Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection

Holger Dreger
TU München
dreger@in.tum.de

Anja Feldmann
TU München
feldmann@in.tum.de

Michael Mai
TU München
maim@in.tum.de

Vern Paxson
ICSI/LBNL
vern@icir.org

Robin Sommer
ICSI
robin@icir.org

Abstract

Many network intrusion detection systems (NIDS) rely on protocol-specific analyzers to extract the higher-level semantic context from a traffic stream. To select the correct kind of analysis, traditional systems exclusively depend on well-known port numbers. However, based on our experience, increasingly significant portions of today's traffic are not classifiable by such a scheme. Yet for a NIDS, this traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. In this paper, we discuss the design and implementation of a NIDS extension to perform dynamic application-layer protocol analysis. For each connection, the system first identifies potential protocols in use and then activates appropriate analyzers to verify the decision and extract higher-level semantics. We demonstrate the power of our enhancement with three examples: reliable detection of applications not using their standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers. Prototypes of our system currently run at the border of three large-scale operational networks. Due to its success, the bot-detection is already integrated into a dynamic inline blocking of production traffic at one of the sites.

1 Introduction

Network intrusion detection systems (NIDSs) analyze streams of network packets in order to detect attacks and, more generally, violations of a site's security policy. NIDSs often rely on protocol-specific analyzers to extract the higher-level semantic context associated with a traffic stream, in order to form more reliable decisions about if and when to raise an alarm [38]. Such analysis can be quite sophisticated, such as pairing up a stream of replies with previously pipelined requests, or extracting the specific control parameters and data items associated with a transaction.

To select the correct analyzer for some traffic, a NIDS faces the challenge of determining which protocol is in use before it even has a chance to inspect the packet stream. To date, NIDSs have resolved this difficulty by assuming use of a set of well-known ports, such as those assigned by IANA [19], or those widely used by convention. If, however, a connection does not use one of these recognized ports—or misappropriates the port designated for a different application—then the NIDS faces a quandary: how does it determine the correct analyzer?

In practice, servers indeed do not always use the port nominally associated with their application, either due to benign or malicious intent. Benign examples include users who run Web or FTP servers on alternate ports because they lack administrator privileges. Less benign, but not necessarily malicious, examples include users that run servers offering non-Web applications on port 80/tcp in order to circumvent their firewall. In fact, some recently emerging application-layer protocols are *designed* to work without any fixed port, primarily to penetrate firewalls and escape administrative control. A prominent example is the voice-over-IP application Skype [2], which puts significant efforts into escaping restrictive firewalls. Sometimes such applications leverage a common protocol and its well-known port, like HTTP, to *tunnel* their payload not just through the firewall but even through application layer proxies. In these cases, analyzing the application's traffic requires first analyzing and stripping off the outer protocol before the NIDS can comprehend the semantics of the inner protocol. Similarly, we know from operational experience that attackers can attempt to evade security monitoring by concealing their traffic on non-standard ports or on ports assigned to different protocols: trojans installed on compromised hosts often communicate on non-standard ports; many botnets use the IRC protocol on ports other than 666x/tcp; and pirates build file-distribution networks using hidden FTP servers on ports other than 21/tcp.

It is therefore increasingly crucial to drive protocol-specific analysis using criteria other than ports. Indeed, a recent study [37] found that at a large university about 40% of the external traffic could not be classified by a port-based heuristic. For a NIDS, this huge amount of traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. Likewise, it is equally pressing to inspect whether traffic on standard ports indeed corresponds to the expected protocol. Thus, NIDSs need the capability of examining such traffic in-depth, including decapsulating an outer protocol layer in order to then examine the one tunneled inside it.

However, none of the NIDSs which are known to us, including Snort [34], Bro [31], Dragon [14], and IntruShield [20], use any criteria other than ports for their protocol-specific analysis. As an initial concession to the problem, some systems ship with signatures—characteristic byte-level payload patterns—meant to *detect* the use of a protocol on a none-standard port. But all only report the mere fact of finding such a connection, rather than adapting their analysis to the dynamically detected application protocol. For example, none of these systems can extract URLs from HTTP sessions on ports other than the statically configured set of ports.[1] With regards to decapsulating tunnels, a few newer systems can handle special cases, e.g., McAfee's IntruShield system [20] can unwrap the SSL-layer of HTTPS connections when provided with the server's private key. However, the decision that the payload *is* SSL is still based on the well-known port number of HTTPS.

In this paper we discuss the design, implementation, deployment, and evaluation of an extension to a NIDS to perform dynamic application-layer protocol analysis. For each connection, the system identifies the protocol in use and activates appropriate analyzers. We devise a general and flexible framework that (i) supports multiple ways to recognize protocols, (ii) can enable multiple protocol analyzers in parallel, (iii) copes with incorrect classifications by disabling protocol analyzers, (iv) can pipeline analyzers to dynamically decapsulate tunnels, and (v) provides performance sufficient for high-speed analysis.

We demonstrate the power our enhancement provides with three examples: (i) *reliable* detection of applications not using their standard ports, (ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based botnet clients and servers. The prototype system currently runs at the border of the University of California, Berkeley (UCB), the Münchener Wissenschaftsnetz (Munich Scientific Network, MWN), and the Lawrence Berkeley National Laboratory (LBNL). These deployments have already exposed a significant number of security incidents, and, due to its success, the staff of MWN has integrated bot-detection into its operations, using it for dynamic inline blocking of production traffic.

The remainder of this paper is organized as follows: §2 presents the three network environments that we use for our study. In §3 we analyze the potential of non-port-based protocol detection and discuss the limitations of existing NIDSs. In §4 we present the design and implementation of our dynamic architecture and discuss the trade-offs one faces in practice. §5 demonstrates the benefits of the dynamic architecture with three example applications. In §6 we evaluate the performance of our implementation in terms of CPU usage and detection capabilities. Finally in §7 we summarize our experience.

2 Environments and Dataset

The impetus for performing protocol analysis free of any assumptions regarding applications using standard ports arose from our operational experiences with NIDSs at three large-scale network environments: the *University of California, Berkeley (UCB)*, the *Münchener Wissenschaftsnetz (Munich Scientific Network, MWN)* and the *Lawrence Berkeley National Laboratory (LBNL)* [10]. We found that increasingly significant portions of the traffic at these sites were not classifiable using well-known port numbers. Indeed, at UCB 40% of all packets fall into this category [37].

All three environments support high volumes of traffic. At UCB, external traffic currently totals about 5 TB/day, with three 2 Gbps uplinks serving about 45,000 hosts on the main campus plus several affiliated institutes. The MWN provides a 1 Gbps upstream capacity to roughly 50,000 hosts at two major universities along with additional institutes, totaling 1-3 TB a day. LBNL also utilizes a 1 Gbps upstream link, transferring about 1.5 TB a day for roughly 13,000 hosts.

Being research environments, the three networks' security policies emphasize relatively unfettered connectivity. The border routers impose only a small set of firewall restrictions (e.g., closing ports exploited by major worms). MWN uses a more restrictive set of rules in order to close ports used by the major peer-to-peer (P2P) applications; however, since newer P2P applications circumvent such port-based blocking schemes, MWN is moving towards a dynamic traffic filtering/shaping system. In a first step it leverages NAT gateways[16] used to provide Internet access to most student residences, and the IPPP2P system for detecting peer-to-peer traffic [21].

In §5 we report on our experiences with running three different example applications of our extended NIDS on live traffic. To enable a systematic evaluation (see §3.2 and §6), we captured a 24-hour *full* trace

[1] To keep our terminology simple, we will refer to a single fixed port when often this can be extended to a fixed set of ports.

at MWN's border router on October 11, 2005, using a high-performance Endace DAG capturing card [13]. The trace encompasses 3.2 TB of data in 6.3 billion packets and contains 137 million distinct connections. 76% of all packets are TCP. The DAG card did not report any packet losses.

3 Analysis of the Problem Space

Users have a variety of reasons for providing servicing on non-standard ports. For example, a site's policy might require private services (such as a Web server) to run on an unprivileged, often non-standard, port. Such private servers frequently do not run continuously but pop up from time to time, in contrast to business-critical servers. From our operational experience, in open environments such servers are common and not viewed as any particular problem. However, compromised computers often also run servers on non-standard ports, for example to transfer sometimes large volumes of pirated content. Thus, some servers on non-standard port are benign, others are malicious; the question of how to treat these, and how to distinguish among them, must in part be answered by the site's security policy.

In addition, users also use standard ports for running applications other than those expected on the ports, for example to circumvent security or policy enforcement measures such as firewalls, with the most prevalent example being the use of port 80/tcp to run P2P nodes. A NIDS should therefore not assume that every connection on HTTP's well-known port is indeed a communication using the HTTP protocol; or, even if it *is* well-formed HTTP, that it reflects any sort of "Web" access. The same problem, although often unintentional and not malicious, exists for protocols such as IRC. These are not assigned a well-known privileged port but commonly use a set of well-known unprivileged ports. Since these ports are unprivileged, other applications, e.g., an FTP data-transfer connection, may happen to pick one of these ports. A NIDS therefore may encounter traffic from a different application than the one the port number indicates. Accordingly the NIDS has to have a way to detect the application layer protocol actually present in order to perform application-specific protocol analysis.

3.1 Approaches to Application Detection

Besides using port numbers, two other basic approaches for identifying application protocols have been examined in the literature: (i) statistical analysis of the traffic within a connection, and (ii) locating protocol-specific byte patterns in the connection's payload.

Previous work has used an analysis of interpacket delays and packet size distribution to distinguish interactive applications like chat and remote-login from bulk-transfer applications such as file transfers [41]. In some particular contexts these techniques can yield good accuracy, for example to separate Web-chat from regular Web surfing [8]. In general, these techniques [29, 35, 23, 40], based on statistical analysis and/or or machine learning components, have proven useful for classifying traffic into broad classes such as interactive, bulk transfer, streaming, or transactional. Other approaches model characteristics of individual protocols by means of decision trees [39] or neural networks [12].

The second approach—using protocol-specific, byte-level payload patterns, or "signatures"—takes advantage of a popular misuse detection technique. Almost all virus-scanner and NIDSs incorporate signatures into their analysis of benign vs. malicious files or network streams. For protocol recognition, we can use such signatures to detect application-specific patterns, such as components of an HTTP request or an IRC login sequence. However, there is no guarantee that such a signature is comprehensive. If it fails to detect all instances of a given application, it exhibits *false negatives*. In addition, if it incorrectly attributes a connection to a given application, it exhibits *false positives*.

We can also combine these types of approaches, first using statistical methods (or manual inspection) to cluster connections, and then extracting signatures, perhaps via machine learning techniques [17]; or using statistical methods to identify some applications, and signatures to identify others [41] or to refine the classification, or to combine ports, content-signatures, and application-layer information [6].

In the context of NIDSs, signature-based approaches are particularly attractive because many NIDSs already provide an infrastructure for signature-matching (§3.3), and often signatures yield tighter protocol identification capabilities.

3.2 Potential of a Signature Set

To evaluate how often common protocols use non-standard ports, and whether signatures appear capable of detecting such uses, we examine a 24-hour full trace of MWN's border router, `mwn-full-packets`. To do so we use the large, open source collection of application signatures included with the *l7-filter* system [24]. To apply these signatures to our trace, we utilize the signature matching engine of the open source NIDS Bro [31, 38]. Rather than running the l7-filter system itself, which is part of the Linux netfilter framework [30], we convert the signatures into Bro's syntax, which gives us the advantages of drawing upon Bro's built-in trace processing, connection-oriented analysis, and powerful signature-matching engine. We note however that while Bro and

l7-filter perform the matching in a similar way, varying internal semantics can lead to slightly different results, as with any two matching engines [38].

We begin by examining the breakdown of connections by the destination port seen in initial SYN packets. Table 1 shows all ports accounting for more than one percent of the connections. Note that for some ports the number of raw connections can be misleading due to the huge number of scanners and active worms, e.g., ports 445, 1042, and 1433. We consider a connection unsuccessful if it either does not complete an initial TCP handshake, or it does but does not transfer any payload. Clearly, we cannot identify the application used by such connections given no actual contents.

We make two observations. First, port-based protocol identification offers little assistance for most of the connections using unprivileged ports (totaling roughly 5.6 million connections). Second, the dominance of port 80 makes it highly attractive as a place for hiding connections using other applications. While an HTTP protocol analyzer might notice that such connections do not adhere to the HTTP protocol, we cannot expect that the analyzer will then go on to detect the protocol actually in use.

To judge if signatures can help improve application identification, for each of a number of popular apparent services (HTTP, IRC, FTP, and SMTP) we examined the proportion identified by the l7-filter signatures as indeed running that protocol. Table 2 shows that most of the successful connections trigger the expected signature match (thus, the signature quality is reasonable). Only for FTP we observe a higher percentage of false negatives. This can be improved using a better FTP signature. However, we also see that for each protocol we find matches for connections on unexpected ports, highlighting the need for closer inspection of their payload.

The differences in Table 2 do not necessarily all arise due to false negatives. Some may stem from connections without enough payload to accurately determine their protocol, or those that use a different protocol. Regarding this latter, Table 3 shows how often a different protocol appears on the standard ports of HTTP, IRC, FTP and SMTP.

While inspecting the results we noticed that a connection sometimes triggers more than one signature. More detailed analysis reveals that l7-filter contains some signatures that are too general. For example, the signature for the Finger protocol matches simply if the first two characters at the beginning of the connection are printable characters. Such a signature will be triggered by a huge number of connections not using Finger. Another example comes from the "whois" signature. Accordingly, the data in Table 3 ignores matches by these two signatures.

Port	HTTP	IRC	FTP	SMTP	Other	No sig.
80	92.2M	59	0	0	41.1K	1.2M
6665-6669	1.2K	71.7K	0	0	4.2	524
21	0	0	98.0K	2	2.3K	52.5K
25	459	2	749	1.4M	195	31.9K

Table 3: Signature-based detection vs. port-based detection for well-known ports (# connections).

Overall, the results show that the problem we pose does indeed already manifest operationally. Furthermore, because security analysis entails an adversary, what matters most is not the proportion of benign connections using ports other than those we might expect, but the prevalence of malicious connections doing so. We later discuss a number of such instances found operationally.

3.3 Existing NIDS Capabilities

Today's spectrum of intrusion detection and prevention systems offer powerful ways for detecting myriad forms of abuse. The simpler systems rely on searching for byte patterns within a packet stream, while the more complex perform extensive, stateful protocol analysis. In addition, some systems offer anomaly-based detection, comparing statistical characteristics of the monitored traffic against "normal network behavior," and/or specification-based detection, testing the characteristics against explicit specifications of allowed behavior.

For analyzing application-layer protocols, all systems of which we are aware depend upon port numbers.[2] While some can use signatures to *detect* other application-layer protocols, all only perform detailed protocol analysis for traffic identified via specific ports. Commercial systems rarely make details about their implementation available, and thus we must guess to what depth they analyze traffic. However, we have not seen an indication yet that any of them initiates stateful protocol analysis based on other properties than specific ports.

The most widely deployed open source NIDS, Snort [34], does not per se ship with signatures for detecting protocols. However the Snort user community constantly contributes new open signatures [4], including ones for detecting IRC and FTP connections. Traditionally, Snort signatures are raw byte patterns. Newer versions of Snort also support regular expressions. Another open source NIDS, Bro [31], ships with a *backdoor* [41] analyzer which follows two approaches. First, to detect interactive traffic it examines inter-packet intervals and packet size distributions. Second, for several

[2] `DSniff` [11] is a network sniffer that extracts protocol-specific usernames and passwords independent of ports. Its approach is similar to ours in that it uses a set of patterns to recognize protocols. It is however not a NIDS and does not provide any further payload analysis.

Port	Connections	% Conns.	Successful	% Success.	Payload [GB]	% Payload
80	97,106,281	70.82%	93,428,872	68.13	2,548.55	72.59
445	4,833,919	3.53%	8,398	0.01	0.01	0.00
443	3,206,369	2.34%	2,855,457	2.08	45.22	1.29
22	2,900,876	2.12%	2,395,394	1.75	59.91	1.71
25	2,533,617	1.85%	1,447,433	1.05	60.00	1.71
1042	2,281,780	1.66%	35	0.00	0.01	0.00
1433	1,451,734	1.06%	57	0.00	0.06	0.00
135	1,431,155	1.04%	62	0.00	0.00	0.00
< 1024	114,747,251	83.68%	101,097,769	73.73	2,775.15	79.05
≥ 1024	22,371,805	16.32%	5,604,377	4.08	735.62	20.95

Table 1: Ports accounting for more than 1% of the `mwn-full-packets` connections.

Method	HTTP	%	IRC	%	FTP	%	SMTP	%
Port (successful)	93,429K	68.14	75,876	0.06	151,700	0.11	1,447K	1.06
Signature	94,326K	68.79	73,962	0.05	125,296	0.09	1,416K	1.03
on expected port	92,228K	67.3	71,467	0.05	98,017	0.07	1,415K	1.03
on other port	2,126K	1.6	2,495	0.00	27,279	0.02	265	0.00

Table 2: Comparison of signature-based detection vs. port-based detection (# connections).

well-known protocols like HTTP, FTP and IRC, it scans the analyzed payload for hard-coded byte patterns. In addition, Bro features a signature matching engine [38] capable of matching the reassembled data stream of a connection against regular expression byte-patterns and leveraging the rich state of Bro's protocol decoders in the process. The engine allows for bidirectional signatures, where one byte pattern has to match on a stream in one direction and another in the opposite direction. The commercial IntruShield system by Network Associates is primarily signature-based and ships with signatures for application detection, including SSH and popular P2P protocols. The technical details and the signatures do not appear accessible to the user. Therefore, it is unclear which property of a packet/stream triggers which signature or protocol violation. We also have some experience with Enterasys' Dragon system. It ships with a few signatures to match protocols such as IRC, but these do not appear to then enable full protocol analysis.

3.4 NIDS Limitations

It is useful to distinguish between the capability of detecting that a given application protocol is in use, versus then being able to continue to analyze that instance of use. Merely detecting the use of a given protocol can already provide actionable information; it might constitute a policy violation at a site for which a NIDS could institute blocking without needing to further analyze the connection. However, such a coarse-grained "go/no-go" capability has several drawbacks:

1. In some environments, such a policy may prove too restrictive or impractical due to the sheer size and diversity of the site. As user populations grow, the likelihood of users wishing to run legitimate servers on alternate ports rises.

2. Neither approach to application detection (byte patterns or statistical tests) is completely accurate (see §3.2). Blocking false-positives hinders legitimate operations, while failing to block false-negatives hinders protection.

3. Protocols that use non-fixed ports (e.g., Gnutella) can only be denied or allowed. Some of these, however, have legitimate applications as well as applications in violation of policy. For example, BitTorrent [5] might be used for distributing open-source software. Or, while a site might allow the use of IRC, including on non-standard ports, it highly desires to analyze all such uses in order to detect botnets.

In addition, some protocols are fundamentally difficult to detect with signatures, for example unstructured protocols such as Telnet. For Telnet, virtually any byte pattern at the beginning is potentially legitimate. Telnet can only be detected heuristically, by looking for plausible login dialogs [31]. Another example is DNS, a binary protocol with no protocol identifier in the packet. The DNS header consists of 16-bit integers and bit fields which can take nearly arbitrary values. Thus, reliably detecting DNS requires checking the consistency across many fields. Similar problem exist for other binary protocols.

Another difficulty is that if an attacker knows the signatures, they may try to avoid the byte patterns that trigger the signature match. This means one needs "tight" signatures which comprehensively capture any use of a protocol for which an attacked end-system might engage. Finding such "tight" signatures can be particularly difficult due to the variety of end-system implementations and their idiosyncrasies.

4 Architecture

In this section we develop a framework for performing dynamic application-layer protocol analysis. Instead of a static determination of what analysis to perform based on port numbers, we introduce a processing path that dynamically adds and removes analysis components. The scheme uses a protocol detection mechanism as a trigger to activate analyzers (which are then given the entire traffic stream to date, including the portion already scanned by the detector), but these analyzers can subsequently decline to process the connection if they determine the trigger was in error. Currently, our implementation relies primarily on signatures for protocol detection, but our design allows for arbitrary other heuristics.

We present the design of the architecture in §4.1 and a realization of the architecture for the open-source NIDS Bro in §4.2. We finish with a discussion of the tradeoffs that arise in §4.3.

4.1 Design

Our design aims to achieve flexibility and power-of-expression, yet to remain sufficiently efficient for operational use. We pose the following requirements as necessary for these goals:

Detection scheme independence: The architecture must accommodate different approaches to protocol detection (§3.1). In addition, we should retain the possibility of using multiple techniques in parallel (e.g., complementing port-based detection with signature-based detection).

Dynamic analysis: We need the capability of dynamically enabling or disabling protocol-specific analysis at any time during the lifetime of a connection. This goal arises because some protocol detection schemes cannot make a decision upon just the first packet of a connection. Once they make a decision, we must trigger the appropriate protocol analysis. Also, if the protocol analysis detects a false positive, we must have the ability to stop the analysis.

Modularity: Reusable components allow for code reuse and ease extensions. This becomes particularly important for dealing with multiple network substacks (e.g., IP-within-IP tunnels) and performing in parallel multiple forms of protocol analysis (e.g., decoding in parallel with computing packet-size distributions).

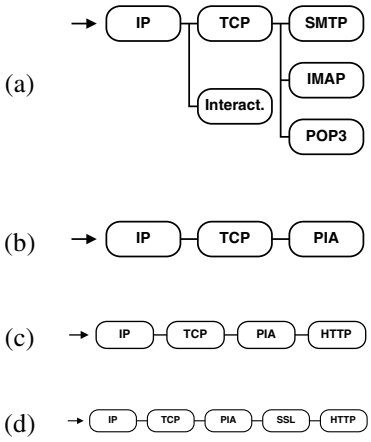

Figure 1: Example analyzer trees.

Efficiency: The additional processing required by the extended NIDS capabilities must remain commensurate with maintaining performance levels necessary for processing high-volume traffic streams.

Customizability: The combination of analysis to perform needs to be easily adapted to the needs of the local security policy. In addition, the trade-offs within the analysis components require configuration according to the environment.

To address these requirements we switch from the traditional static data analysis path to a dynamic one inside the NIDS's core. Traditional port-based NIDSs decide at the time when they receive the first packet of each connection which analyses to perform. For example, given a TCP SYN packet with destination port 80, the NIDS will usually perform IP, TCP, and HTTP analysis for all subsequent packets of this flow. Our approach, on the other hand, relies on a per-connection data structure for representing the *data path*, which tracks what the system learns regarding what analysis to perform for the flow. If, for example, the payload of a packet on port 80/tcp—initially analyzed as HTTP—looks like an IRC session instead, we replace the HTTP analysis with IRC analysis.

We provide this flexibility by associating a tree structure with each connection. This tree represents the data path through various analysis components for all information transmitted on that connection (e.g., Figure 1(a)). Each node in this tree represents a self-contained unit of analysis, an *analyzer*. Each analyzer performs some kind

of analysis on data received via an *input channel*, subsequently providing data via an *output channel*. The input channel of each node connects to an output channel of its data supplier (its predecessor in the data path tree). The input channel of the tree's root receives packets belonging to the connection/flow. Each intermediate node receives data via its input channel and computes analysis results, passing the possibly-transformed data to the next analyzer via its output channel.

Figure 1(a) shows an example of a possible analyzer tree for decoding email protocols. In this example, all analyzers (except `INTERACTIVE`) are responsible for the decoding of their respective network protocols. The packets of the connection first pass through the IP analyzer, then through the TCP analyzer. The output channel of the latter passes in replica to three analyzers for popular email protocols: SMTP, IMAP, and POP3. (Our architecture might instantiate such a tree for example if a signature match indicates that the payload looks like email but does not distinguish the application-layer protocol.) Note, though, that the analyzers need not correspond to a protocol, e.g., `INTERACTIVE` here, which examines inter-packet time intervals to detect surreptitious interactive traffic [41], performing its analysis in parallel to, and independent from, the TCP and email protocol analyzers.

To enable *dynamic* analysis, including analysis based on application-layer protocol identification, the analyzer tree changes over time. Initially, the analyzer tree of a new connection only contains those analyzers definitely needed. For example, if a flow's first packet uses TCP for transport, the tree will consist of an IP analyzer followed by a TCP analyzer.

We delegate application-layer protocol identification to a *protocol identification analyzer* (`PIA`), which works by applying a set of protocol detection heuristics to the data it receives. We insert this analyzer into the tree as a leaf-node after the TCP or UDP analyzer (see Figure 1(b)). Once the `PIA` detects a match for a protocol, it instantiates a child analyzer to which it then forwards the data it receives (see Figure 1(c)). However, the `PIA` also continues applying its heuristics, and if it finds another match it instantiates additional, or alternative, analyzers.

The analyzer tree can be dynamically adjusted throughout the entire lifetime of a connection by inserting or removing analyzers. Each analyzer has the ability to insert or remove other analyzers on its input and/or output channel. Accordingly, the tree changes over time. Initially the `PIA` inserts analyzers as it finds matching protocols. Subsequently one of the analyzers may decide that it requires support provided by a missing analyzer and instantiates it; for instance, an IRC analyzer that learns that the connection has a compressed payload can insert a decompression analyzer as its predecessor.

If an analyzer provides data via its output channel, selecting successors becomes more complicated, as not all analyzers (including the TCP analyzer) have the capability to determine the protocol to which their output data conforms. In this case the analyzer can choose to instantiate *another* `PIA` and delegate to it the task of further inspecting the data. Otherwise it can simply instantiate the appropriate analyzer; see Figure 1(d) for the example of a connection using HTTP over SSL.

Finally, if an analyzer determines that it cannot cope with the data it receives over its input channel (e.g., because the data does not conform to its protocol), it removes its subtree from the tree.

This analyzer-tree design poses a number of technical challenges, ranging from the semantics of "input channels", to specifics of protocol analyzers, to performance issues. We now address each in turn.

First, the semantics of "input channels" differ across the network stack layers: some analyzers examine packets (e.g., IP, TCP, and protocols using UDP for transport), while others require byte-streams (e.g., protocols using TCP for transport). As the `PIA` can be inserted into arbitrary locations in the tree, it must cope with both types. To do so, we provide two separate input channels for each analyzer, one for packet input and one for stream input. Each analyzer implements the channel(s) suitable for its semantics. For example, the TCP analyzer accepts packet input and reassembles it into a payload stream, which serves as input to subsequent stream-based analyzers. An RPC analyzer accepts both packet and stream input, since RPC traffic can arrive over both UDP packets and TCP byte streams.

Another problem is the difficulty—or impossibility—of starting a protocol analyzer in the middle of a connection. For example, an HTTP analyzer cannot determine the correct HTTP state for such a partial connection. However, most non-port-based protocol detection schemes can rarely identify the appropriate analyzer(s) upon inspecting just the first packet of a connection. Therefore it is important that the `PIA` buffers the beginning of each input stream, up to a configurable threshold (default 4KB in our implementation). If the `PIA` decides to insert a child analyzer, it first forwards the data in the buffer to it before forwarding new data. This gives the child analyzer a chance to receive the total payload if detection occurred within the time provided by the buffer. If instantiation occurs only after the buffer has overflowed, the `PIA` only instantiates analyzers capable of resynchronizing to the data stream, i.e., those with support for partial connections.

Finally, for efficiency the `PIA` requires very lightweight execution, as we instantiate at least one for every flow/connection. To avoid unnecessary resource consumption, our design factors out the user configura-

tion, tree manipulation interface, and functions requiring permanent state (especially state independent of a connection's lifetime) into a single central management component which also instantiates the initial analyzer trees.

In summary, the approach of generalizing the processing path to an analyzer tree provides numerous new possibilities while addressing the requirements. We can: *(i)* readily plug in new protocol detection schemes via the `PIA`; *(ii)* dynamically enable and disable analyzers at any time (protocol semantics permitting); *(iii)* enable the user to customize and control the processing via an interface to the central manager; *(iv)* keep minimal the overhead of passing data along the tree branches; *(v)* support pure port-based analysis using a static analyzer tree installed at connection initialization; and *(vi)* support modularity by incorporating self-contained analyzers using a standardized API, which allows any protocol analyzer to also serve as a protocol *verifier*.

4.2 Implementation

We implemented our design within the open-source *Bro* NIDS, leveraging both its already existing set of protocol decoders and its signature-matching engine. Like other systems, Bro performs comprehensive protocol-level analysis using a static data path, relying on port numbers to identify application-layer protocols. However, its modular design encourages application-layer decoders to be mainly self-contained, making it feasible to introduce a dynamic analyzer structure as discussed in §4.1.

We implemented the `PIA`, the analyzer trees, and the central manager, terming this modification of Bro as `PIA-Bro`; for details see [26]. We use signatures as our primary protocol-detection heuristic (though see below), equipping the `PIA` with an interface to Bro's signature-matching engine such that analyzers can add signatures corresponding to their particular protocols. For efficiency, we restricted the signature matching to the data buffered by the `PIA`s; previous work[36, 28] indicates that for protocol detection it suffices to examine at most a few KB at the beginning of a connection. By skipping the tails, we can avoid performing pattern matching on the bulk of the total volume, exploiting the heavy-tailed nature of network traffic [32].

In addition to matching signatures, our implementation can incorporate other schemes for determining the right analyzers to activate. First, the `PIA` can still activate analyzers based on a user-configured list of well-known ports.[3] In addition, each protocol analyzer can register a specific detection function. The `PIA` then calls this function for any new data chunk, allowing the use of arbitrary heuristics to recognize the protocol. Finally, leveraging the fact that the central manager can store state, we also implemented a *prediction table* for storing anticipated future connections along with a corresponding analyzer. When the system eventually encounters one of these connections, it inserts the designated analyzer into the tree. (See §5.2 below for using this mechanism to inspect FTP data-transfer connections.) Together these mechanisms provide the necessary flexibility for the connections requiring dynamic detection, as well as good performance for the bulk of statically predictable connections.

As Bro is a large and internally quite complex system, we incrementally migrate its protocol analyzers to use the new framework. Our design supports this by allowing old-style and new-style data paths to coexist: for those applications we have adapted, we gain the full power of the new architecture, while the other applications remain usable in the traditional (static ports) way.

For our initial transition of the analyzers we have concentrated on protocols running on top of TCP. The Bro system already encapsulates its protocol decoders into separate units; we redesigned the API of these units to accommodate the dynamic analyzer structure. We have converted four of the system's existing application-layer protocol decoders to the new API: FTP, HTTP, IRC, and SMTP.[4] The focus on TCP causes the initial analyzer tree to always contain the IP and TCP analyzers. Therefore we can leverage the existing static code and did not yet have to adapt the IP and TCP logic to the new analyzer API. We have, however, already moved the TCP stream reassembly code into a separate "Stream" analyzer. When we integrate UDP into the framework, we will also adapt the IP and TCP components.

The Stream analyzer is one instance of a *support analyzer* which does not directly correspond to any specific protocol. Other support analyzers provide functionality such as splitting the payload-stream of text-based protocols into lines, or expanding compressed data.[5] We have not yet experimented with pipelining protocol analyzers such as those required for tunnel decapsulation, but intend to adapt Bro's SSL decoder next to enable us to analyze HTTPS and IMAPS in a pipelined fashion when we provide the system with the corresponding secret key.

[3] This differs from the traditional Bro, where the set of well-known ports is hard-coded.

[4] Note that it does not require much effort to convert an existing application-layer analyzer to the new API. For example, the SMTP analyzer took us about an hour to adapt.

[5] Internally, these support analyzers are implemented via a slightly different interface, see [26] for details.

4.3 Trade-Offs

Using the PIA architecture raises some important trade-offs to consider since protocol recognition/verification is now a multi-step process. First, the user must decide what kinds of signatures to apply to detect *potential* application-layer protocols. Second, if a signature matches it activates the appropriate protocol-specific analyzer, at which point the system must cope with possible false positives; when and how does the analyzer fail in this case? Finally, we must consider how an attacker can exploit these trade-offs to subvert the analysis.

The first trade-off involves choosing appropriate signatures for the protocol detection. On the one hand, the multi-step approach allows us to *loosen* the signatures that initially detect protocol candidates. Signatures are typically prone to false alerts, and thus when used to generate alerts need to be specified as tight as possible—which in turn very often leads to false negatives, i.e., undetected protocols in this context. However, by relying on analyzers *verifying* protocol conformance after a signature match, false positives become more affordable. On the other hand, signatures should not be *too* lose: having an analyzer inspect a connection is more expensive than performing pure pattern matching. In addition, we want to avoid enabling an attacker to trigger expensive protocol processing by deliberately crafting bogus connection payloads.

Towards these ends, our implementation uses bidirectional signatures [38], which only match if *both* endpoints of a connection appear to participate in the protocol. If an attacker only controls one side (if they control both, we are sunk in many different ways), they thus cannot force activation of protocol analyzers by themselves. In practice, we in fact go a step further: before assuming that a connection uses a certain protocol, the corresponding analyzer must also *parse* something meaningful for both directions. This significantly reduces the impact of false positives. Figure 2 shows an example of the signature we currently use for activating the HTTP analyzer. (We note that the point here is not about signature quality; for our system, signatures are just one part of the NIDS's configuration to be adapted to the user's requirements.)

Another trade-off to address is *when* to decide that a connection uses a certain protocol. This is important if the use of a certain application violates a site's security policy and should cause the NIDS to raise an alert. A signature-match triggers the activation of an analyzer that analyzes and verifies the protocol usage. Therefore, before alerting, the system waits until it sees that the analyzer is capable of handling the connection's payload. In principle, it can only confirm this with certainty once the connection completes. In practice, doing so will delay alerts significantly for long-term connections. Therefore our implementation assumes that if the analyzer can parse the connection's *beginning*, the rest of the payload will also adhere to the same protocol. That is, our system reports use of a protocol if the corresponding analyzer is (still) active after the exchange of a given volume of payload, or a given amount of time passes (both thresholds are configurable).

Another trade-off stems from the question of protocol verification: at what point should an analyzer indicate that it *cannot* cope with the payload of a given connection? Two extreme answers: (i) reject immediately when something occurs not in accordance with the protocol's definition, or (ii) continue parsing come whatever may, in the hope that eventually the analyzer can resynchronize with the data stream. Neither extreme works well: real-world network traffic often stretches the bounds of a protocol's specification, but trying to parse the entire stream contradicts the goal of verifying the protocol. The right balance between these extremes needs to be decided on a per-protocol basis. So far, we have chosen to reject connections if they violate basic protocol properties. For example, the FTP analyzer complains if it does not find a numeric reply-code in the server's response. However, we anticipate needing to refine this decision process for instances where the distinction between clear noncompliance versus perhaps-just-weird behavior is less crisp.

Finally, an attacker might exploit the specifics of a particular analyzer, avoiding detection by crafting traffic in a manner that the analyzer believes reflects a protocol violation, while the connection's other endpoint still accepts or benignly ignores the data. This problem appears fundamental to protocol detection, and indeed is an instance of the more general problem of evasion-by-ambiguity [33, 18], and, for signatures, the vulnerability of NIDS signatures to attack variants. To mitigate this problem, we inserted indirection into the decision process: in our implementation, an analyzer *never* disables itself, even if it fails to parse its inputs. Instead, upon severe protocol violations it generates Bro events that a user-level policy script then acts upon. The default script is fully customizable, capable of extension to implementing arbitrary complex policies such as disabling the analyzer only after repeated violations. This approach fits with the Kerkhoff-like principle used by the Bro system: the code is open, yet sites code their specific policies in user-level scripts which they strive to keep secret.

5 Applications

We now illustrate the increased detection capabilities that stem from realizing the PIA architecture within Bro, using three powerful example applications: (i) reliable detection of applications running on non-standard ports,

```
signature http_server {                  # Server-side signature
  ip-proto == tcp                        # Examine TCP packets.
  payload /^HTTP\/[0-9]/                 # Look for server response.
  tcp-state responder                    # Match responder-side of connection.
  requires-reverse-signature http_client # Require client-side signature as well.
  enable "http"                          # Enable analyzer upon match.
}
signature http_client {                  # Client-side signature
  ip-proto == tcp                        # Examine TCP packets.
  payload /^[[:space:]]*GET[[:space:]]*/ # Look for requests [simplified]
  tcp-state originator                   # Match originator-side of connection.
}
```

Figure 2: Bidirectional signature for HTTP.

(ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based bot clients and servers. All three schemes run in day-to-day operations at UCB, MWN, and LBNL (see §2), where they have already identified a large number of compromised hosts which the sites' traditional security monitoring could not directly detect.

5.1 Detecting Uses of Non-standard Ports

As pointed out earlier, a PIA architecture gives us the powerful ability to verify protocol usage and extract higher-level semantics. To take advantage of this capability, we extended the reporting of PIA-Bro's analyzers. Once the NIDS knows which protocol a connection uses, it can leverage this to extract more semantic context. For example, HTTP is used by a wide range of other protocols as a transport protocol. Therefore, an alert such as "connection uses HTTP on a non-standard port 21012", while useful, does not tell the whole story; we would like to know *what* that connection then does. We extended PIA-Bro's HTTP analysis to distinguish the various protocols using HTTP for transport by analyzing the HTTP dialog. Kazaa, for example, includes custom headers lines that start with X-Kazaa. Thus, when this string is present, the NIDS generates a message such as "connection uses Kazaa on port 21021". We added patterns for detecting Kazaa, Gnutella, BitTorrent, Squid, and SOAP applications running over HTTP. In addition, the HTTP analyzer extracts the "Server" header from the HTTP responses, giving an additional indication of the underlying application.

We currently run the dynamic protocol detection for FTP, HTTP, IRC, and SMTP on the border routers of all three environments, though here we primarily report on experiences at UCB and MWN. As we have particular interest in the use of non-standard ports, and to reduce the load on PIA-Bro, we exclude traffic on the analyzers' well-known ports[6] from our analysis. (This setup prevents PIA-Bro from finding some forms of port abuse, e.g., an IRC connection running on the HTTP port. We postpone this issue to §6.)

At both UCB and MWN, our system quickly identified many servers[7] which had gone unnoticed. At UCB, it found within a day 6 internal and 17 remote FTP servers, 568/54830 HTTP servers (!), 2/33 IRC servers, and 8/8 SMTP servers running on non-standard ports. At MWN, during a similar period, we found 3/40 FTP, 108/18844 HTTP, 3/58 IRC, and 3/5 SMTP servers.

For FTP, IRC, and SMTP we manually checked whether the internal hosts were indeed running the detected protocol; for HTTP, we verified a subset. Among the checked servers we found only one false positive: PIA-Bro incorrectly flagged one SMTP server due to our choice regarding how to cope with false positives: as discussed in §4.3, we choose to not wait until the end of the connection before alerting. In this case, the SMTP analyzer correctly reported a protocol violation for the connection, but it did so only *after* our chosen maximal interval of 30 seconds had already passed; the server's response took quite some time. In terms of connections, HTTP is, not surprisingly, the most prevalent of the four protocols: at UCB during the one-day period, 99% of the roughly 970,000 reported off-port connections are HTTP. Of these, 28% are attributed to Gnutella, 22% to Apache, and 12% to Freechal [15]. At MWN, 92% of the 250,000 reported connections are HTTP, and 7% FTP (of these 70% were initiated by the same host). Of the HTTP connections, roughly 21% are attributed to BitTorrent, 20% to Gnutella, and 14% to SOAP.

That protocol analyzers can now extract protocol semantics not just for HTTP but also for the other protocols proves to be quite valuable. PIA-Bro generates detailed protocol-level log files for all connections. A short glance at, for example, an FTP log file quickly reveals whether an FTP server deserves closer attention. Figure 3 shows an excerpt of such a log file for an ob-

[6] For "well-known" we consider those for which a traditional Bro triggers application-layer analysis. These are port 21 for FTP, ports 6667/6668 for IRC, 80/81/631/1080/3128/8000/8080/8888 for HTTP (631 is IPP), and 25 for SMTP. We furthermore added 6665/6666/6668/6669/7000 for IRC, and 587 for SMTP as we encountered them on a regular basis. To further reduce the load on the monitor machines, we excluded a few high volume hosts, including the PlanetLab servers at UCB and the heavily accessed leo.org domain at MWN.

[7] In this context, a *server* is an IP address that accepts connections and participates in the protocol exchange. Due to NAT address space, we may underestimate or overestimate the number of actual hosts.

```
xxx.xxx.xxx.xxx/2373 > xxx.xxx.xxx.xxx/5560 start
response (220 Rooted Moron Version 1.00 4 WinSock ready...)
USER ops (logged in)
SYST (215 UNIX Type: L8)
[...]
LIST -al (complete)
TYPE I (ok)
SIZE stargate.atl.s02e18.hdtv.xvid-tvd.avi (unavail)
PORT xxx,xxx,xxx,xxx,xxx,xxx (ok)
*STOR stargate.atl.s02e18.hdtv.xvid-tvd.avi, NOOP (ok)
ftp-data video/x-msvideo 'RIFF (little-endian) data, AVI'
[...]
response (226 Transfer complete.)
[...]
QUIT (closed)
```

Figure 3: Application-layer log of an FTP-session to a compromised server (anonymized/edited for clarity).

viously compromised host at MWN. During a two-week period, we found such hosts in both environments, although UCB as well as MWN already deploy Snort signatures supposed to detect such FTP servers.

With `PIA-Bro`, any protocol-level analysis automatically extends to non-standard ports. For example, we devised a detector for HTTP proxies which matches HTTP requests going into a host with those issued by the same system to external systems. With the traditional setup, it can only report proxies on well-known ports; with `PIA-Bro` in place, it has correctly identified proxies inside the UCB and MWN networks running on different ports;[8] two of them were world-open.

It depends on a site's policy whether offering a service on a non-standard port constitutes a problem. Both university environments favor open policies, generally tolerating offering non-standard services. For the internal servers we identified, we verified that they meet at least basic security requirements. For all SMTP servers, for example, we ensured that they do not allow arbitrary relaying. One at MWN which did was quickly closed after we reported it, as were the open HTTP proxies.

5.2 Payload Inspection of FTP Data

According to the experience of network operators, attackers often install FTP servers on non-standard ports on machines that they have compromised. `PIA-Bro` now not only gives us a reliable way to detect such servers but, in addition, can *examine* the transferred files. This is an impossible task for traditional NIDSs, as FTP is a protocol for which for the data-transfer connections by design use arbitrary port combinations. For security monitoring, inspecting the *transferred* data for files exchanged via non-standard-port services enables alerts on sensitive files such as system database accesses or download/upload of virus-infected files. We introduced a new *file analyzer* to perform such analysis for FTP data connections, as well as for other protocols used to transfer files. When `PIA-Bro` learns, e.g., via its analysis of the control session, of an upcoming data transfer, it adds the expected connection to the dynamic prediction table (see §4.2). Once this connection is seen, the system instantiates a file analyzer, which examines the connection's payload.

The file analyzer receives the file's full content as a reassembled stream and can utilize any file-based intrusion detection scheme. To demonstrate this capability, our file-type identification for `PIA-Bro` leverages `libmagic` [25], which ships with a large library of file-type characteristics. This allows `PIA-Bro` to log the file-type's textual description as well as its MIME-type as determined by `libmagic` based on the payload at the beginning of the connection. Our extended FTP analyzer logs—and potentially alerts on—the file's content type. Figure 3 shows the result of the file type identification in the `ftp-data` line. The NIDS categorizes the data transfer as being of MIME type `video/x-msvideo` and, more specifically, as an AVI movie. As there usually are only a relatively small number of ftp-data connections, this mechanism imposes quite minimal performance overhead.

We envision several extensions to the file analyzer. One straight-forward improvement, suggested to us by the operators at LBNL, is to match a file's name with its actual content (e.g., a file `picture.gif` requested from a FTP server can turn out to be an executable). Another easy extension is the addition of an interface to a virus checker (e.g., Clam AntiVirus [7]). We also plan to adapt other protocol analyzers to take advantage of the file analyzer, such as TFTP (once `PIA-Bro` has support for UDP) and SMTP. TFTP has been used in the past by worms to download malicious code [3]. Similarly, SMTP can pass attachments to the file analyzer for inspection. SMTP differs from FTP in that it transfers files in-band, i.e., inside the SMTP session, rather than out-of-band over a separate data connection. Therefore, for SMTP there is no need to use the dynamic prediction table. Yet, we need the capabilities of `PIA-Bro` to pipeline the analyzers: first the SMTP analyzer strips the attachments' MIME-encoding, then the file analyzer inspects the file's content.

5.3 Detecting IRC-based Botnets

Attackers systematically install trojans together with *bots* for remote command execution on vulnerable systems. Together, these form large *botnets* controlled by a human *master* that communicates with the bots by sending commands. Such commands can be to flood a victim, send spam, or sniff confidential information such as passwords. Often, thousands of individual bots are controlled

[8] As observing both internal as well as outgoing requests at *border* is rather unusual, this detection methodology generally detects proxies other than the site's intended ones.

by a single master [1], constituting one of the largest security threats in today's Internet.

The IRC protocol [22] is a popular means for communication within botnets as it has some appealing properties for remote control: it provides *public channels* for one-to-many communication, with *channel topics* well-suited for holding commands; and it provides *private channels* for one-to-one communication.

It is difficult for a traditional NIDS to reliably detect members of IRC-based botnets. Often, the bots never connect to a standard IRC server—if they did they would be easy to track down—but to a separate *bot-server* on some non-IRC port somewhere in the Internet. However, users also sometimes connect to IRC servers running on non-standard ports for legitimate (non-policy-violating) purposes. Even if a traditional NIDS has the capability of detecting IRC servers on non-standard ports, it lacks the ability to then distinguish between these two cases.

We used `PIA-Bro` to implement a reliable bot-detector that has already identified a significant number of bot-clients at MWN and UCB. The detector operates on top of the IRC analyzer and can thus perform protocol-aware analysis of *all* detected IRC sessions. To identify a bot connection, it uses three heuristics. First, it checks if the client's nickname matches a (customizable) set of regular expression patterns we have found to be used by some botnets (e.g., a typical botnet "nick" identifier is `[0]CHN|3436036`). Second, it examines the channel topics to see if it includes a typical botnet command (such as `.advscan`, which is used by variants of the SdBot family[1]). Third, it flags new clients that establish an IRC connection to an already identified bot-server as bots. The last heuristic is very powerful, as it leverages the state that the detector accumulates over time and does not depend on any particular payload pattern. Figure 4 shows an excerpt of the list of known bots and bot-servers that one of our operational detectors maintains. This includes the server(s) contacted as well as the timestamp of the last alarming IRC command. (Such timestamps aid in identifying the owner of the system in NAT'd or DHCP environments.) For the servers, the list contains channel information, including topics and passwords, as well as the clients that have contacted them.

At MWN the bot-detector quickly flagged a large number of bots. So far, it has identified more than 100 distinct local addresses. To exclude the danger of false positives, we manually verified a subset. To date, we have not encountered any problems with our detection. Interestingly, at UCB there are either other kinds of bots, or not as many compromised machines; during a two-week time period we reported only 15 internal hosts to the network administrators. We note that the NIDS, due to only looking for patterns of *known* bots, certainly misses victims; this is the typical drawback of such a misuse-detection approach, but one we can improve over time as we learn more signatures through other means.

```
Detected bot-servers:
IP1 - ports 9009,6556,5552 password(s) <none> last 18:01:56
 channel #vec:
 topic ".asc pnp 30 5 999 -b -s|.wksescan 10 5 999 -b -s|[...]"
 channel #hv:
 topic ".update http://XXX/image1.pif f'', password(s) XXX"
[...]
Detected bots:
IP2 - server IP3 usr 2K-8006 nick [P00|DEU|59228] last 14:21:59
IP4 - server IP5 usr XP-3883 nick [P00|DEU|88820] last 19:28:12
[...]
```

Figure 4: Excerpt of the set of detected IRC bots and bot-servers (anonymized/edited for clarity).

Of the detected bots at MWN, only five used static IP addresses, while the rest used IP addresses from a NAT'd address range, indicating that most of them are private, student-owned machines. It is very time-consuming for the MWN operators to track down NAT'd IP addresses to their actual source. Worse, the experience at MWN is that even if they do, many of the owners turn out to not have the skills to remove the bot. Yet, it is important that such boxes cannot access the Internet.

The MWN operators accomplish this with the help of our system. They installed a blocking system for MWN's NAT subnets to which we interface with our system. The operators have found the system's soundness sufficiently reliable for flagging bots that they enabled it to block all reported bots *automatically*. They run this setup operationally, and so far without reporting to us any complaints. In the beginning, just after our system went online, the average number of blocked hosts increased by 10-20 addresses. After about two weeks of operation, the number of blocked hosts has almost settled back to the previous level, indicating that the system is effective: the number of bots has been significantly reduced.

Finally, we note that our detection scheme relies on the fact that a bot uses the IRC protocol in a fashion which conforms to the standard IRC specification. If the bot uses a custom protocol dialect, the IRC analyzer might not be able to parse the payload. This is a fundamental problem similar to the one we face if a bot uses a proprietary protocol. More generally we observe that the setting of seeing malicious clients *and* servers violates an important assumption of many network intrusion detection systems: an attacker does not control both endpoints of a connection [31]. If he does, any analysis is at best a heuristic.

6 Evaluation

We finish with an assessment of the performance impact of the `PIA` architecture and a look at the efficacy of the

multi-step protocol recognition/verification process. The evaluation confirms that our implementation of the PIA framework does not impose an undue performance overhead.

6.1 CPU Performance

To understand the performance impact of the PIA extensions to Bro, we conduct CPU measurements for both the unmodified Bro (developer version 1.1.52), referred to as Stock-Bro, and PIA-Bro running on the first 66 GB of the mwn-full-packets trace which corresponds to 29 minutes of traffic. (This trace again excludes the domain leo.org.) The trace environment consists of a dual-Opteron system with 2GB RAM, running FreeBSD 6.0.

In addition to the processing of the (default) Stock-Bro configuration, PIA-Bro must also perform four types of additional work: (i) examining *all* packets; (ii) performing *signature matches* for many packets; and (iii) buffering and reassembling the beginnings of all streams to enable reliable protocol detection; (iv) performing application-layer protocol analysis on additionally identified connections. In total, these constitute the cost we must pay for PIA-Bro's additional detection capabilities.

To measure the cost of each additional analysis element, we enable them one by one, as reported in Table 4. We begin with basic analysis (Config-A): Bro's generation of one-line connection summaries, as well as application-layer protocol analysis for FTP, HTTP, IRC, SMTP connections, as identified via port numbers. The first line reports CPU times for both versions of Bro performing this regular analysis, and the second line when we also enable Bro signatures corresponding to those used by PIA-Bro for protocol detection. We find the runtimes of the two systems quite similar, indicating that our implementation of the PIA architecture does not add overhead to Bro's existing processing. (The runtime of PIA-Bro is slightly less than Stock-Bro due to minor differences in their TCP bytestream reassembly process; this also leads PIA-Bro to make slightly fewer calls to Bro's signature engine for the results reported below. The runtimes of both systems exceed the duration of the trace, indicating that we use a configuration which, in this environment, requires multiple NIDS instances in live operation.)

With Config-A, the systems only need to process a subset of all packets: those using the well-known ports of the four protocols, plus any with TCP SYN/FIN/RST control packets (which Bro uses to generate generic TCP connection summaries). Bro uses a BPF [27] filter to discard any other packets. However, PIA-Bro cannot use this filtering because by its nature it needs to examine *all* packets. This imposes a significant performance penalty, which we assess in two different ways.

First, we prefilter the trace to a version containing only those packets matched by Bro's BPF filter, which in this case results in a trace just under 60% the size of the original. Running on this trace rather than the original approximates the benefit Bro obtains when executing on systems that use in-kernel BPF filtering for which captured packets must be copied to user space but discarded packets do not require a copy. The Table shows these timings as Config-A'. We see that, for this environment and configuration, this cost for using PIA is minor, about 3.5%.

Second, we manually change the filters of both systems to include all TCP packets (Config-B). The user time increases by a fairly modest 7.5% for Stock-Bro and 7.4% for PIA-Bro compared to Config-B. Note that here we are not yet enabling PIA-Bro's additional functionality, but are only assessing the cost to Bro of processing the entire packet stream using the above configuration; this entails little extra work for Bro since it does not perform application analysis on the additional packets.

PIA-Bro's use of signature matching also imposes overhead. While most major NIDSs rely on signature matching, the Bro system's default configuration does not. Accordingly, applying the PIA signatures to the packet stream adds to the system's load. To measure its cost, we added signature matching to the systems' configuration (second line of the table, as discussed above). The increase compared to Config-A is about 15–16%.

When we activate signatures for Config-B, we obtain Config-C, which now enables essentially the full PIA functionality. This increases runtime by 24–27% for Stock-Bro and PIA-Bro, respectively. Note that by the comparison with Stock-Bro running equivalent signatures, we see that capturing the entire packet stream and running signatures against it account for virtually all of the additional overhead PIA-Bro incurs.

As the quality of the signature matches improves when PIA-Bro has access to the reassembled payload of the connections, we further consider a configuration based on Config-C that also reassembles the data which the central manager buffers. This configuration only applies to PIA-Bro, for which it imposes a performance penalty of 1.2%. The penalty is so small because most packets arrive in order [9], and we only reassemble the first 4KB (the PIA buffer size).

As we can detect most protocols within the first KBs (see §4.2), we also evaluated a version of PIA-Bro that restricts signature matching to only the first 4KB. This optimization, which we annotate as PIA-Bro-M4K, yields a performance gain of 16.2%. Finally, adding reassembly has again only a small penalty (2.1%).

		Stock-Bro	PIA-Bro	PIA-Bro-M4K
Config-A	Standard	3335	3254	—
	Standard + sigs	3843	3778	—
Config-A'	Standard	3213	3142	—
Config-B	All TCP pkts	3584	3496	—
Config-C	All TCP pkts + sigs	4446	4436	3716
	All TCP pkts + sigs + reass.	—	4488	3795

Table 4: CPU user times on subset of the trace (secs; averaged over 3 runs each; standard deviation always < 13s).

In summary, for this configuration we can obtain nearly the full power of the PIA architecture (examining all packets, reassembling and matching on the first 4KB) at a performance cost of about 13.8% compared to Stock-Bro. While this is noticeable, we argue that the additional detection power provided merits the expenditure. We also note that the largest performance impact stems from applying signature matching to a large number of packets, for which we could envision leveraging specialized hardware to speed up. Finally, because we perform dynamic protocol analysis on a per-connection basis, the approach lends itself well to front-end load-balancing.

6.2 Detection Performance

We finish with a look at the efficacy of the PIA architecture's multi-step analysis process. To do so, we ran PIA-Bro with all adapted analyzers (HTTP, FTP, IRC, SMTP) on the 24-hour mwn-full-packets trace, relying only our bidirectional PIA-signatures for protocol detection, i.e., no port based identification. (Note that as these signatures differ from the L7-signatures used in §3, the results are not directly comparable.) PIA-Bro verifies the detection as discussed in §4.3, i.e., when the connection has either run for 30 seconds or transferred 4 KB of data (or terminated).

Our goal is to understand the quality of its detection in terms of false positives and false negatives. In trading off these two, we particularly wish to minimize false positives, as our experience related in §5 indicates that network operators strongly desire actionable information when reporting suspected bot hosts or surreptitious servers.

Table 5 breaks down PIA-Bro's detections as follows. The first column shows how often (i) a protocol detection signature flagged the given protocol as running on a non-standard port, for which (ii) the corresponding analyzer verified the detection. With strong likelihood, these detections reflect actionable information.

The second and third columns list how often the analyzer did *not* agree with the detection, but instead rejected the connection as exhibiting the given protocol, for non-standard and standard ports, respectively. Thus, the second column highlights the role the analyzer plays in reducing false positives; had we simply employed signatures without subsequent verification, then in these cases we would have derived erroneous information.

	Detected and verified non-std. port	Rejected by analyzer non std. port	Rejected by analyzer std. port
HTTP	1,283,132	21,153	146,202
FTP	14,488	180	1,792
IRC	1,421	91	3
SMTP	69	0	1,368

Table 5: # of connections with detection and verification.

The third column, on the other hand, raises questions regarding to what degree our protocol detection might be missing instances of given protocols. While we have not yet systematically assessed these rejections, those we have manually inspected have generally revealed either a significant protocol failure, or indeed an application other than that associated with the standard port. Examples of the former include errors in HTTP headers, non-numeric status codes in FTP responses, mismatches in SMTP dialogs between requests and responses, use of SMTP reply codes beyond the assigned range, and extremely short or mismatched IRC replies.

While we detect a large number of verified connections on non-standard ports—with the huge number of HTTP connections primarily due to various P2P applications—for this trace the only instance we found of a different protocol running on a privileged standard port was a (benign) IRC connection running on 80/tcp. On the unprivileged ports used for IRC, however, we found a private Apache HTTP server, a number of video-on-demand servers, and three FTP servers used for (likely illicit) music-sharing. (Note that, different than in §3.2, when looking for protocols running on standard ports, we can only detect instances of FTP, HTTP, IRC, and SMTP; also, protocols running *on top* of HTTP on port 80 are not reported.)

Finally, Figure 5 shows the diversity of the non-standard ports used by different types of servers. The

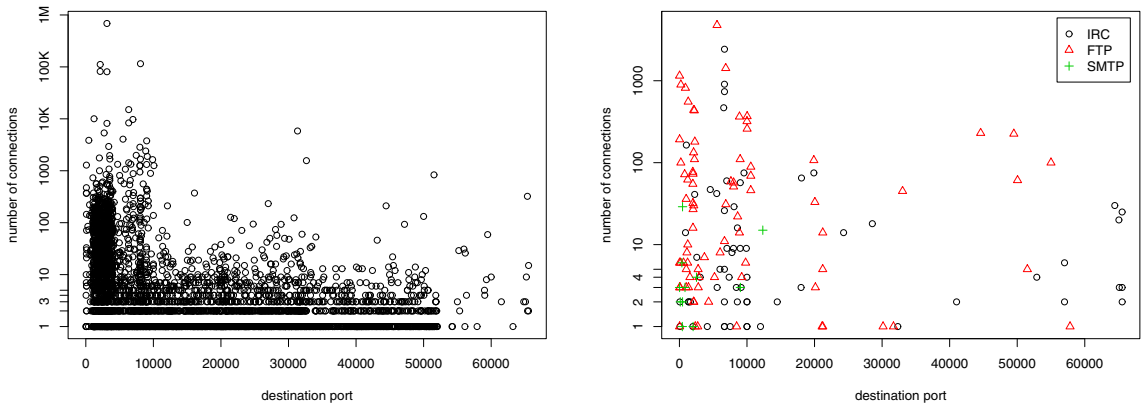

Figure 5: Connections using the HTTP (left) and the IRC, FTP, SMTP (right) protocol.

x-axis gives the port number used and the y-axis the number of connections whose servers resided on that port (log-scaled). The 22,647 HTTP servers we detected used 4,024 different non-standard ports, some involving more than 100,000 connections. We checked the top ten HTTP ports (which account for 88% of the connections) and found that most are due to a number of PlanetLab hosts (ports 312X, 212X), but also quite a large number are due to P2P applications, with Gnutella (port 6346) contributing the largest number of distinct servers. Similar observations, but in smaller numbers, hold for IRC, FTP, and SMTP, for which we observed 60, 81, and 11 different non-standard server ports, respectively. These variations, together with the security violations we discussed in §5, highlight the need for dynamic protocol detection.

7 Conclusion

In this paper we have developed a general NIDS framework which overcomes the limitations of traditional, port-based protocol analysis. The need for this capability arises because in today's networks an increasing share of the traffic resists correct classification using TCP/UDP port numbers. For a NIDS, such traffic is particularly interesting, as a common reason to avoid well-known ports is to evade security monitoring and policy enforcement. Still, today's NIDSs rely exclusively on ports to decide which higher-level protocol analysis to perform.

Our framework introduces a dynamic processing path that adds and removes analysis components as required. The scheme uses protocol detection mechanisms as triggers to activate analyzers, which can subsequently decline to process the connection if they determine the trigger was in error. The design of the framework is independent of any particular detection scheme and allows for the addition/removal of analyzers at arbitrary times. The design provides a high degree of modularity, which allows analyzers to work in parallel (e.g., to perform independent analyses of the same data), or in series (e.g., to decapsulate tunnels).

We implemented our design within the open-source Bro NIDS. We adapted several of the system's key components to leverage the new framework, including the protocol analyzers for HTTP, IRC, FTP, and SMTP, as well as leveraging Bro's signature engine as an efficient means for performing the initial protocol detection that is then verified by Bro's analyzers.

Prototypes of our extended Bro system currently run at the borders of three large-scale operational networks. Our example applications—reliable recognition of uses of non-standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers—have already exposed a significant number of security incidents at these sites. Due to its success, the MWN site has integrated our bot-detection into dynamic blocking of production traffic.

In the near future, we will migrate the remainder of Bro's analyzers to the new framework. From our experiences to date, it appears clear that using dynamic protocol analysis operationally will significantly increase the number of security breaches we can detect.

8 Acknowledgments

We would like to thank the Lawrence Berkeley National Laboratory; the Leibniz-Rechenzentrum, München; and the University of California, Berkeley. We would also like to thank Deti Fliegl, John Ives, Jason Lee, and Brian Tierney. This work was supported by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD), by the US National Science Foundation under grants ITR/ANI-0205519 and NSF-0433702, and by a grant from the Bavaria California Technology Center, for which we are grateful.

References

[1] T. H. P. . R. Alliance. Know your enemy: Tracking botnets. http://www.honeynet.org/papers/bots, 2005.

[2] S. A. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *Proc. IEEE Infocom 2006*, 2006.

[3] CERT Advisory CA-2003-20 W32/Blaster worm.

[4] BleedingSnort. http://bleedingsnort.com.

[5] BitTorrent. http://www.bittorrent.com.

[6] T. Choi, C. Kim, S. Yoon, J. Park, B. Lee, H. Kim, H. Chung, and T. Jeong. Content-aware Internet Application Traffic Measurement and Analysis. In *Proc. Network Operations and Management Symposium*, 2004.

[7] Clam AntiVirus. http://www.clamav.net.

[8] C. Dewes, A. Wichmann, and A. Feldmann. An Analysis Of Internet Chat Systems. In *Proc. ACM Internet Measurement Conference*, 2003.

[9] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly In the Presence of Adversaries. In *Proc. USENIX Security Symposium*, 2005.

[10] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *Proceedings of ACM CCS*, 2004.

[11] DSniff. www.monkey.org/~dugsong/dsniff.

[12] J. Early, C. Brodley, and C. Rosenberg. Behavioral Authentication of Server Flows. In *Proc. Annual Computer Security Applications Conference*, 2003.

[13] ENDACE Measurement Systems. http://www.endace.com.

[14] Enterasys Networks, Inc. Enterasys Dragon. http://www.enterasys.com/products/ids.

[15] Freechal P2P. http://www.freechal.com.

[16] D. Fliegl, T. Baur, and H. Reiser. Nat-O-Mat: Ein generisches Intrusion Prevention System. In *Proc. 20. DFN-Arbeitstagung über Kommunikationsnetze*, 2006.

[17] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. In *Proc. ACM Workshop on Mining Network Data*, 2005.

[18] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. 10th USENIX Security Symposium*, 2001.

[19] The Internet Corporation for Assigned Names and Numbers. http://www.iana.org.

[20] McAfee IntruShield Network IPS Appliances. http://www.networkassociates.com.

[21] The IPP2P project. http://www.ipp2p.org/.

[22] C. Kalt. Internet Relay Chat: Client Protocol. RFC 2812, 2000.

[23] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. ACM SIGCOMM*, 2005.

[24] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net.

[25] libmagic — Magic Number Recognition Library.

[26] M. Mai. Dynamic Protocol Analysis for Network Intrusion Detection Systems. Master's thesis, TU München, 2005.

[27] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[28] A. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Proc. Passive and Active Measurement Workshop*, 2005.

[29] A. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. ACM SIGMETRICS*, 2005.

[30] Linux NetFilter. http://www.netfilter.org.

[31] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[32] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–224, June 1995.

[33] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[34] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[35] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class of Service Mapping for QoS: A Statistical Signature Based Approach To IP Traffic Classification. In *Proc. ACM Internet Measurement Conference*, 2004.

[36] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proc. World Wide Web Conference*, 2004.

[37] R. Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, TU München, 2005.

[38] R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[39] K. Tan and B. Collie. Detection and classification of TCP/IP network services. In *Proc. Annual Computer Security Applications Conference*, 1997.

[40] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *Proc. ACM SIGCOMM*, 2005.

[41] Y. Zhang and V. Paxson. Detecting Backdoors. In *Proc. USENIX Security Symposium*, 2000.