# Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context


Holger Dreger
TU München
Computer Science Department
dreger@in.tum.de

Christian Kreibich
University of Cambridge
Computer Laboratory
christian.kreibich@cl.cam.ac.uk

Vern Paxson
International Computer Science Institute and
Lawrence Berkeley National Laboratory
vern@icir.org

Robin Sommer
TU München
Computer Science Department
sommer@in.tum.de



## Abstract

In the recent past, both network- and host-based approaches to intrusion detection have received much attention in the network security community. No approach, taken exclusively, provides a satisfactory solution: network-based systems are prone to evasion, while host-based solutions suffer from scalability and maintenance problems. In this paper we present an integrated approach, leveraging the best of both worlds: we preserve the advantages of network-based detection, but alleviate its weaknesses by improving the accuracy of the traffic analysis with specific host-based context. Our framework preserves a separation of policy from mechanism, is highly configurable and more flexible than sensor/manager-based architectures, and imposes a low overhead on the involved end hosts. We include a case study of our approach for a notoriously hard problem for purely network-based systems: the correct processing of HTTP requests.



**Keywords:** network security, distributed intrusion detection, host-supplied context, evasion


## 1 Introduction

In recent years, intrusion detection systems (IDSs) have become a central component in the tool chest of security analysts. Assuming proper maintenance and attention, IDSs provide essential information for the investigation of user activity, both in real-time and for post-incident forensics. Traditionally, one dimension along which IDSs have been classified is their *vantage point*: network-based systems (NIDSs) benefit from their wide field of vision, but suffer from both ambiguity in their observations [1] and challenging performance requirements. Host-based systems (HIDSs) solve the ambiguity problem, but often impose a significant performance overhead on executing processes and monitor individual hosts only. A number of solutions have been proposed to improve the accuracy of the network-based analysis process and to reduce the ambiguity problem [2, 3]. Furthermore, a number of distributed approaches have been proposed for improving the coverage of activity throughout the network (e.g., [4, 5, 6]). However, widespread adoption of such systems has not occurred. Despite well-known shortcomings, most systems deployed today still operate in a network-based and centralized fashion. The reasons are manifold and include ease of maintenance of a single device, potentially high coverage from a single point of view, and ease of deployment.

In this paper, we acknowledge this situation and present an architecture based on the Bro IDS [7] that remains faithful to its primarily network-based approach, while improving its accuracy by providing *host-based context* where it matters most in the analysis process. Our architecture allows for a *gradual transition* toward more distributed detection. We improve Bro's field of vision by augmenting its *mechanism* without sacrificing flexibility at the *policy* level: we integrate host-based components by allowing them to send and receive Bro events, the building blocks of the analysis policy in Bro deployments. We focus our attention on crucial and frequently exploited services that typically run on only a handful of machines. Compared to the usual host-based paradigm of performing all analysis on the end host itself, our solution incurs very modest performance and maintenance overhead on the end hosts because the actual analysis work is performed not by it but on a different system. From the perspective of the NIDS, our approach trades off an additional burden of communicating with the end systems for potentially saving a considerable number of cycles in the analysis process by obviating the need for costly NIDS processing to resolve ambiguity. A key question for the approach is to what degree this tradeoff of increased communication for decreased processing is a net gain. As we will show, this is indeed generally a significant win.

We note that the idea of leveraging host-based context in network-based IDSs is not itself novel [8, 9]. The contributions of our work are twofold: first, we move the idea forward by tightly integrating it with the well-established policy-driven approach of the Bro system. Second, we identify novel ways of leveraging the context provided by similar processing stages in the NIDS and host-based applications. In a detailed case study, we instrument the Apache web server with an interface to Bro. To demonstrate the feasibility of the architecture, we deploy such a setup in two production environments. Additionally, we examine the effectiveness of our multi-point analysis approach in a testbed by launching a large number of scripted attacks against the web server.

In the remainder of this paper we first recapitulate Bro's architecture in in Section 2, including an overview of the recent addition of a communication framework to the system. We then discuss the benefits of including host-supplied context in Section 3. In Section 4 we conduct a case study: we instrument the Apache web server to supply information to concurrently executing Bros. Section 5 presents our experiences with instrumented Apaches in a test-lab installation as well as in two productional environments. We summarize the paper and point out future work in Section 6.

# 2 Bro: A Distributed Event-Based Intrusion Detection System

Bro's architecture has remained faithful to the original philosophy developed in the original paper [7]. A significant recent improvement has been the introduction of a communications framework as the basis of a more powerful event model suitable for distributed event communication [10, 11]. We summarize the architecture's key elements here in condensed form to put in context our integration of host-supplied context. Figure 1 illustrates Bro's architecture.

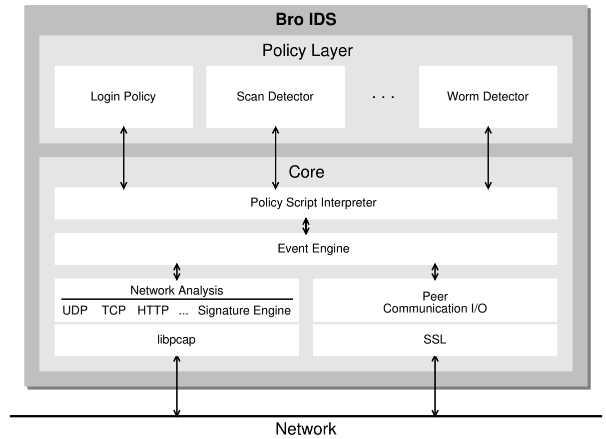


Figure 1: Architecture of the Bro IDS.

## 2.1 Separation of Mechanism from Policy

A core idea of Bro is to split event detection mechanisms from event processing policies. Event generation is performed by *analyzers* in Bro's core: these analyzers operate continuously based on input observed by Bro instances and trigger events asynchronously when corresponding activity is observed. Bro's core contains analyzers for a wide range of network protocols such as RPC, FTP, HTTP, ICMP, SMTP, TCP, UDP, and others. These analyzers are *connection-oriented*: they associate state with connections observed on the network and trigger events whenever interesting protocol activity is encountered.[1] Examples include the establishment of a new TCP connection or an HTTP request. Bro also provides a *signature engine* for typical misuse-based intrusion detection: it matches byte string signatures against traffic flows and triggers events whenever a signature matches [12]. Once an event is triggered, the engine passes it to the *policy layer*, which then takes care of processing the event, possibly triggering new ones. The design takes care to minimize CPU load: only analyzers responsible for triggering the events used at the policy layer are actually enabled.

## 2.2 Policy Configuration

Each Bro peer runs a policy configuration in its policy layer. This policy embodies the site's security policy, expressed in scripts containing statements in the special-purpose Bro scripting language. To understand the significance of this approach it is important to realize that

[1] Bro's concept of a connection is protocol-dependent; for connectionless protocols such as UDP a connection, it is defined as a bidirectional flow that shares the same endpoint addresses and ports and is terminated upon an inactivity timeout.

the relevance of an event varies from site to site. A very simple example is that some sites may consider the detection of a Microsoft IIS exploit attempt on a pure UNIX network a threat, while others may not; much more detailed, subtle, and contextual policy distinctions are not only supported but often seen in operational use. Bro's policy language is strongly typed, procedural in style, and provides a wide range of elementary data types to facilitate the analysis of activity on a network.

### 2.3 Communication Framework & State Management

Bro's communication framework supports the serialization and transmission of arbitrary kinds of state between Bro instances. The driving idea behind its design is to allow the realization of *independent* state [10]: that is, we should no longer think of state accumulated at the policy layer as a local concept, but rather as information dispersed throughout the network, and potentially shared between past and future executions of Bro. The communication model imposes no hierarchical structure. Examples of exchangeable state include triggered events; state kept in data structures managed by policies; and the policy definitions themselves. For the purpose of this paper it is sufficient to think of the entities exchanged between peers as events, though that ignores a large part of its flexibility.

To interface other applications to Bro, we have implemented a lightweight, highly portable library supporting Bro's communication protocol called *Broccoli*[2] that allows nodes that are not instances of the Bro IDS to partake in its event communication [13]. Broccoli nodes can request, send, and receive Bro events just like Bro itself, but cannot be configured using Bro's policy language. A Broccoli node's policy has to be implemented directly in the client's code, or through mechanisms such as configuration files.

## 3 Using Host-Supplied Context in Network Intrusion Detection

Having a distributed Network Intrusion Detection System at hand, we can use the NIDS's communication mechanisms to implement host-based sensors to supplement the NIDS's analyses. In this section we explain how a NIDS can benefit from this additional information and how we integrated host-supplied context into Bro's event-based framework.

[2] *Broccoli* is the healthy acronym for "Bro Client Communications Library."

### 3.1 Motivation

Our motivation for augmenting network-based analysis with host-supplied context is fivefold:

1. OVERCOMING ENCRYPTION. One major benefit of host-supplied context is that the host has access to information before and after any flow encryption takes place. The recipient of an encrypted connection can be instrumented to report selected information to the NIDS, such as user login names or requested objects. Thus, instrumenting server applications that employ encrypted communication allows us to do the same protocol analysis as for clear-text protocols.

2. COMPREHENSIVE PROTOCOL ANALYSIS. Having host applications report to the NIDS enables us to access additional information about the applications' internal protocol state. As endpoints fully decode the application-layer protocol in any case, they can easily provide the NIDS with context that for the NIDS is hard to derive itself.

   A simple example is user authentication during a Telnet login session. The Telnet protocol does not include any information about login success or failure, so Bro must resort to heuristics in an attempt to infer the result of an authentication attempt based on the keystroke/response dialog [7]. But the Telnet server end host immediately and unambiguously knows the outcome of such attempts.

3. ANTI-EVASION. Evasion attacks are one of the most fundamental problems of network intrusion detection. They exploit ambiguities inherently present in observing network traffic from a location other than one of the endpoints. These ambiguities render it hard, or even impossible, for a NIDS to correctly interpret skillfully crafted packet sequences in the same fashion as the end host receiving them. Such attacks can exploit differing interpretations of traffic at multiple protocol levels. From the application layer's point of view, it is generally not possible to pinpoint the exact location in the protocol stack where the ambiguity was introduced: for a web server, it might have been within HTTP itself, but could just as well have occurred due to TCP retransmissions (layer 4) or IP fragmentation (layer 3). In a seminal paper [1], Ptacek and Newsham describe several network- and transport-layer attacks that lead to different payload streams perceived by the end-system and the NIDS. Approaches that alleviate the problem exist (e.g., normalization [2] and active mapping [3]), but have not

seen deployment in large-scale networks yet.

The NIDS's analysis can likewise leverage host-based context at multiple levels. One way to use this is for learning how the application interprets the received data, i.e., we can use additional information to detect evasion attacks against the NIDS. By including application-layer state of the host into the analysis, such attacks can be detected and/or avoided. Another interesting approach is the instrumentation of a host's network stack, which would allow it to share information about its stream reassembly with the NIDS. A key question here is how to minimize the amount of information that needs to be shared to allow such a comparison. For example, we can envision exchanging checksums of the stream to detect mismatches in a lightweight fashion. Such instrumentation would allow us to monitor *multiple types* of applications for evasion attacks without the need to instrument each application individually.

4. ADAPTIVE SCRUTINY. Generally, there is a wealth of things that can cause an IDS to become suspicious about a connection's intent: unusual destination hosts or ports, scanning behavior by the source host in the past, matches to traffic flow signatures, or a large number of IP fragments are just a small set of examples. Our approach adds another indicator to the toolbox: deviation of the interpretations on the end host and the NIDS can also be used to classify a connection as more suspicious than others, initiating closer scrutiny of such traffic.

5. IDS HARDENING. Lastly, differing interpretations of the same data might simply point out subtle bugs in the implementation of the NIDS, or even in the application itself.

More generally, we see that there are two – somewhat complementary – approaches to leveraging host-supplied context. First, the host can provide *additional* context for the NIDS to include into analysis. Second, the host can supply *redundant* context which the NIDS uses to verify information is has distilled itself.

### 3.2 Integration into Bro

We incorporate host-supplied context into Bro's analysis by letting selected applications send events to a central Bro instance. Similar to Bro's core-generated events, remote events still represent policy neutral descriptions of phenomena occurring within individual process executions. This implies that the policy that determines the relevance of these events is exclusively maintained on the Bro host. The benefits of maintaining the policy here, rather than pushed out to the end hosts, are twofold: first, the policy is accessible centrally and thus easier to adapt; second, this approach imposes less overhead on the monitored host than ordinary HIDSs since the data is not analyzed on the host itself. Generating and sending an event does not cost the host much more effort than writing to a log file. In addition, we can instrument a host process with fairly little effort using the Broccoli library. Since Broccoli implements bidirectional event communication, an instrumented application can also be made controllable by Bro in order to react in accordance to the policy.

Bro's connection-oriented view of traffic analysis raises significant issues for the integration of remote events with existing local state. Essentially, we need to unite the stream of events generated by observing a connection on the wire with the stream of events generated by the remote application that processes the connection's data. One avenue for doing so is to have the remote application send along the parameters identifying the connection, for example the IP/port quadruple. In order for this to work, the analyzer must be structured in a way to allow this fusion of event streams. This means that we must make available all state required to process the events to all relevant event handlers. Furthermore, this state must be structured to support the processing of events of different origins and levels of abstraction levels. One instance of this problem space is the need for synchronization when we cannot guarantee that the Bro host can monitor all relevant traffic: we must ensure that new state can be instantiated by both local and remote events, and that this state is not expired prematurely.

## 4 Analysis of HTTP Sessions

For our case study, we decided to take a closer look at HTTP, the most widely used application layer protocol in the Internet. All major NIDSs provide components to detect HTTP-based attacks, which at a minimum extract the requested URLs from the network stream and match these against a set of signatures to detect malicious requests.

The main observation here is that there are at least two HTTP decoders which dissect the same HTTP connection, namely the web server and the NIDS. While this is a duplication of work, the separation of the tasks is indeed reasonable: per our discussion above, we prefer the web server not to perform the intrusion detection itself (and, naturally, it does not make sense for the NIDS to serve HTTP requests). However, this redundancy allows us to benefit from both additional and redundant context, as discussed in Section 3.1. We will now discuss both approaches in turn. While we will focus on URLs extracted from the requests, we note that similar reasoning holds for deeper inspection.

### 4.1 Leveraging Additional Web Server Context

With respect to the semantics of a given HTTP request, it is obviously the web server that is authoritative: its environment-specific configuration defines the interpretation of the request and the meaning of any reply. Thus, providing the NIDS with information from the web server promises to offer a significant increase in contextual information.

Web servers can provide several kinds of context that are hard or impossible for the NIDS to derive by itself:

- **Decryption:** SSL-enabled sessions have become quite common for transferring sensitive data. While quite desirable, this poses severe restrictions on passive network monitoring. However, since the web server decrypts such requests, it can provide them as clear-text to the NIDS via an independent (and again encrypted) channel.

- **Full request processing:** The web server always fully decodes the request stream it receives. In contrast, many NIDSs perform this task rather half-heartedly; e.g., Snort [14] may miss requests in pipelined/persistent connections if they cross packet boundaries (older versions used to extract only the very first URL from each packet).

- **Full reply processing:** Some information can be easily provided by the web server while a NIDS needs to put considerable effort into deriving it. For example, Bro is able to extract the server's reply code from HTTP sessions. But, to our experience in several high-performance environments, this comes at a prohibitive processing cost. On the other hand, for the web server there is no additional cost involved in providing the result, other than that of sending the data to the NIDS.

- **Disambiguation:** The document eventually served can substantially differ from the one requested. The server resolves the path inside a URL in a virtual namespace; without further context it may not be predictable which *file* is given in response. Redirection and rewriting mechanisms internal to the server can change the URL path arbitrarily. For a NIDS to follow the exact same steps as the web server, it would need to know all related configuration statements as well as the full file system layout of the web server — infeasible in practical terms. Furthermore, most NIDSs are simply not flexible enough to accommodate such a "shadow configuration".

### 4.2 Avoiding Evasion using Redundant Context

Evasion attacks can be used to mislead the NIDS's HTTP protocol decoding. If the NIDS extracts a different HTTP request than the web server — or if it does not see one at all — it may produce both false negatives and false positives. However, if we can compare the outcome of the two HTTP decoders, we have an opportunity to detect these mismatches.

For a web session, network- and transport layers evasion attacks [1] can be used to hide, alter, or inject URLs. Moreover, there are ways to evade the application-layer HTTP decoders of NIDSs. The most prevalent form is *URL encoding* [15]. Per RFC 2396 [16], URLs may only contain a subset of the US-ASCII characters. However, to represent other characters, arbitrary values can be encoded using special control sequences. For example, web servers are required to support the "percent-encoding" which can encode arbitrary hexadecimal values. Some web servers — most notably Microsoft's IIS — also provide more sophisticated encodings, such as Unicode [17].

For a NIDS, it is hard to precisely mimic these encodings and character sets. In the past, many systems required fixes upon the discovery of new encoding tricks (e.g., [18]). In general, a web server's eventual interpretation of an URL depends on its local environment and configuration, making it nearly impossible for a NIDS to derive it. This issue is part of the more general problem of NIDSs often lacking context required to reliably detect attacks [12].

Often, such application-layer encoding attacks target not the NIDS but the web server itself. Due to implementation bugs, such an encoding may circumvent internal checks. For example, CVE entry 2001-0333 [19] discusses a flaw in the ISS server which leads to a filename being decoded twice. We can detect such bugs if we compare the decoding the web server performs with the independent result of the NIDS. Similarly, the NIDS might have flaws that show up when verified with the outcome of the web server.

Finally, while comparing the output of the two decoders can detect both evasion attacks and implementation flaws, we must also prepare ourselves for the possibility of numerous *benign* differences, which we explore further below.

## 5 Deployment and Results

For our case study, we have evaluated our approach in three installations: an experimental test bed and two production environments. All use the Apache web server and the Bro NIDS.

### 5.1 Setup

We instrumented the Apache web server with a Broccoli client that communicates with an instance of the Bro NIDS running concurrently on either the same machine or a remote host. Semantically, the communication between Apache and Bro is one-way. For each request, Apache sends the involved hosts & TCP ports, the original request string, the URL as canonicalized by Apache, the name of the file being served, and the HTTP reply code. This information is available through Apache's default logging module (except we need a slight extension to access the ports).

There are two different ways of connecting the server with Broccoli. The first, which is particularly unobtrusive, is using a separate process for the Broccoli client, which either reads the Apache log file (so no modification to Apache at all) or communicates with Apache via a pipe. The second is to integrate use of Broccoli directly into Apache. We implemented both of these. We used the first for our operational deployments, and the second for our performance testing (detailed below).

When Bro receives an Apache request, it runs two kinds of analysis, corresponding to the two main uses identified in Sections 4.1 and 4.2. First, it passes the canonicalized URL through its standard detection process. This includes both script-layer analysis and event-layer signature matching. Second, it matches the URL against the one extracted by Bro itself from the connection's packet stream. If it encounters a difference, it generates an alert.

In our test bed, we installed Apache 2.0.52 and a recent development version of Bro on the same host. We let Bro run its default HTTP analysis on the packet stream as seen on the loopback device. The Apache-supplied information was sent over a TCP connection from the Broccoli client to the Bro system.

We also instrumented two production web servers at Technische Universität München, Germany: the main web server of the the Computer Science Department, and the server of the Network Architectures Group. Both are connected to a backbone network with a Gb/s uplink to the Internet. The main server handles between 20.000 and 30.000 requests per day. To monitor it, we used the approach above of a separate Broccoli client reading from its log file. The Network Architecture Group's server processes about 5.000-6.000 requests a day. For it, we ran Bro on the same host and used a direct connection between Apache and the Broccoli client, like we did with the testbed.

### 5.2 Experiences

We operated these setups for two weeks, with very encouraging results. We first discuss how the additional context indeed provided significant benefits for the detection process, and then our preliminary experiences with evaluating redundant context to detect evasion attacks and decoding flaws. We also note that maintaining the analysis policy on the Bro side while keeping the Broccoli client policy-neutral proved valuable: we could change the configuration of the NIDS at will without needing to touch the web servers.

#### 5.2.1 Additional Context

Incorporating context supplied by Apache proved to be a major gain. First, we could confirm that the NIDS reliably saw all requests served by the web server — a major benefit, since in high-volume environments a NIDS running on commodity hardware regularly drops packets and therefore may miss accesses [20].

Next, we confirmed that Bro could perform signature matching on the URLs and filenames even if we omitted HTTP decoding from Bro's configuration. For high-volume web servers, this holds the potential to realize a major performance gain, since HTTP analysis can easily increase total CPU usage by a factor of 4–6 [20].

Bro's signature engine assumes internal connection state already exists when matching signatures for a given connection. But if Bro is not decoding the HTTP traffic directly, but rather only receiving it as a feed from Apache, it will not have instantiated this state. Fortunately, we can arrange for Bro to instantiate such state by having it capture only TCP control packets (SYNs, FINs and RSTs). In our experience, it is quite feasible to analyze *all* such control packets even in highly loaded Gb/s environments. Note, though, that this approach limits internal signature matching to HTTP sessions which Bro sees itself. Matching on requests from unseen connections (for example, those internal to the site) will require additional internal modifications, which we plan to implement soon. Also, we note that this restriction only applies to the internal signature engine. Script-level analysis, such as regular expression matching, is generally possible even without internal connection state.

Bro uses bidirectional signatures to avoid false positives. For example, many of the HTTP signatures only alert if the server does not respond with an error message. Since Apache supplies us with its reply code as well, we retain this important feature.

Finally, we now for the first time are able to detect attacks in SSL-encrypted sessions. We verified that Bro indeed received the decrypted information and spotted sensitive accesses within them.

#### 5.2.2 Redundant Context

We configured the Bro system to automatically compare the URLs received from the Apache server with those distilled by its own HTTP decoder. There are cases in which differences in the URLs are legitimate. Most importantly, the web server may internally expand the requested URL, for example when expanding a request like `/foo/bar/` into `/foo/bar/index.html`. However, from our preliminary experiences with the two production servers, it appears that in practice such differences may be rare enough explicitly code into the NIDS's configuration. Consequently, for Bro we implemented an expansion table of regular expressions that reproduces such URL translations.

Before we compare two URLs, we also strip CGI parameters. When logging a URL, Apache does not remove the URL-encoded parameters. Bro, on the other hand, decodes the parameters fully. Therefore, such stripping is required to avoid mismatches in accesses to CGI scripts.

This policy is running well on our production servers. The main source of differences we encountered were with requests of the form `GET http://www.foo.bar/index.html HTTP/1.x`. Such requests indicate that somebody is trying to use the web server as a proxy. Apache strips `http://www.foo.bar` before processing the request; Bro does not. Examining these requests more closely, we saw that they were mostly scans for open proxies. Others indicated client misconfigurations.

We found additional differences between Apache and Bro. None of these turned out to be security-relevant (e.g., we saw client requests which included labels of the form "foo.html#label"; these labels are removed by Apache). However, the question remains whether in a larger-scale environment such differences would occur often enough, and in sufficiently varied forms, to significantly complicate the use of redundant context for detecting evasion attempts and decoder flaws.

To stress both Apache and Bro more intensively, we installed three evasion tools in our test-lab. Libwhisker [21] is a Perl library which includes various URL encoding tricks supposed to evade NIDSs or the security mechanisms of a web server [22]. It includes a command-line script for issuing individual requests to a server. We patched this script to selectively enable one or more of the evasion methods. We also installed the penetration testing tool Nikto [23], which ships with a large library of HTTP requests to exploit known server vulnerabilities. Internally, Nikto leverages libwhisker. Therefore, it is able to encode its requests using libwhisker's evasion techniques. Finally, we used a small stand-alone encoder [24], which converts arbitrary strings into different Unicode representations.

The results of our evasion experiments are encouraging. Both systems, Apache and Bro, do a good job at decoding the crafted requests, with the following differences:

- Libwhisker can insert relative directory references into URLs, turning `/foo/bar/` into e.g. `/foo/./bar/` or `/garbage/../foo/bar/`. Apache canonicalizes the path. Bro leaves it untouched, which for a NIDS not knowing the web server's filesystem layout makes sense: subsequent analysis may want to alert on these references.

- To avoid ambiguities, double-encoded requests are never to be decoded more than once. (In a double encoding, a character such as 'z' — ASCII `0x7a` — is encoded as `%%37%41`. The first decoding step yields `%7a`, then the second gives '`z`'). If Apache encounters such a request, it logs the result of the first decoding step but sends an error to the client. Bro also decodes it only once, but removes the additional percentage sign before further processing. In addition, it reports the ambiguity. While their behaviors differ, both systems recognize the situation and report an error.

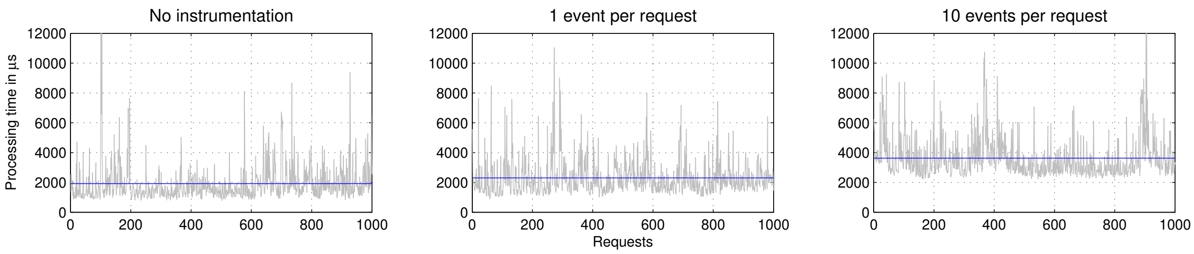


Figure 2: Overhead of Bro event transmission on service time for a sequence of 1000 requests to the same, static webpage. The left graph shows an unmodified Apache's operation, the middle one shows service times with a single event transmitted per request, the right one shows service times with 10 identical events transmitted per request. In each case, the horizontal line indicates the average value across all requests.

- Requests containing Unicode characters (literally, or encoded with the IIS-proprietary %u encoding) are either left untouched or treated as an error by Apache.[3] Bro always leaves such characters untouched. Thus, either the two systems agree, or Apache does not serve any document.

To summarize, we see that Apache and Bro appear to work well together in terms of HTTP URL-canonicalization. If in the future we encounter more mismatches, we can now detect them as soon as they occur. We note that our results may not readily apply to other web servers. For example, Microsoft's IIS supports a handful of other encodings [17] not supported by Bro. In particular, Bro does not include a Unicode decoder yet. In addition, past experience with IIS vulnerabilities suggests that its more complex decoder may also be more vulnerable than Apache's.

## 5.3 Performance Evaluation

A key question is whether the performance overhead of the instrumentation is tolerable. We tested the performance impact incurred on Apache using httperf [25] as a load generator. We ran each of httperf, Apache, and Bro on separate machines (2.53Ghz Pentium 4s with 500MB RAM) connected on a 100Mb/s network. For these measurements, we implemented the Broccoli client in the form of an Apache 1.3 logging module, `mod_bro`, requiring only an additional 120 lines of C code.

We first measured the per-request overhead of sending Bro events from a lightly loaded Apache. We requested a single, static webpage 1000 times at a rate of 20 connections per second, measuring the request processing times using the `mod_benchmark` module [26], and averaged the results of the $n$th request across 10 separate runs. The results are shown in Figure 2: on average, Apache required around 2ms for each request. Sending the single Bro event necessary for our contextual analysis had quite low performance impact, on the order of 300$\mu$s per request, so capable of supporting say 1000 requests/sec.

[3] This is true for Unix systems. On Windows, Apache may handle Unicode differently but we have not examined this further.

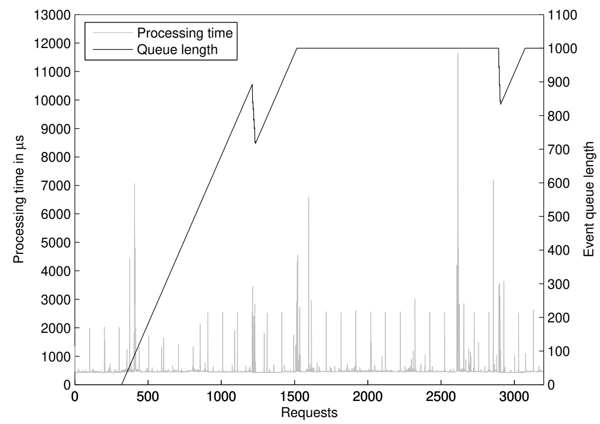


Figure 3: Overhead of event transmission when the collecting Bro is overloaded. The size of the event queue in the instrumented application has no noticeable impact on the application's performance.

The second experiment tested the overhead with a Bro under heavy load. To emulate this situation reliably, we artificially introduced a processing delay of 0.2s per received event on the Bro side. Broccoli clients have a bounded per-connection event queue that we configured to a maximum size of 1000 events. Additional events enqueued at this point lead to the oldest events being dropped. To simplify the queuing behavior, we ran Apache with a single process serving requests only.

The results are shown in Figure 3: the workload of the receiving Bro host does not noticeably affect the instrumented application's performance.

In our production installations we always connected a single web server to Bro. To explore how our setup might scale with more instrumented servers, we measured the amount of data exchanged between one instance of Apache and the receiving Bro. This volume depends on the number of HTTP requests as well as the length of the requested URLs, but is independent of the HTTP connection's actual payload size. A single run of Nikto (see Section 5.2.2) issues 2443 requests to the web server. On average, for every request 455 bytes of payload are transmitted between Apache and Bro.[4] Thus, the network load is modest: under 1 Mbps for 2000 requests/sec, a level that can accommodate a good number of busy web servers. For the Bro side, the amount of work to process the received bytes is, in general, much less than to parse the full HTTP stream. Therefore, one option here is to significantly lighten the load on Bro by leveraging the web server's processing and context, which should enable Bro's monitoring to scale to significantly higher HTTP loads than before.

To summarize, from our preliminary assessment the overhead imposed by instrumenting applications to participate in the event communication of a network of Bro nodes appears quite acceptable.

## 6 Summary & Future Work

In this paper we have developed the notion of the extensive enhancements possible by supplementing network-based intrusion detection with host-supplied context. By incorporating a host's authoritative state into the NIDS's analysis, we can provide the NIDS with both *additional* context and *redundant* context. These allow us to analyze encrypted traffic, leverage the host's protocol decoder, detect evasion attacks, increase scrutiny for suspicious hosts, and both offload and harden the NIDS itself.

As a case study we instrumented the Apache web server with an interface to the open-source Bro NIDS. We extended Bro to incorporate the web server accesses into its detection process. Additionally, Bro can compare the URLs provided by Apache with the URLs it distilled itself by passive HTTP protocol analysis, providing a means for detecting evasion attacks and flawed decoders (either the server's or its own).

We installed the Apache/Bro combo in two production environments and examined it in more detail in a test bed. The proof-of-principle results from these deployments are quite encouraging. A critical question to now explore concerns *scaling*: will the projections we obtained from our preliminary experiments indeed hold up when we deploy such instrumentation more widely within a site? In particular, the direct communication of redundant context *(i)* doubles the volume of data the NIDS processes, and *(ii)* may wind up generating many more benign differences in deployments where a wider diversity of server configurations comes into play. These problems may be amenable to refinements in the basic technique — for example, rather than transmitting the entire redundant context from the server to the NIDS, instead only sending an incremental checksum, greatly reducing the network volume in the common case of the streams agreeing; and finding additional canonicalizations to remove benign variations — but it will take broader operational experiences to properly explore these possibilities.

Another area ripe for future work concerns extending the approach to other host applications. In particular, we are working on an SSH server instrumented to report both the results of authentication attempts and the clear text inputs and outputs of login sessions. These then will allow us to leverage Bro's existing Rlogin and Telnet analyzers for the examination of encrypted user sessions, which operationally has proved increasingly critical with the now widespread use of SSH.


## Acknowledgments

This work is carried out in collaboration with Intel Research Cambridge. We would like to thank Jon Crowcroft and Anja Feldmann for helpful discussion and feedback. We would also like to thank Alexander Lüdtke for his help when setting up the test environment.


## References


[1] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[2] Mark Handley, Christian Kreibich, and Vern Paxson. Network Intrusion Detection: Evasion, Traffic


[4] Roughly two thirds of these bytes come from protocol overhead. While high, note that Bro's communication protocol can exchange serializations of Bro's complex data structures while ensuring type-safety, reconstructing reference structures, and performing architecture-independent data marshaling. We thus trade off efficiency for flexibility here.

Normalization, and End-to-End Protocol Semantics. In *Proc. 10th USENIX Security Symposium*, 2001.

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

[4] Phillip A. Porras and Peter G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore, MD, October 1997.

[5] Giovanni Vigna and Richard A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[6] Eugene H. Spafford and Diego Zamboni. Intrusion Detection Using Autonomous Agents. *Computer Networks*, 34(4):547–570, 2000.

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

[8] Magnus Almgren and Ulf Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[9] Marc Welz and Andrew Hutchison. Interfacing Trusted Applications with Intrusion Detection Systems. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[10] Robin Sommer and Vern Paxson. Exploiting Independent State For Network Intrusion Detection. Technical Report TUM-I0420, TU München, 2004.

[11] Christian Kreibich and Robin Sommer. Policy-controlled Event Management for Distributed Intrusion Detection. 2005. (Under submission).

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

[13] Broccoli: The Bro Client Communications Library. `http://www.cl.cam.ac.uk/~cpk25/broccoli/`.

[14] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. 13th Systems Administration Conference (LISA)*, pages 229–238, 1999.

[15] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison Wesley Professional, 2004.

[16] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax, 1998. RFC 2396.

[17] Daniel J. Roelker. HTTP IDS Evasions Revisited. `http://www.sourcefire.com/products/downloads/secured/sf_HTTP_IDS_evasio%ns.pdf`, 2004.

[18] Internet Security Systems Security Alert. Multiple Vendor IDS Unicode Bypass Vulnerability. `http://xforce.iss.net/xforce/alerts/id/advise95`, 2001.

[19] CVE-2001-0333. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0884`, 2001.

[20] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *Proc. 11th ACM Conference on Computer and Communications Security*, 2004.

[21] libwhisker. `http://www.wiretrip.net/rfp`.

[22] Rain Forrest Puppy. A Look At Whisker's Anti-IDS Tactics. `http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html`, 1999.

[23] Nikto. `http://www.cirt.net/code/nikto.shtml`.

[24] Daniel J. Roelker. URL encoder. `http://code.idsresearch.org/encoder.c`.

[25] D. Mosberger and T. Jin. httperf - A Tool For Measuring Web Server Performance. In *Proc. of the First Workshop on Internet Server Performance (WISP '98), Madison, WI*, pages 59–67, June 1998.

[26] mod_benchmark Apache plugin. `http://www.trickytools.com/php/mod_benchmark.php`.

[27] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proceedings of the 9th Usenix Security Symposium*, Aug 2000.

[28] D. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale Stepping-Stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2002.