at the application level protocol layer, *above* the transport layer. For example, a shield (conceptually, there is one shield per vulnerability) designed to protect against a buffer overrun vulnerability would detect and drop traffic that resulted in an excessively long value being placed in the vulnerable buffer. Shield differs from previous anti-worm strategies (Section X) in attempting to remove a specific vulnerability directly, rather than mitigate or counter the effects of its exploitation.

Unlike software patches, shields can be deployed in the network as well as in the network stack of the end host. They are thus more separated from the vulnerable application—and its wide variety of potential environment configurations—and therefore less likely to have unforeseen side effects. In particular, their compatibility with normal operation is in principle relatively easy to test: since they operate only on network traffic, and are intended only to drop attack traffic, they can be tested simply by exposing them to a suitably rich collection of network traffic—such as a long trace of past network activity, or a synthetic test suite of representative traffic—to verify that they would allow it all through unaffected.

In this paper, we focus our attention on the design and implementation of an end host-based Shield system. The most efficient kind of end-host Shield would be positioned at the highest protocol layer, namely the application layer — assuming that hooks into that layer is available for traffic interception and manipulation. This way, any redundant message parsing would be avoided. For example, URLScan [6] is essentially a Shield specific to Microsoft IIS web server, which uses IIS ISAPI extension package that offers hooks for HTTP request interception and manipulations. However, most applications do not offer such extensilibiliy into their protocols. To this end, we have designed a Shield framework that lies between the application layer and the transport layer and offers shielding for *any* application level protocols. Being above the transport layer, Shield does not need to deal with IPSec-encrypted traffic. Encrypted traffic above the transport layer, such as SSL- or application-specific encrypted traffic are difficult for such a framework to handle[1]. Nevertheless, it is sensible to build respective Shield frameworks for commonly used protocols such as SSL and RPC [25]. And the techniques described in this paper can be readily applied to them.

In our Shield framework, we model vulnerability signatures as a combination of partial protocol state machines and vulnerability-parsing instructions for specific payloads (Section III). For generality, we abstract out the generic elements of application-level protocols in our Shield architecture (Section IV). For flexibility and simplicity, we express vulnerability signatures and their countermeasures in a safe, restrictive, and yet expressive policy language, interpreted by the Shield framework at runtime (Section VI). We also minimize Shield's maintenance of protocol state for scalability, and apply defensive design to ensure robustness (Section V-A). We have implemented a preliminary Shield prototype and experimented with a number of known vulnerabilities, including the ones behind the (in)famous MSBlast, Slammer, and CodeRed worms (Section VIII). Our evaluation provides evidence of zero false positives and manageable impact on application throughput (Section IX). An examination of a sample set of known vulnerabilities suggests that Shield could be used to prevent exploitation of a substantial fraction of the most dangerous ones (Section IX-A).

II. Overview of Shield Usage

Figure 2 gives an overview of basic Shield operations. An end-host Shield Network Filter intercepts the network traffic, and examines and manipulates the traffic according to the installed shield policies. Each shield policy specifies a vulnerability signature, i.e., how to recognize network traffic that exploits a given vulnerability, as well as the actions to take

[1] Nonetheless, since Shield runs with the root priviledge on end hosts, it is possible for Shield to obtain encryption keys and perform decryption. In fact, the Shield framework would not incur heavy overhead for stream-based ciphers.

when such traffic is encountered. Conceptually, there is one Shield policy per vulnerability. When a new vulnerability is discovered, a shield designer—typically the vulnerable application's vendor—creates a Shield policy for the vulnerability and distributes it to users running the application[2]. *Incoming* shields protect a host from potentially malicious incoming traffic, in a similar fashion to a firewall, but with much more application-specific knowledge. There can also be *outgoing* shields filtering out traffic that triggers vulnerability-exploiting responses back to the host itself, or protects other hosts on the same local network from the host's own vulnerability-exploiting outgoing traffic. The latter use assumes that the shield is installed at a higher privilege level than the malicious or compromised sender of the vulnerability-exploiting traffic. Otherwise, the sender could simply disable the shield before attacking the other hosts.

When receiving a shield policy, the end host enacts the policy by installing the policy into the Shield system. Note that this action does not require re-starting the vulnerable service or rebooting the machine. Once a software patch is applied to the vulnerable application, eliminating the vulnerability, the corresponding policy can be removed from Shield.

III. Vulnerability Modeling

An essential part of the Shield design is the method of modeling and expressing vulnerability signatures. A *Shield vulnerability signature* specifies all possible sequences of network events and specific payload characteristics that lead to *any* exploit of the vulnerability. (Note that not all vulnerabilities are suitable for shielding. This issue is discussed further in Section IX-A.) For example, the signature for the vulnerability behind the Slammer worm [30] is the arrival of a UDP packet at port 1434 with a size that exceeds the legal limit of the vulnerable buffer used in the Microsoft SQL server 2000 implementation. More sophisticated vulnerabilities require tracing a sequence of messages leading up to the actual message that can potentially exploit the vulnerability.

To express vulnerability signatures precisely, we have developed a taxonomy for modeling vulnerabilities, as illustrated in Figure 3.

Each application can be considered as a finite state machine, which we call the *application state machine*. Overlaying on top of the application state machine is the *Protocol State Machine* where the transitions are network event arrivals. The protocol state machine is much smaller and simpler than the application state machine. And the application state machine can be viewed as a refinement of the protocol state machine. That is, when zooming into a particular state of the protocol state machine, there is a fine-grained application state machine (e.g., enlarged state $S2$ in the figure). Shield is primarily concerned with the protocol state machine. We define the *pre-vulnerability state* as the state in the protocol state machine at which receiving an exploitation network event could cause damage ($v4$ in the figure). We call the partial protocol state machine that leads to the pre-vulnerability state the *vulnerability state machine*, and the network event that can potentially contain the vulnerability the *vulnerable event*.

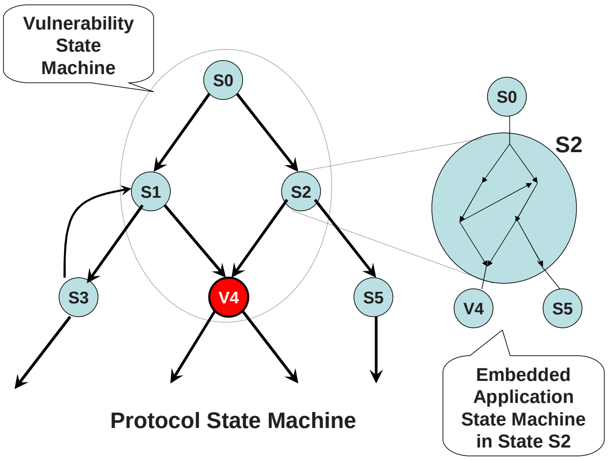
Vulnerability State Machine
S0
S1
S2
S3
V4
S5
Protocol State Machine
S0
S2
V4
S5
Embedded Application State Machine in State S2

Fig. 3. Vulnerability Modeling

A Shield vulnerability signature essentially specifies the vulnerability state machine and describes how to recognize the vulnerability in the vulnerable event. A Shield policy for a vulnerability includes both the vulnerability signature and the actions to take on recognizing an exploit of the vulnerability. In Section VI, we detail our design of a policy language for Shield policy specification.

At a high level, a Shield for a vulnerability intercepts its application's traffic and walks through the vulnerability state machine; when reaching the pre-vulnerability state, the Shield examines the vulnerable event for possible exploits and takes the specified actions to protect against the exploits if they are present.

IV. Shield Architecture

A. Goals and Overview

The objective of Shield is to emulate the part of the application level protocol state machine that is relevant to its vulnerabilities and counter any exploits at runtime.

We identify three main goals for the Shield design:

1) *Minimize and limit the amount of state maintained by Shield*: Shield must be designed to resist any resource consumption ("Denial-Of-Service", or "DoS") attacks. Therefore, it must carefully manage its state maintenance. For an end-host-based Shield, the bar is not high: Shield only needs to be as DoS-resilient as the service it is shielding.
2) *Enough flexibility to support any application level protocol*: Flexibility must be designed into Shield so that vulnerabilities related to any application level protocol can be protected by Shield. Moreover, the Shield system design itself should be independent of specific application level protocols, because the Shield system design

[2] Secure and expeditious distribution of Shield policies is an open research question by itself, and is out of the scope of this paper.

and implementation would simply not scale if it were necessary to add individual application level protocols to the core system one at a time.

3) *Defensive design*: We must design Shield in such a way that Shield does not become an easier alternative attack target. A robust Shield design must ensure that Shield's state machine emulation is consistent with the actual state machine running in the vulnerable application under all conditions. In other words, it is crucial for us to defend against carefully crafted malicious messages that may lead to Shield's misinterpretation of the application's semantics.

Shield achieves goal 2 by applying the well-known principle of "separating policy from mechanism". Shield's mechanism is generic, implementing operations common among all application level protocols. Shield policies specify the varying aspects of individual application level protocol design as well as the corresponding vulnerabilities. This separation ensures that Shield has the flexibility to support any application-level protocol.

We identify the following mechanisms as the necessary generic elements of an application level protocol implementation: (Less obvious generic elements will be explained throughout the next section.)

- Application level protocols between two parties (say, a client and server) are implemented using finite state automata according to some state machine design specification.
- To carry out state machine transitions, each party must perform event identification and session dispatching if the protocol allows multiple parallel sessions. (For transaction-based protocols such as HTTP where a transaction is a pair of request and response, a session has just one packet for each direction). As a result, application-level messages must indicate a message type and session ID (if applicable).
- Implementations of datagram-based protocols must handle out-of-order application datagrams for its sessions. (See Section V-B.)
- Implementations of application-level protocols whose messages cross packet boundaries must handle fragmentation. (See Section V-C.)

The policy specifies the following:

- *Application identification*: how to identify which packets are destined for which application. The port number used serves this purpose.
- *Event identification*: how to retrieve the message type from a received message.
- *Session identification* (if applicable): how to determine which session a message belongs to.
- *State machine specification*: the states, events and transitions defining the protocol automaton. In our setting, the specification is for the vulnerability state machine, a subgraph of a complete protocol state machine (see Section III).

We first present the Shield mechanisms, including the policy-enabling mechansims, in Section IV-B. Then, we present our policy language in Section VI.

B. Components and Data Structures

In this section, we describe the essential components and data structures of an end-host Shield system. Figure 4 depicts the Shield architecture.

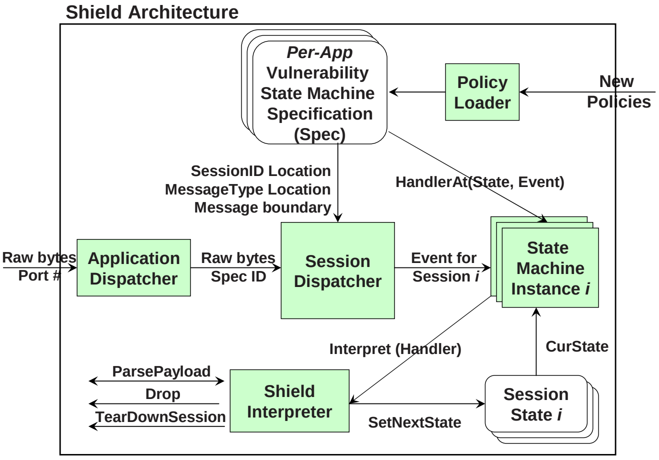

Fig. 4. Shield Architecture

Data Structures

There are two main data structures: the *application vulnerability state machine specifications* ("Spec") and the runtime *session states*.

The *Policy Loader* transforms Shield policies into Specs. Multiple vulnerability state machines for the same application are compiled into one "application vulnerability state machine specification". Therefore, there is effectively one state machine specification per application. The purpose of a Spec is to instruct Shield on how to emulate the application vulnerability state machines at runtime. As mentioned in Section IV-A, the Spec contains the state machine specification, port number(s) for application identification, and event and session identification information.

For event and session identification, a Spec indicates the location (i.e., offset and size) vector for the event type and session ID information in the packet, as well as the event type values that are of concern to Shield. For application protocols that are not session-oriented, the session ID is left unspecified, and each session consists of a single message. Sometimes, an application-level protocol may involve negotiating for a dynamically selected port number as a session ID for further communications (e.g., FTP [22] and RTP [26]). In this case, the new port number will be registered with Shield for application identification, and the session ID is specified as "PORT", indicating that all communication on this port is considered as a single session. Upon termination of the session, the dynamic port is de-registered with Shield.

To generalize event recognition and session dispatching to text-based application level protocols such as HTTP [7] and SMTP[12], we allow the units of “offset” and “size” to be defined as *words* (made of characters), in addition to bytes. For example, in HTTP and SMTP, the message type is indicated at offset 0, with a size of 1 word. In HTTP, this field contains the request-line method, such as ”GET” or ”POST”; when it is an HTTP version, it represents a status message type [7]. And in SMTP, this field contains the SMTP command, such as “MAIL”, “RCPT”, or “DATA”. Of course, we can generalize the unit even further using unit delimiters. For example, the unit deliminator for words is space while the unit delimiter for bytes is nothing. Nonetheless, we have not found this to be necessary for a handful of protocols we have examined.

Some application-level protocols (such as HTTP) allow multiple application-level messages to be received in a single buffer. Therefore, in addition to the session ID and message type, the Spec also specifies the *application level message boundary marker*, if any. For example, for HTTP, the message boundary marker is CRLF CRLF; and for SMTP, it is CRLF.

One key challenge is that application level messages may not be received in their entirety (due to congestion control or application-specific socket usage) or in order (due to the use of a protocol such as UDP). Even the essential event-identifying parts of a message, such as event type and session ID, may not arrive together. We address this problem using DoS-resilient copying or buffering, which is detailed in Section V.

Note that the *session* is an important abstraction for packet dispatching and as a unit of shielding, apart from socket descriptors or host pairs. This is because one socket descriptor may be used for multiple sequential sessions; and multiple sockets may be used to carry out communications over one session (e.g., FTP [22]). Similarly, one pair of hosts may be carrying out multiple sessions. In these cases, the use of sessions eliminates any ambiguities on which packets belong to which session.

The other data structure in Shield is *session state*. At runtime, Shield maintains session state for each potentially vulnerable communication session. The session state includes the current state of the session and other context information needed for shielding.

Shield Modules

Now, we describe each Shield module in turn:

- *Policy Loader:* Whenever a new Shield policy arrives or an old policy is modified, the Policy Loader integrates the new policy with an existing Spec if one exists, or creates a new one otherwise. The Shield policy is expressed in the Shield policy language. Policy loading involves syntax parsing and the syntax tree is also stored in the Spec for the purpose of run-time interpretation of shielding actions (For details on the policy language design and interpretation, please see Section VI).
- *Application Dispatcher:* When raw bytes arrive at Shield from a port, the *Application Dispatcher* is invoked to determine which Spec to reference for the arrived data, based on the port number. While an application-level protocol may use many port numbers, each port number corresponds to a single application. The Application Dispatcher forwards the raw bytes and the identified Spec to the Session Dispatcher for event and session identification.
- *Session Dispatcher:* On obtaining the locations of the session ID, message type, and message boundary marker from the corresponding Spec, the Session Dispatcher extracts multiple messages (if applicable), recognizes the event type and session ID, and then dispatches the event to the corresponding state machine instance.
- *Runtime State Machine Instance (SMI):* There is one state machine instance per session. Given a newly-arrived event and the current state maintained by the corresponding session state, the SMI consults the Spec regarding which event handler to invoke. (Event handlers are included in Shield policies.) Then the SMI calls the Shield Interpreter to interpet the event handler.
- *Shield Interpreter:* The Shield Interpreter interprets the event handler, which specifies how to parse the application-level protocol payload and examine it for vulnerabilities. It also carries out actions like packet-dropping, session tear-down, registering a newly-negotiated dynamic port with Shield, or setting the next state for the current SMI.

V. Implementation Issues

A. Scattered Arrivals of an Application Message

Although Shield intercepts traffic above the transport layer and does not need to cope with network-layer fragments, each data arrival perceived by Shield does *not* necessarily represent a complete application level message that is independently interpretable by the application. The scattered arrivals of a single application level message could be due to TCP congestion control or some specific message-handling implementations of an application. For instance, a UDP Server may make multiple calls to $recvfrom()$ to receive a single application level message. In this case, Shield would recognize multiple data arrivals for such messages. This complicates session dispatching when session ID or message type are not received “in one shot”. It also complicates payload parsing in the event handlers when not enough data has arrived for an event-handler to finish parsing and checking. Here, we must *copy* (i.e., buffer and pass on) *part of* the incompletely-arrived data in the Shield system, and wait for the rest of the data to arrive, before we can interpret it.

In addition, we need to *index* copy buffers so that later arrivals of the same message for a given session can be stitched together properly. Although socket descriptors are not appropriate to identify sessions (Section IV-B), they are safe for indexing copy buffers: while multiple sockets could be used for one session (e.g., FTP [22]), a single application message is typically [3] not scattered over multiple sockets—otherwise the application would not be able to interpret the parts of the message due to the lack of information such as session ID or message type; similarly, although one socket descriptor could be used for multiple parallel sessions, an application message has to be received on a socket continuously to its completion without interruptions from any other application messages. Therefore, we can apply *per-socket* copying for incomplete message arrivals.

We differentiate between *pre-session copying* and *in-session copying*. *Pre-session copying* happens when the session ID information has not completely arrived, while *in-session copying* refers to the copying of the data whose session is known. A copy buffer is associated with a socket initially before a session ID fully arrives. Once the session ID is received, the copy buffer is associated with both a socket and its respective session. Once a complete application message has been received, the copy buffer is de-allocated.

We do not need to save the entire partially arrived message, but only the partially arrived *field*. For example, when a Session ID field has not arrived completely—say only 2 out of 4 bytes—Shield only needs to remember that it is parsing the Session ID field, and saves the received two bytes only. Therefore, copying in Shield is small.

Here, we introduce another runtime data structure that needs to be maintained by Shield: *parsing state*. This state is per application-level message, and it records which field of an application message is being parsed, and how many bytes have already been received for that field. The field has to be a "terminal" field rather than a structure of fields: For a field nested in other structures or an array, the field is represented as something like "someStructure.fields[i]". This restriction is to minimize the amount of copying—copying for a terminal field is typically small.

For a vulnerable application, we must maintain the state of the current field being parsed for *each* of the application messages, *even* when Shield had already determined that the session to which the message belongs would not lead to any exploit. (Nonetheless, we do not maintain session state and the copy buffer for such sessions.) This is to avoid ambiguity: if we do not keep the parsing state for the message, other parts of the message would be treated as new application messages. Attackers could easily craft parts of a single application level message, send them separately, and cause inconsistencies between the emulated state machine in Shield and the actual state machine in the application.

For Shield to be able to parse application messages, parsing instructions (or payload formats) for all message types of an application must be specified to Shield through policy descriptions. Therefore, payload formats are also part of the Spec. Fortunately, Shield does not need to parse all messages in detail, but only the parts necessary for detecting the presence of an exploit. Therefore, we can aggressively bundle many fields of an application message into one field with a total byte count or word count, which will be the number of bytes or words to skip during parsing. When Shield concludes the innocence of a session, the goal of parsing its subsequent application-level messages is only to find the end of those messages. Hence, parsing for those messages can be even further streamlined.

While specifying all application messages seems daunting, if an application level protocol were specified in a standard and formalized format (such as our policy language-like format—see Section VI), we could automatically extract payload format specifications and vulnerability state machines from that format to our policy language. On the other hand, if a Shield designer knows for a fact that scattered arrivals of a message does not happen (e.g., single rather than multiple $recvfrom()$ calls when receiving a single application message in a UDPServer implementation), then only events involved in the vulnerability state machine need to be specified.

B. Out-of-Order Arrival of Application Datagrams

When an application-level protocol runs on top of UDP, its datagrams can arrive out of order. Applications that care about the ordering of these datagrams will have a sequence number field in their application level protocol headers. For Shield to properly carry out its exploit detection functions, Shield copies the out-of-order datagrams, and passes them on to the applications. This way, Shield can examine the packets in their intended sequence. Shield sets the upper limit of the number of copied datagrams to be the maximum number of out-of-order datagrams that application level protocol can handle. Hence, this maximum also needs to be expressed in the policy descriptions, so does the sequence number location.

C. Application Level Fragmentation

Shield runs on top of the transport layer. Hence, Shield does not need to deal with network-layer fragmentation and re-assembly.

Nonetheless, some application-level protocols use application data units and perform application level fragmentation and re-assembly. For protocols on top of TCP, bytes are received in order. And for protocols on top of UDP, Shield copies their out-of-order datagrams to retain the correct packet sequence (see the above section). Therefore receiving and processing application level fragments is no different from processing partially arrived data, as explained in Section V-A. However, the Spec needs to contain the location of the application-level fragment ID in the message, so that a fragment is not treated as an entire message event.

VI. SHIELD POLICY LANGUAGE

In this section, we present the Shield policy language which is used to describe the vulnerabilities and their countermeasures for an application.

Figure 5, 6, 7 shows some examples of our policy language usage. They are policy scripts for the vulnerabilities behind MSBlast [18], Slammer [30], and CodeRed [5], respectively.

[3] pTCP [10] proposes the use of TCP-v as an abstract of a connection which can use multiple sockets instead of one as in TCP. If pTCP were deployed, Shield would use TCP-v to index the copy buffer instead.

```
# SHIELD (Name, Transport_Protocol, (port-list))
SHIELD (Vulnerability_Behind_MSBlast, TCP, (135, 139, 445))

# where to retrieve SESSION_ID and MSG_TYPE from
SESSION_ID_LOCATION = (12, 4);
MSG_TYPE_LOCATION = (2, 1);

INITIAL_STATE S_WaitForRPCBind;
FINAL_STATE   S_Final;
STATE       S_WaitForRPCBindAck;
STATE       S_WaitForRPCAlterContextResponse;
STATE       S_WaitForRPCRequest;
STATE       S_WaitForSessionTearDown;

# EVENT eventName = (<eventTypeValue>, <direction>)
EVENT E_RPCBind         = (0x0B, INCOMING);
EVENT E_RPCBindAck = (0x0C, OUTGOING);
EVENT E_RPCBindNak = (0x0D, OUTGOING);
EVENT E_RPCAlterContext = (0x0E, INCOMING);
EVENT E_RPCAlterContextResponse = (0x0F, OUTGOING);
EVENT E_RPCRequest = (0x0,  INCOMING);
EVENT E_RPCShutdown = (0x11, OUTGOING);
EVENT E_RPCCancel = (0x12, INCOMING);
EVENT E_RPCOrphaned = (0x13, INCOMING);

STATE_MACHINE = {
# (State, Event, Handler),
(S_WaitForRPCBind, E_RPCBind, H_RPCBind),
(S_WaitForRPCBindAck, E_RPCBindAck, H_RPCBindAck),
(S_WaitForRPCRequest, E_RPCRequest, H_RPCRequest),
...
};

# payload parsing instruction for P_Context
PAYLOAD_STRUCT {
   SKIP BYTES(6)                        dummy1,
   BYTES(1)                 numTransferContexts,
   SKIP BYTES(1)                        dummy2,
   BYTES(16)                 UUID_RemoteActivation,
   SKIP BYTES(4)                        version,
   SKIP BYTES(numTransferContexts * 20) allTransferContexts,
} P_Context;

# payload parsing instruction for P_RPCBind
PAYLOAD_STRUCT {
   SKIP BYTES(24)              dummy1,
   BYTES(1)                    numContexts,
   SKIP BYTES(3)        dummy2,
   P_Context[numContexts]      contexts,
   ...
} P_RPCBind;

HANDLER H_S_RPCBind (P_RPCBind)
{
   # if invoking the RemoteActivation RPC call
   IF (>>P_RPCBind.contexts[0] == 0xB84A9F4D1C7DCF11861E0020AF6E7C57)
      RETURN (S_WaitForRPCBindAck);
   FI
   RETURN (S_Final);
};

HANDLER H_RPCBindAck (P_RPCBindAck)
{
  RETURN (S_WaitForRPCRequest);
};

HANDLER H_RPCRequest (P_RPCRequest)
{
   IF (>>P_RPCRequest.bufferSize > 1023)
      TEARDOWN_SESSION;
      PRINT ("MSBlast!");
      # since other RPC requests can come as well
      RETURN (S_Final);
   FI
   RETURN (S_WaitForSessionTearDown);
};

# ... other PAYLOAD_STRUCTs and Handlers not included here ...
```

Fig. 5. Excerpt from the policy description of the vulnerability behind MSBlast [18]

```
# Vulnerability behind Slammer
SHIELD (VulnerabilityBehind_Slammer, UDP, (1434))

# 0 offset, size of 1 byte
MSG_TYPE_LOCATION = (0, 1);

INITIAL_STATE S_WaitForSSRPRequest;
FINAL_STATE S_Final;

# MsgType = 0x4
EVENT E_SSRP_Request = (0x4, INCOMING);

STATE_MACHINE = {
(S_WaitForSSRPRequest, E_SSRP_Request, H_SSRP_Request),
};


HANDLER H_SSRP_Request (DONT_CARE) {
  COUNTER legalLimit = 128;
   # MSG_LEN returns legalLimit + 1 when legalLimit is exceeded
  COUNTER c = MSG_LEN (legalLimit);
  IF (c > legalLimit)
    DROP;
    RETURN (S_FINAL);
  FI
  RETURN (S_Final);
};
```

Fig. 6. Policy description of the vulnerability behind Slammer [30]

```
# Shield for vulnerability behind CodeRed
SHIELD(CodeRed, TCP, (80))

INITIAL_STATE       S_WaitForGetRequest;
FINAL_STATE         S_Final;

#
MSG_TYPE_LOCATION= (0, 1) WORD;

MSG_BOUNDARY = "\r\n\r\n";

EVENT E_GET_REQUEST = ("GET", INCOMING);

STATE_MACHINE = {
(S_WaitForGetRequest, E_GET_Request, H_Get_Request),
};

PAYLOAD_STRUCT {
  WORDS(1)    method,
  WORDS(1)    URI,
  BYTES(REST) dummy2,
} P_Get_Request;

HANDLER H_Get_Request (P_Get_Request) {
   COUNTER legalLimit = 239;
   COUNTER c = 0;

   # \?(.*)$ is the regular expression to retrieve the
   # query string in the URI
   # MATCH_STR_LEN returns legalLimit + 1 when legalLimit is exceeded
   c = MATCH_STR_LEN (>>P_Get_Request.URI, "\?(.*)$", legalLimit);
   IF (c > legalLimit)
      # Exploit!
      TEARDOWN_SESSION;
      RETURN (S_FINAL);
   FI
   RETURN (S_FINAL);
};
```

Fig. 7. Policy description of the vulnerability behind CodeRed [5]

There are two parts of the policy specification in the Shield language. The first part includes states, events, state machine transitions, and generic application level protocol information such as ports used, the locations of the event type, session ID, sequence number or fragment ID in a packet, and the message boundary marker. This part of the policy specification is loaded into the Application Vulnerability State Machine Specification (Spec) directly by the Policy Loader (Figure 4), and is independent of runtime conditions.

The second part of the policy specification is for run-time interpretation during exploit checking. This includes the handler specification and payload parsing instructions (i.e., PAYLOAD_STRUCT definitions in the figures). The role of the handler is to examine the packet payload and pinpoint any exploit in the current packet payload, or to record the session context that is needed for a later determinination of exploit occurrence. To examine a packet, a handler needs to follow the policy's payload parsing instructions.

When a policy is loaded, the Policy Loader parses the syntax of the handlers and the payload parsing instructions, and stores the syntax tree in the Spec for run-time interpretation.

A. *Payload Specification*

The PAYLOAD_STRUCT definitions specify how to parse an application-level message. Shield needs not parse out all the fields of a payload as in the actual applications, but only needs to parse the fields relevant to the vulnerability. We allow the policy writers to simplify payload parsing by clustering insignificant fields together as a single dummy field of the required number of bytes (e.g., dummy1 field of P_RPC_Bind in Figure 5). Such fields are marked as skippable during parsing though the keyword $SKIP$ so that no copy buffer is maintained for such fields (Section V-A). From examining a number of application level protocols, we find that payload parsing specification only needs to support a limited set of types for fields including bytes of any size (i.e., BYTES(num) where "num" could be a variable size or an expression), words of any size (i.e., WORDS(num)) for text-based protocols, (multi-dimensional) array of PAYLOAD_STRUCT's, and boolean.

In a sense, our payload parsing specification for an application-level protocol message is like the Network Data Representation (NDR) [25] layout of the RPC stub data expressed in Interface Definition Language (IDL) [25] definition. IDL provides syntax for describing structured data types and values for RPC procedure call inputs and outputs; and NDR provides a mapping of IDL data types onto octet streams [25]. In fact, any application payload can be expressed in IDL-like syntax, then serialized into raw bytes with NDR-like encoding. Therefore, we believe a payload specification like ours is potentially generic enough to express any application payload.

B. Handler Specification

The Shield language for handler specification is very simple, and highly specialized for our purpose. Variables have only two scopes: They are either local to a handler or "global" within a session across its handlers. There are only four data types: BOOL for boolean, COUNTER for whole number, byte arrays such as "BYTES(numBytes)", and word arrays such as "WORD(numWords)". We also have built-in variables for handlers to use, such as SESSION_ID.

Built-in functions include DROP , TEARDOWN_SESSION, length-based functions such as MSG_LEN (Figure 6) and MATCH_STR_LEN (Figure 7), and regular expression functions that may be needed for text-based protocols. DROP drops a UDP packet while TEARDOWN_SESSION closes all sockets associated with a session. The regular expression functions are data stream-based rather than string buffer-based. That is, they must be able to cope with scattered message arrivals as well. Similarly, length functions are also stream-based with a required parameter of "stop count" (e.g., "legalLimit" in our example scripts in the figures) to facilitate the handling of buffer overrun-type of vulnerabilities. When the length reaches "stop count", counting stops and returns "stop count"$+1$. This way, Shield will not count and maintain state beyond what is necessary in the case of buffer-overrun exploits.

The syntax "$\gg$payload" instructs Shield to parse the bytes that represent "payload" of the packet, according to the parsing instruction defined for "payload" (i.e., there should be a definition: "PAYLOAD_STRUCT {...} payload;" earlier in the policy decription). The parsed fields of a PAYLOAD_STRUCT are treated as local variables for that handler. Within the handler, we allow assignments, if-statements, special-purpose for-loops, and return-statements that exit the handler and indicates the next state the session should be in. The specialized for-loops are more of a syntax sugar for parsing iterative payload structures such as array of items rather than traditional general-purpose for-loops. For example, given "FOR (item IN $\gg$ Payload.itemArray) $\{$ ... $\}$", the interpreter parses items of the "itemArray" field of the "Payload" iteratively, according to the "Payload" definition, and along the way, performs some operations on the bytes representing each "item". Note that the interpreter does not keep state as we parse the items out of an array.

During handler interpretation, the current payload being parsed may not be completely received. In this case, we save the execution state of the handler as part of the session state so that when new data arrives, the handler's execution can be resumed. This is very much like call continuation. In our case, the continuation state includes a queue of current handler statements being executed (because of potentially nested statements) and the parsing state (Section V-A) for the payload—that is, the current field of the payload being parsed, and the bytes read for that field.

The restrictive nature of our language makes it a safer language than general-purpose languages. While our language is restrictive, we find it sufficient for all of the vulnerabilities that we have worked with and the application level protocols that we have examined. However, it is still evolving as we gain experience from shielding more vulnerabilities.

Our language is simpler than the Bro language [20] that is used for scripting the security policies of the Bro Nework Intrusion Detection System (NIDS). This is because Bro performs network monitoring and intrusion detection for the network layer and above of the network stack and also monitors cross-application, cross-session interactions based on various attack patterns. In contrast, Shield is only concerned with application-specific traffic passing over top of the transport protocols or even higher level protocols such as HTTP and RPC. Furthermore, a key advantage of the vulnerability-driven approach of Shield over attack- or exploit-driven approachs such as NDIS is that Shield does not need to consider attack activities before the vulnerable application is involved (as in, for example, multi-stage attacks. Shield only needs to screen the traffic of particular vulnerable applications.

On the other hand, our language is more complex than the declarative Click router configuration language [13] because it has to cover more tasks than just configuration. Shield needs to parse the payload and perform actions based on the runtime events.

VII. Analysis

A. Scalability with Number of Vulnerabilties

In this section, we discuss how Shield scales with the number of vulnerabilities on a machine.

The number of Shields on an end host should *not* grow arbitrarily large, because Shields will presumably be removed when its corresponding vulnerability is patched.

Also, Shields are application-specific, adding negligible overhead to applications to which they do not apply. Hence, N

Shields for N different applications is equivalent to a single shield in terms of their effect on the performance of any single application.

An application may have multiple vulnerabilities over time. The state machines that model these vulnerabilities should preferably be merged into a single one. Otherwise, each state machine must be traversed for each packet, resulting in linear overhead. When these vulnerabilities appear on disjoint paths of the merged state machine, per-packet shield processing overhead for them is almost equivalent to the overhead for just one vulnerability. For vulnerabilities that share the same path in the state machine, however, shield overhead may be cumulative. On the other hand, our data on vulnerabilities presented in Section IX-A suggests that this cumulative effect is not significant: For worm-exploitable vulnerabilities, no more than three vulnerabilities ever appeared over a single application protocol throughout the whole year.

In any case, for vulnerable applications, the application throughput with shield is, at worst, halved, since the network traffic is processed at most twice —once in Shield and once in the application. Nonetheless, our experiment over our Shield prototype indicates that the Shield's impact on application throughput is quite small (Section IX-B).

B. False Positives

By design, shields are able to recognize and filter *only* traffic that exploits a specific vulnerability, and hence should have very low false positives. However, false positives may arise from incorrect policy specification due to misunderstanding of the protocol state machines or payload formats. Such incorrect policy specification can be debugged with stress test suites or simply by replaying a substantial application traffic traces. Trace replay at the application level is easy since it is not necessary to replay the precise transport protocol behavior.

Another source of false positives may come from the application behavior upon receiving an exploit event. The application state machine embedded at that state may only trigger the vulnerable code based on the local machine setting or some runtime conditions. While such information can also be incorporated in Shield, it is difficult to generalize such application-specific implementation details to simple and safe policy language constructs. For the vulnerabilities with which we have experimented, we have not observed such false positives (Section IX-C).

VIII. Implementation

We have prototyped an end host-based Shield system on Microsoft Windows XP. In particular, we have implemented Shield as a Microsoft WinSock2 Layered Service Provider (LSP) [11]. WinSock2 API is the latest socket programming interface for network applications on Windows. At runtime, these network applications link in the appropriate socket functions from WinSock2 dymamically linked library (DLL) upon socket function calls. The LSP mechanism in WinSock2 allows new service providers to be created for intercepting WinSock2 calls to the kernel socket system calls. An LSP is compiled into a dynamically linked library. Upon installation, any applications making WinSock2 calls links in both the WinSock2 DLL and the LSP DLL. We use this mechanism to implement Shield for intercepting application traffic above the transport layer (see Figure 8).

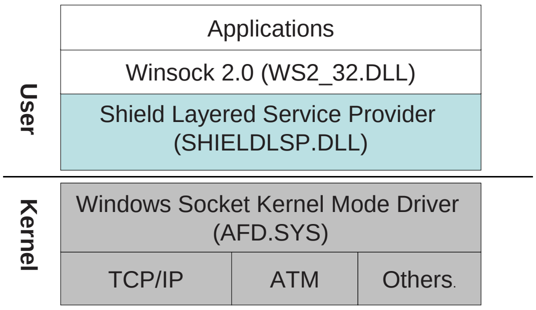
User
Applications
Winsock 2.0 (WS2_32.DLL)
Shield Layered Service Provider (SHIELDLSP.DLL)
Kernel
Windows Socket Kernel Mode Driver (AFD.SYS)
TCP/IP
ATM
Others.

Fig. 8. Shield Implementation using WinSock2 LSP

Our Shield LSP implements the architecture depicted in Figure 4 with 10,702 lines of C++ code [4]. We employ Flex [19] and Byacc [3] to parse the syntax of the Shield policy language. And the Policy Loader calls the Byacc API to obtain the syntax trees of the policy scripts.

We have used the vulnerabilities behind Slammer [30], MSBlast [18], CodeRed [5], and twelve other vulnerabilities from Microsoft security bulletin board to drive our design and implementation. They are all input validation type of vulnerabilities such as buffer overruns, integer overflow, or malformed URLs. Slammer exploits a proprietary application level protocol SSRP [30] on top of UDP. MSBlast exploits RPC [25] over either TCP or UDP. And CodeRed uses HTTP [7]. Other vulnerabilities exploit Telnet [21], SMB [28], HTTP or RPC. We have also examined some other application level protocols such as RTP [26] and SMTP [12] to design our policy language. Once we obtained the protocol specification [5] and the occurrence of the vulnerability in the corresponding payload, writing Shield policy was easy.

IX. Evaluations

A. Applicability of Shield

How applicable is Shield to real-world vulnerabilities? Shield was designed to catch exploits in a wide variety of application-level protocols, but there are several potential gaps in its coverage:

- Vulnerabilities that result from bugs that are deeply embedded in the application's logic are difficult for Shield to defend against without replicating that application logic in the network. For example, browser-based vulnerabilities that can be exploited using HTML scripting languages are difficult for Shield to prevent, since those languages are so flexible that incoming scripts would

[4] This line count does not include the generated Flex and Byacc files.

[5] It should be easy for application vendors to produce Shield policies since they have easy access to the protocol specifications.

Number of Vuln.	Nature	Worm-Exploitability	Shield-Applicability
6	Local	No	No
24	Client	No	Hard
12	Server buffer overruns	**Yes**	**Easy**
3	Cross-site scripting	No	Hard
3	Server Denial-of-service	No	Hard

TABLE I
APPLICABILITY OF SHIELD FOR VULNERABILITIES OF MSRC OVER THE YEAR 2003.

likely have to be parsed and run in simulation to discover if they are in fact exploits.

- Even simple vulnerabilities that are exploitable by malformed, *network protocol-independent* application objects (such as files) are difficult for Shield to catch. For example, a shield against otherwise simple buffer overruns in application file formats would have to spot an incoming file arriving over many different protocols. For file-based vulnerabilities, vulnerability-specific anti-virus softwares (rather than exploit-specific ones as being widely used today) would be more appropriate for them.
- Application specific encryption poses a problem for Shield as mentioned in Section I.

To assess the significance of these obstacles, we analyzed the entire list of security bulletins published by the Microsoft Security Response Center (MSRC) for the year 2003. Table I summarizes our findings. Of the 49 bulletins, six described vulnerabilities that were purely local, not involving a network in any way. Of the rest, 24 described "client" vulnerabilities (in the sense of requiring local user action on the vulnerable machine–such as navigating to a malicious Website, or opening an emailed application–to trigger), and the remaining 19 described server vulnerabilities (in the sense of being possible to trigger via the network, from outside the machine).

The client vulnerabilities generally appear difficult to design shields for. However, none of the client vulnerabilities are likely to result in self-propagating worms, because they cannot be exploited without some kind of user action upon the browser. For example, seven involved application file formats. Of the remainder, two were email client vulnerabilities, one was a media player vulnerability, and the rest were found in the browser, and hence invoked via HTML or client-side scripting.

Of the server vulnerabilities, twelve might conceivably be exploitable by worms, under "ideal" conditions—i.e., the server application being very widely deployed in an unprotected, unpatched and unfirewalled configuration. The remainder included three denial-of-service attacks, three "cross-site scripting" attacks, and a potential information disclosure. These are not vulnerable to exploitation by worms.

Of the potentially worm-exploitable vulnerabilities, five involved application protocols running over HTTP. The rest involved specific application protocols—typically directly over TCP or UDP—none of which appear inherently incompatible to the Shield approach. Moreover, all twelve were based on buffer overruns, and hence shield-applicable.

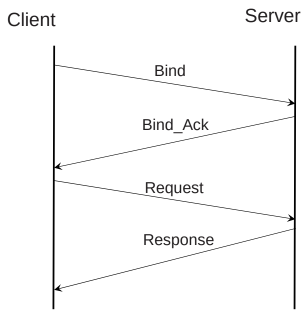

Fig. 9. The RPC message exchanges between our clients and server for throughput evaluation.

Thus, while many vulnerabilities may not appear to be suitable for the Shield treatment, in fact the most threatening–those prone to exploitation by worms–appear to be disproportionately Shield-compatible.

We also assessed the reliability of the patches associated with our sample set of security bulletins. Of the patches associated with the 49 bulletins, ten (including three repairing potentially worm-exploitable vulnerabilities) were updated at least once following their initial release. Eight of those (including two involving wormable vulnerabilities) were updated to mitigate reported negative side effects of the patch. (The others were augmented with extra patches for legacy versions of the product.) These side effects would likely have been avoided had Shield been used in place of the patch since a key advantage of shields over patches is its easy testability (Section I).

Finally, with the exception of HTTP-related vulnerabilities, no single application-level protocol exhibited more than RPC's three vulnerabilities during the entire year. Hence, apart from the HTTP port, no port is likely to be burdened with combining so many shields at a given time that the cumulative performance costs of large numbers of shields become an issue separate from the overhead of using Shield on a port in the first place.

B. Application Throughput

To evaluate the impact of Shield on the application throughput, we have devised the following experiment: We have clients establishing simultaneous client-server RPC sessions over TCP; for each session, the server sends 1 MB of data back to each of its clients in the RPC response (Figure 9). There are 1 server and n clients. The server is a Dell GX270 with Pentium 4 CPU 2.8GHZ and 512MB of RAM. The clients and the server are connected via a 100Mb Ethernet switch. All computers run Windows XP SP1.

To estimate the worse case impact, we used a policy that examines every byte of the traffic on the server. We measure the server's output throughput with Shield LSP enabled and disabled. Table II shows the result: the throughput with and without Shield do not have significant differences in this setting, hinting that our shield design and implementation do

Number of clients	Without Shield (Mbps/s)	With Shield (Mbps/s)
10	86.51	86.20
15	86.57	86.36
20	86.66	86.20
50	86.48	85.86
100	86.67	86.24
150	85.92	86.36
200	86.06	81.70
500	84.27	82.29
1000	66.29	57.56

TABLE II
APPLICATION THROUGHPUT WITH AND WITHOUT SHIELD

not have significant impact on the performance of the existing network protocol stack.

C. False Positives

As mentioned in Section VII-B, false positives come from either the misunderstanding of the protocol state machine or the differential treatment of an exploit in the application. In this section, we evaluate the false positve nature of our ShieldLSP implementation. We focus our attention to the Shield we designed for Slammer [30] which exploits the SSRP protocol of SQL Server 2000.

We obtained a stress test suite for SSRP from the vendor. SSRP is a very simple protocol with only 12 message types. The test suite contains a total of 36 test cases for exhaustive testing of SSRP requests of various forms. Running this test suite against our Shield, we did not observe any false postives. Although this does not prove Shield being false positive-free, it serves as an evidence of Shield's low false positiveness.

X. RELATED WORK

Shield is a network-based system for defending against vulnerability-exploiting attacks. Other network-based tools for defending against attacks include firewalls and network intrusion detection systems (NIDS). Firewalls have a similar function to Shield, but work in a much cruder way—rarely customized in response to a particular vulnerability, for instance. Moreover, they are usually not deployed on the end host, and are unaware of application-level protocols (and may not even have access to them—if, for example, traffic is encrypted).

NIDS systems, exemplified by Bro [20] and Snort [31], monitor the network traffic and detect attacks of known *exploits*. NIDS are usually more customized by application than firewalls, but to deal with known exploits rather than known vulnerabilities. Unlike Shield, NIDS is not on the traffic forwarding path. Moreover, they focus on detection rather than prevention of vulnerability exploits. For a more reliable attack detection, "traffic normalizers" [27], [9], [8] or "protocol scrubbers" [14] have been proposed to be used on the forwarding path by eliminating potential ambiguities before the traffic is seen by the monitor, removing evasion opportunities. The functions of the traffic normalizer is similar to that of Shield's. However, the traffic normalizer mainly deals with transport layer anomalies for the purpose of detection, while Shield is run above the transport layer and blocks the actual attack traffic. Shield has lower false postive rates and false negative rates than NIDS because of Shield's vulnerability-specific nature.

Malicious traffic filters specific to HTTP traffic and web servers [23] have also been proposed and deployed, such as URLScan[6] for Microsoft IIS web servers. These are most akin to Shield's approach. In comparison, Shield is a generic framework that supports any application level protocols.

The onsets of CodeRed [5], Slammer [30] and MSBlast [18] in the past few years have set a new stage for worm defense research. A number of papers [16], [32], [15], [33] have characterized and analyzed the fast- and wide-spreading nature and potential [34], [4] of modern-day worms. Moore et al [17] further showed that for existing containment systems such as firewalls and content filters to be successful against realistic worms, they must react automatically in a matter of minutes and must interdict nearly all Internet paths. This finding has spurred research on fast worm signature generation such as Earlybird [29]. The signatures can then be used by signature-based network intrusion prevention systems (NIPS) to filter traffic matching the signatures. Rate-limiting [35] is another containment method that throttles the sending rate at an infected end host. Another interesting worm detection mechanism is through "honeypots" which are unpatched vulnerable machines in the network with a number of IP addresses. Any unsolicited outgoing traffic from the honeypots represents the ocurrence of some attacks.

While most of the ongoing research copes with new *exploit* detections and counteractions, Shield prevents any exploits of *known* vulnerabilities which have been the sources of major damages so far. In the past, it has been assumed that removing vulnerabilities has been a matter of patch distribution and management [1], but recent research [24], [2] suggests that patching is not a complete solution. Shield thus provides an alternative or complement to the conventional approach of removing vulnerabilities by patching in the network stack.

XI. CONCLUSIONS AND FURTHER WORK

We have shown that network-based vulnerability-specific filters are feasible to implement, with low false positive rates, manageable scalability, and broad applicability across protocols. There are, however, still a number of natural directions for future research on Shield:

- Further experience writing shields for specific vulnerabilities will better indicate the range of Shield's applicability and the adequacy of the Shield policy language. It may also be possible to develop automated tools to ease Shield policy generation.

  For example, writing a shield policy currently requires a fairly deep understanding of the protocol over which the vulnerability is exploited. For protocols described in a standard, formalized format, however, it should be possible to build an automated tool that generates most of the protocol-parsing portion of a shield policy. The rest of the task of writing the policy would still be manual, but it is often relatively easy, since the vulnerability-exploiting portion of the incoming traffic—say, an overly long field that causes a buffer overrun—is often easy to identify once the traffic has been parsed.

- Shield need not necessarily be implemented at the end host. It may be preferable in some cases, from an administration or performance point of view, to deploy Shield in a firewall or router, or even in a special-purpose box. However, these alternate deployment options have yet to be explored.
- One of the advantages of Shield is that shields can in principle be tested in a relatively simple way, verifying that some collection of traffic (test suites or real-world traces) is not interfered with. Automating this process would make the shield installation process even easier.
- Ensuring the secure, reliable and expeditious distribution of Shields is crucial. While releasing a patch enables attackers to reverse-engineer the patch to understand its corresponding vulnerability, and thus to exploit it, Shield makes reverse-engineering even easier since vulnerability signatures are spelled out in Shield policies. Therefore, Shield distribution and installation is in an even tighter race with the exploit-designing hacker.
- It is possible that Shield's design might prove useful when applied to the virus problem, since some viruses exploit a vulnerability in the application that is invoked when an infected file is opened. Today, most anti-virus software is signature-based, identifying specific exploits rather than vulnerabilities. Incorporating shield-like technology into anti-virus systems might allow them to protect against generic classes of viruses that use a particular infection method.

XII. Acknowledgement

Jon Pincus has given us insightful and constant advice since the idea formation stage of the Shield project. Jay Lorch gave us many thoughtful critiques on the first draft of this paper. Many Microsoft colleagues from the product side have graciously helped us with understanding various aspects of many vulnerabilities from the Microsoft Security Bulletin. They include: Tahsin Erdogan, Mike Howard, Kamen Moutafov, Riyaz Pishori, Yong Qu, Jiri Richter, David Ross, Chris Walker, Nengwu Zhu. We have also obtained and learned how to use stress test suites for a number of application level protocols from our product group colleagues, they are: Andu Balakrishnan, Lie Lu, Jiri Richter, and Pavel Yatsuk. Stephen Adams, Andrew Begel, and Zhe Yang offered us helpful discussions on our policy language design and interpreter implementation. Anthony Jones helped us with understanding WinSock LSP programming model. This work also benefited from our discussions with John Dunagan, Jitu Padhye, Stefan Savage, David Thaler, Nick Weaver, and Brian Zill. We are thankful to everyone's help.

References

[1] Mohd A. Bashar, Ganesh Krishnan, Markus G. Kuhn, Eugene H. Spafford, and S. S. Wagstaff Jr. Low-Threat Security Patches and Tools. In *IEEE International Conference on Software Maintenance*, October 1997.

[2] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, and Chris Wright. Timing the application of security patches for optimal uptime. In *LISA XVI*, November 2002.

[3] Byacc. http://dickey.his.com/byacc/byacc.html.

[4] Z. Chen, L. Gao, and K. Kwiat. Modeling the Spread of Active Worms. In *Infocomm*, 2003.

[5] Microsoft Security Bulletin MS01-033, November 2003.

[6] Microsoft Corp. URLScan Security Tool. http://www.microsoft.com/technet/security/URLScan.asp.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, January 1997.

[8] Gregory R. Ganger, Gregg Economou, and Stanley M. Bielski. Finding and Containing enemies within the walls with self-securing network interfaces. Technical Report CMU-CS-03-109, Carnegie Mellon University, January 2003.

[9] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of USENIX Security Symposium*, August 2001.

[10] Hung-Yun Hsieh and Raghupathy Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *ACM Mobicom*, September 2002.

[11] Anthony Jones and Jim Ohlund. *Network Programming for Microsoft Windows*. Microsoft Publishing, 2002.

[12] J. Klensin. *RFC 2821 - Simple Mail Transfer Protocol*, April 2001.

[13] Eddie Kohler, Bejie Chen, M. Frans Kaashoek, Robert Morris, and Massimiliano Poletto. Programming language techniques for modular router configurations. Technical Report LCS-TR-812, MIT Laboratory for Computer Science, 2000.

[14] G. Robert Malan, David Watson, and Farnam Jahanian. Transport and application protocol scrubbing. In *IEEE Infocomm*, 2000.

[15] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. http://www.computer.org/security/v1n4/j4wea.htm, 2003.

[16] David Moore, Colleen Shannon, and Jeffery Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *ACM Internet Measurement Workshop (IMW)*, 2002.

[17] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of IEEE Infocom*, April 2003.

[18] Microsoft Security Bulletin MS03-026, September 2003.

[19] Vern Paxson. *Flex - a scanner generator - Table of Contents*. http://www.gnu.org/software/flex/manual/.

[20] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, Dec 1999.

[21] J. Postel and J. Reynolds. *RFC 854 - Telnet Protocol Specification*, May 1983.

[22] J. Postel and J. Reynolds. *RFC 765 - FILE TRANSFER PROTOCOL (FTP)*, October 1985.

[23] Valentin Razmov and Daniel Simon. Practical Automated Filter Generation to Explicitly Enforce Implicit Input Assumptions. In *Proceedings of 17th Annual Computer Security Applications Conference*, New Orleans, LA, December 2001.

[24] Eric Rescorla. Security holes... Who cares? In *Proceedings of USENIX Security Symposium*, August 2003.

[25] *DCE 1.1: Remote Procedure Call*.

[26] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RFC1889 RTP: A Transport Protocol for Real-Time Applications*, January 1996.

[27] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2003.

[28] Richard Sharpe. Server message block. http://samba.anu.edu.au/cifs/docs/what-is-smb.html.

[29] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The earlybird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California at San Diego, 2003.

[30] Microsoft security bulletin ms02-039, January 2003.

[31] The Open Source Network Intrusion Detection System. http://www.snort.org/.

[32] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[33] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. Large Scale Malicious Code: A Research Agenda. http://www.cs.berkeley.edu/ñweaver/large_scale_malicious_code.pdf, 2003.

[34] Nick Weaver. The potential for very fast internet plagues. http://www.cs.berkeley.edu/nweaver/warhol.html.

[35] Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Labs Bristol, 2002.