Honeycomb – Creating Intrusion Detection Signatures Using Honeypots

Christian Kreibich, Jon Crowcroft
University of Cambridge Computer Laboratory
JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
{firstname.lastname}@cl.cam.ac.uk

***Abstract*— This paper describes a system for automated generation of attack signatures for network intrusion detection systems. Our system applies pattern-matching techniques and protocol conformance checks on multiple levels in the protocol hierarchy to network traffic captured a honeypot system. We present results of running the system on an unprotected cable modem connection for 24 hours. The system successfully created precise traffic signatures that otherwise would have required the skills and time of a security officer to inspect the traffic manually.**

***Index Terms*— network intrusion detection, traffic signatures, honeypots, pattern detection, protocol analysis, longest-common-substring algorithms, suffix trees.**

I. Introduction

CURRENT network intrusion detection systems (NIDSs) often work as *misuse detectors*, where the packets in the monitored network are compared against a repository of *signatures* that define characteristics of an intrusion. Successful matchings then fire alerts.

This work focuses on signature generation. At present, the creation of these signatures is a tedious, manual process that requires detailed knowledge of each software exploit that is supposed to be captured. Simplistic signatures tend to generate large numbers of false positives, too specific ones cause false negatives.

To address these issues, we present Honeycomb, a system that generates signatures for malicious network traffic automatically. Our system uses pattern-detection techniques and packet header conformance tests on traffic captured on honeypots. Looking only at traffic on a honeypot provides the major benefit of knowing that one is dealing with suspicious traffic, since the whole point of honeypots is to capture such activity (see Section II-B).

We have extended the open-source honeypot `honeyd` by a subsystem that inspects traffic inside the honeypot at different levels in the protocol hierarchy; currently we examine IP, TCP and UDP headers as well as payload data. Our tests indicate that this is a promising approach to automating the generation of intrusion detection signatures for unknown attacks.

The remainder of this paper is structured as follows: Section II reviews NIDS signatures, honeypot systems and pattern detection algorithms, Section III describes the Honeycomb architecture in detail and Section IV presents initial results. Finally, Section V discusses our approach, and Section VI summarizes the paper.

II. Background

A. Intrusion Detection Signatures

The purpose of attack signatures is to describe the characteristic elements of attacks. There is currently no common standard for defining these signatures. As a consequence, different systems provide signature languages of varying expressiveness.

Generally, a good signature must be *narrow* enough to capture precisely the characteristic aspects of exploit it attempts to address; at the same time, it should be *flexible* enough to capture variations of the attack. Failure in one way or the other leads to either large amounts of false positives or false negatives.

Our system supports signatures for the Bro[1] and Snort[2] NIDSs. Bro has a powerful signature language that allows the use of regular expressions, association of traffic going in both directions, and encoding of attacks that comprise multiple stages. Snort's signature language is currently not as expressive as Bro's. We include Snort here because of its current popularity and large signature repository.

B. Honeypots

Honeypots are decoy computer resources set up for the purpose of monitoring and logging the activities of entities that probe, attack or compromise them[3][4][5]. Activities on honeypots can be considiered suspicious by definition, as there is no point for benign users to interact with these systems. Honeypots come in many shapes and sizes; examples include dummy items in a database, low-interaction network components like preconfigured traffic sinks, or full-interaction hosts with real operating systems and services.

Our system is an extension of `honeyd` [6], a popular low-interaction open-source honeypot. `honeyd` simulates hosts with individual networking *personalities*. It intercepts traffic sent to nonexistant hosts and uses the simulated systems to respond to this traffic. Each host's personality can be individually configured in terms of OS type (as far as detectable by common fingerprinting tools) and running network services (termend *subsystems*).

C. String-based Pattern Detection Algorithms

Our system is unique in that it *generates* signatures. In contrast to NIDSs, it cannot read a database of signatures upon startup to match them against live traffic to spot matches. Thus, the commonly employed pattern-matching algorithms in NIDSs are of no use to us. Instead, the system tries to spot patterns in traffic previously seen on the honeypot: we overlay parts of flows in the traffic and use a longest common substring (LCS) algorithm to spot similarities in packet payloads. Like pattern matching, LCS algorithms have been thoroughly studied in the past. Our LCS implementation is based on suffix trees, which are used as building blocks for a variety of string algorithms. Using suffix trees, the longest common substring of two strings is straightforward to find in linear time [7]. Several algorithms have been proposed to build suffix trees in linear time [8][9]; our implementation uses Ukkonen's algorithm[10].

III. Honeycomb Architecture

The following sections explain individual aspects of our system in detail.

A. honeyd *Extension*

We have added two new concepts to honeyd: a plugin infrastructure and event callback hooks. The plugin infrastructure allowes us to write extensions that remain logically separated from the honeyd codebase[1], while the event hooks provide a mechanism to integrate the plugins into the activities inside the honeypot. Currently, hooks allow a plugin to be informed when packets are received and sent, when data is passed to and received from the subsystems and to receive updates about honeyd's connection state. Honeycomb is implemented as a honeyd plugin. Figure 1 illustrates the architecture.

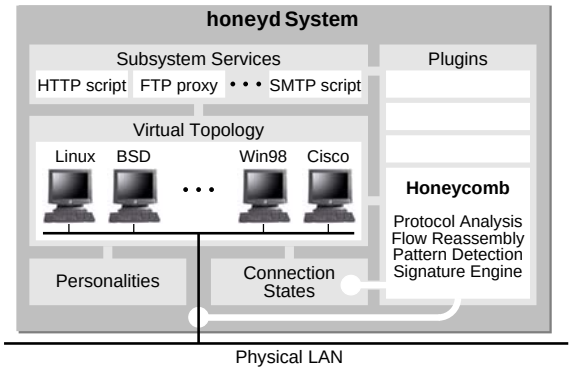

Fig. 1. Honeycomb's architecture, illustrated as a typical honeyd setup. honeyd is simulating a number of different machines, each running a number of pre-configured services. The Honeycomb plugin has hooked itself into the wire to see in- and outgoing connections, and into honeyd's connection state management.

[1]The plugins are implemented as shared libraries, dynamically linked in at runtime.

Integrating our system into honeyd has several advantages over a standalone *bump-in-the-wire* approach:

- No Duplication of Effort: Our system needs access to network traffic. For a standalone application, libpcap [11] would be an obvious choice. honeyd already does this – it inspects the network traffic using libpcap and passes the relevant packets to the network stacks of the simulated hosts and eventually to their configured subsystems. Our approach is a minimum-effort solution that avoids performance hits by making use of packet data already transferred to userspace.
- Avoidance of Cold-Start Issues: the bigger advantage lies in the fact that honeyd is not passively listening to traffic going in and out of the honeypot, it *creates* the traffic coming out of it through the simulated network stacks and the configured subsystems. By integrating Honeycomb into honeyd we avoid desynchronization from the current state of connections: when honeyd receives a packet that starts a new connection (whether in a legal fashion or not), Honeycomb *knows* that this starts the connection. The question whether it may have missed the beginning of the connection is a non-issue, in contrast to other systems that use the bump-in-the-wire approach[12][13].

B. Signature Creation Algorithm

The philosophy behind our approach is to keep the system free of any knowledge specific to certain application layer protocols. Each received packet causes Honeycomb to initiate the same sequence of activities:

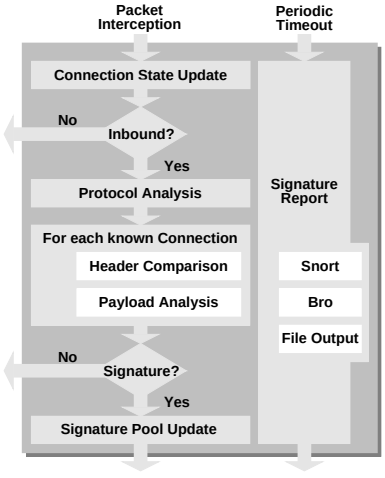

Fig. 2. High-level overview of Honeycomb's signature creation algorithm.

- If there is any existing connection state for the new packet, that state is updated, otherwise new state is created.
- If the packet is outbound, processing stops here.
- Honeycomb performs protocol analysis at the network and transport layer.

- For each stored connection:
  - Honeycomb performs header comparison in order to detect matching IP networks, initial TCP sequence numbers, etc.
  - If the connections have the same destination port, Honeycomb attempts pattern detection on the exchanged messages.
- If no useful signature was created in the previous step, processing stops. Otherwise, the signature is used to augment the *signature pool* as described in Section III-F.
- Periodically, the signature pool is logged in a configurable manner, for example by appending the Bro representation of the signatures to a file on disk.

Figure 2 illustrates the algorithm. Each activity is explained in more detail in the following sections.

C. Connection Tracking

Honeycomb maintains state for a limited number of TCP and UDP connections[2], but has rather unique requirements concerning network connection statekeeping. Since our aim is to generate signatures by comparing new traffic on the honeypot to previously seen one, we cannot release all connection state immediately when a connection is terminated. Instead, we only mark connections as terminated but keep them around as long as possible, or until we can be sure that we will not benefit from storing them any longer.

Connections that have exchanged lots of information are potentially more valuable for detecting matches with new traffic. The system must prevent aggressive port scans from overflowing the connection hashtables which would cause the valuable connections to be dropped. Therefore, both UDP and TCP connections are stored in a two-stage fashion: Connections are at first stored in a "handshake" table and move to an "established" table when actual payload is exchanged.

The system performs stream reassembly: for TCP, we reassemble flows up to a configurable total maximum of bytes exchanged in the connection. We store the reassembled stream as a list of exchanged messages up to a maximum allowed size, where a message is all the payload data that was transmitted in one direction without any payload (i.e., at most pure ACKs) going the other way. For example, a typical HTTP request is stored as two messages: one for the HTTP request and one for the HTTP reply. For UDP, we similarly create messages for all payload data going in one direction without payload data going the other way. Figure 3 illustrates the idea.

D. Protocol Analysis

After updating connection state, Honeycomb creates an empty signature record for the flow and starts inspecting the packet. Each signature record has a unique identifier and stores discovered *facts* (i.e., characteristic properties) about the currently investigated traffic independently of any particular NIDS signature language. The signature record is then augmented continuously throughout the detection process, maintaining a count of the number of facts recorded[3].

We currently perform protocol analysis at the network and transport layers for IP, TCP and UDP packet headers, using the header-walking technique previously used in traffic normalization [12]. Instead of correcting detected anomalies, we record them in the signature, for example invalid IP fragmentation offsets or unusual TCP flag combinations. Note that for these checks, Honeycomb does not need to perform any comparison to previously seen packets. We refer to a signature at this point as the *analysis signature*.

Honeycomb then performs header comparison with each currently stored connection of the same type (TCP or UDP). If the stored connection has already moved to the second-level hashtable, Honeycomb tries to look up the corresponding message and uses the headers associated with that message.

If any overlaps are detected (e.g., matching IP identifiers or address ranges), the analysis signature is cloned and becomes specific to the currently compared flows. The discovered facts are then recorded in the new signature.

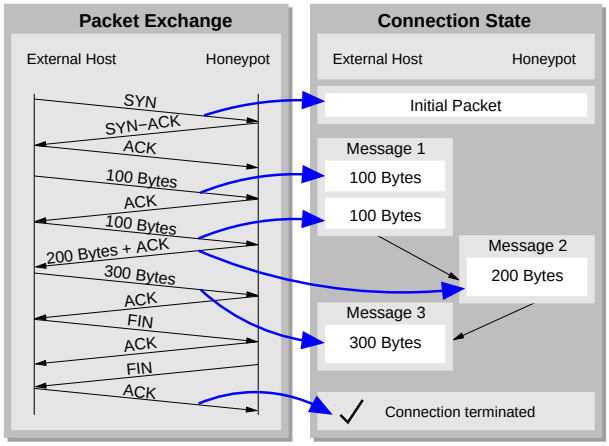

Fig. 3. A TCP packet exchange (left) and the way Honeycomb traces the connection (right). The packet initiating the connection is copied separately; afterwards, two 100-Byte payloads are received and assembled as one message. 200 Bytes follow in response, forming a new message. This in turn is answered by another 300 Bytes, forming the final message. The successful completion of the TCP teardown triggers the labeling of the connection as "terminated".

E. Pattern Detection in Flow Content

After protocol analysis, Honeycomb proceeds to the analysis of the reassembled flow content. Honeycomb applies the LCS algorithm to binary strings built out of the exchanged messages. It does this in two different ways, illustrated in Figures 4 and 5.

[2]When referring to UDP "connections" we just mean the packets exchanged using the same IP address and port number pairs.

[3]We use the terms "signature record" and "signature" interchangeably throughout the text, except for cases when we want to stress the difference between a signature record and a NIDS-specific signature string produced from the record.

- HORIZONTAL DETECTION: Assume that the number of messages in the current connection after the connection state update is n. Honeycomb then attempts pattern detection on the nth messages of all currently stored connections with the same destination port at the honeypot by applying the LCS algorithm to the payload strings directly.
- VERTICAL DETECTION: Honeycomb also concatenates incoming messages of an individual connection up to a configurable maximum number of bytes and feeds the concatenated messages of two different connections to the LCS algorithm. The point here is that horizontal detection will fail to detect patterns in interactive sessions like Telnet, whereas vertical detection will still work. More importantly, vertical detection also *masks* TCP dynamics: the concatenation suppresses the effects of slicing the communication flow into individual messages, which proved to be valuable (see Section IV-B).

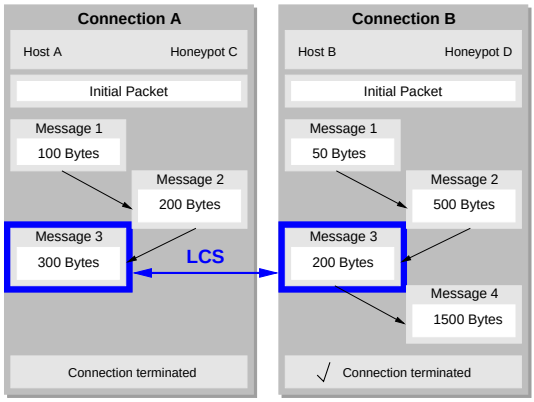

Fig. 4. Horizontal pattern detection: two messages at the same depth into the stream are passed as input to the LCS algorithm for detection.

In either case, if a common substring is found that exceeds a configurable minimum length, the substring is added to the signature as a new payload byte pattern.

F. Signature Lifecycle

If the signature record contains no facts at this point, processing of the current packet ends. Otherwise, Honeycomb checks hows the signature can be used to improve the signature pool, which represents the recent history of detected signatures.

The signature pool is implemented as a queue with configurable maximum size; once more signatures are detected that can be stored in the pool, old ones are dropped. Dropped signatures are not lost, since the contents of the signature pool are reported in regular intervals (see Section III-G).

Honeycomb tries to reduce the number of reported signatures as much as possible by performing *signature aggregation*. We have defined a number of relational operators for the generated signatures for this purpose:

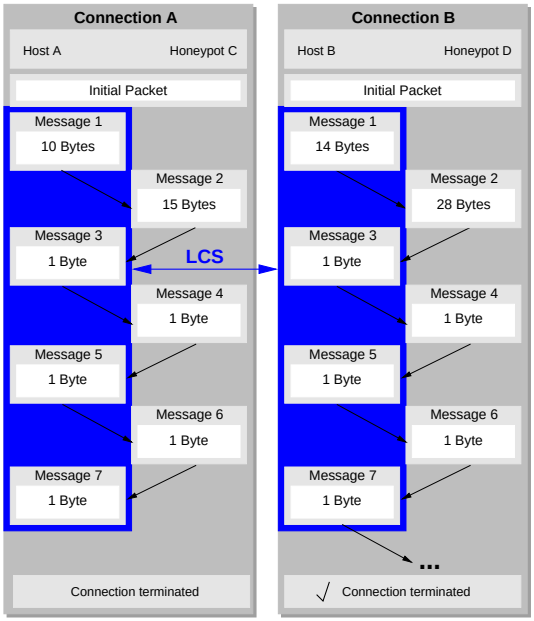

Fig. 5. Vertical pattern detection: for both connections, several incoming messages are concatenated into one string and then passed as input to the LCS algorithm for detection.

- $sig_1 = sig_2$: signature identity. This operator evaluates to true when sig_1 and sig_2 match in all attributes except those which can be expressed as lists in resulting signatures (e.g., ephemeral source port numbers).
- $sig_1 \subset sig_2$: signature sig_1 defines only a subset of sig_2's facts. This particularly includes any payload patterns detected by the LCS algorithm: A byte sequence b_1 is considered weaker than b_2 when b_1 is a substring of b_2.

If a new signature is a superset of an existing one, the new signature improves the old one, otherwise the new signature is added to the pool as a new entry.

G. Signature Output

The contents of the signature pool are periodically reported to an output module which implements the actual logging of the signature records. At the moment, there are modules that convert the signature records into Bro or pseudo-Snort format [4], and a module that dumps the signature strings to a file.

The periodic reporting scheme is an easy way to make sure all signatures are reported while in the signature pool and also allows for tracking of the evolution of signature records through the signature identifier in a post-processing stage.

IV. EVALUATION

The implementation consists of roughly 9000 lines of C code, with about 3000 lines for a separate library implementing the LCS algorithm. We tested our system on an unfiltered cable modem connection in three consecutive sessions,

[4]We require the ability to define a list of non-contiguous ports, and Snort currently does not permit this.

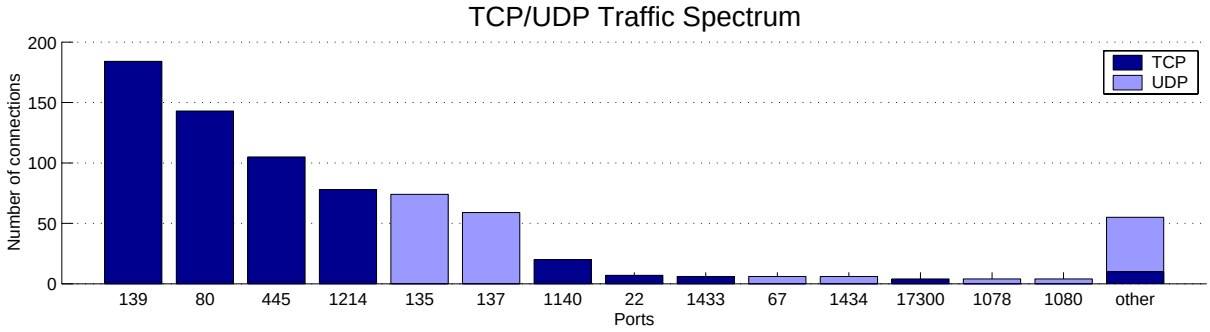

Fig. 6. Distribution of TCP and UDP traffic destination ports in packets directed at the honeypot, as observed in the 24 hours.

```
alert udp any any -> 192.168.169.2/32 1434 (msg: "Honeycomb Fri Jul 18 11h46m33 2003 "; content: "|04 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 DC C9 B0|B|EB 0E 01 01 01 01 01 01 01|p|AE|B |01|p|AE|B|90 90 90 90 90 90 90 90|h
|DC C9 B0|B|B8 01 01 01 01|1|C9 B1 18|P|E2 FD|5 |01 01 01 05|P|89 E5|Qh.dllhel32hkernQhounthickChGetTf|B9|llQh32.dhws2_f
|B9|etQhsockf|B9|toQhsend|BE 18 10 AE|B|8D|E|D4|P|FF 16|P|8D|E|E0|P|8D|E|F0|P|FF 16|P|BE 10 10 AE|B|8B 1E 8B 03|=U
|8B EC|Qt|05 BE 1C 10 AE|B|FF 16 FF D0|1|C9|QQP|81 F1 03 01 04 9B 81 F1 01 01 01 01|Q|8D|E|CC|P|8B|E|C0|P|FF 16|j|11|
j|02|j|02 FF D0|P|8D|E|C4|P|8B|E|C0|P|FF 16 89 C6 09 DB 81 F3|<a|D9 FF 8B|E|B4 8D 0C|@|8D 14 88 C1 E2 04 01 C2 C1 E2 08|
)|C2 8D 04 90 01 D8 89|E|B4|j|10 8D|E|B0|P1|C9|Qf|81 F1|x|01|Q|8D|E|03|P|8B|E|AC|P|FF D6 EB|"; )
```

Fig. 7. Signature Honeycomb created for the Slammer Worm.

```
alert tcp 80.0.0.0/8 any -> 192.168.169.2/32 80 (msg: "Honeycomb Mon May  5 16h59m09 2003 "; flags: A; flow: established;
content: "u|08 81|˜0|9A 02 00 00 0F 84 C4 00 00 00 C7|F0|9A 02 00 00 E8 0A 00 00 00|CodeRedII|00 8B 1C|$|FF|U|D8|f|0B C0
0F 95 85|8|FE FF FF C7 85|P|FE FF FF 01 00 00 00|j|00 8D 85|P|FE FF FF|P|8D 85|8|FE FF FF|P|8B|E|08 FF|p|08 FF 90 84 00
00 00 80 BD|8|FE FF FF 01|thS|FF|U|D4 FF|U|EC 01|E|84|i|BD|T|FE FF FF|,|01 00 00 81 C7|,|01 00 00 E8 D2 04 00 00 F7 D0
0F AF C7 89|F4|8D|E|88|Pj|00 FF|u|08 E8 05 00 00 00 E9 01 FF FF FF|j|00|j|00 FF|U|F0|P|FF|U|D0|Ou|D2 E8|;|05 00 00|i|BD|
T|FE FF FF 00|\&|05 81 C7 00|\&|05|W|FF|U|E8|j|00|j|16 FF|U|8C|j|FF FF|U|E8 EB F9 8B|F4)E|84|jd|FF|U|E8 8D 85|<|FE FF FF|
P|FF|U|C0 0F B7 85|<|FE FF FF|=|88 88 00 00|s|CF 0F B7 85|>|FE FF FF 83 F8 0A|s|C3|f|C7 85|p|FF FF FF 02 00|f|C7 85|r
|FF FF FF 00|P|E8|d|04 00 00 89 9D|t|FF FF FF|j|00|j|01|j|02 FF|U|B8 83 F8 FF|t|F2 89|E|80|j|01|Th˜f|04 80 FF|u|80 FF|U
|A4|Yj|10 8D 85|p|FF FF FF|P|FF|u|80 FF|U|B0 BB 01 00 00 00 0B C0|tK3|DB FF|U|94|=3’|00 00|u?|C7 85|h|FF FF FF 0A 00 00
00 C7 85|l|FF FF FF 00 00 00 00 C7 85|‘|FF FF FF 01 00 00 00 8B|E|80 89 85|d|FF FF FF 8D 85|h|FF FF FF|Pj|00 8D 85|’|FF
FF FF|Pj|00|j|01 FF|U|A0 93|j|00|Th˜f|04 80 FF|u|80 FF|U|A4|Y|83 FB 01|u1|E8 00 00 00 00|X-|D3 03 00 00|j|00|h|EA 0E 00
00|P|FF|u|80 FF|U|AC|=|EA 0E 00 00|u|11|j|00|j|01 8D 85|\|FE FF FF|P|FF|u|80 FF|U|A8 FF|u|80 FF|U|B4 E9 E7 FE FF FF BB
00 00 DF|w|81 C3 00 00 01 00 81 FB 00 00 00|xu|05 BB 00 00 F0 BF|‘|E8 0E 00 00 00 8B|d$|08|dg|8F|"; )
```

Fig. 8. Signature Honeycomb created for the CodeRed II Worm.

covering a total period of three days. We were particularly interested in the traffic patterns and signatures created for a typical home-user connection, which can be assumed to be often only weakly protected, if at all.

A. Traffic Characteristics

During the 24-hour period, we captured 224 KB of traffic, comprising 557 TCP connections, 145 UDP connections and 27 ICMP pings. Figure 6 shows the distribution of the ports requested at the honeypot, in terms of numbers of connections.

B. Signature Detection

Honeycomb created 38 signatures for hosts that just probed common ports. 25 signatures were created containing flow content strings. These are relatively long; on average they contain 136 bytes. The longest strings are those describing worms: Honeycomb managed to create precise signatures for the Slammer and CodeRed II worms, see Figures 7 and 8. Note that these are the overlaps after 6 hits of Slammer and 3 CodeRed II hits, so they are more reliable than a match found in only two flows! Also, Honeycomb did not report the typical, long HTTP GET string for CodeRed II; rather, it reported the longer string of bytes following afterwards.

C. Performance Overhead

We measured the performance overhead involved when running Honeycomb compared to normal `honeyd` operation; the results are displayed in Figure 9.

V. Discussion

Our tests show that Honeycomb works and produces interesting signatures. It has to be seen how easily the generated signatures can be used in production environments, and how the system performs under higher load. However, honeypots generally see only relatively little traffic, so this problem should be manageable.

The performance as illustrated in Figure 9 shows that Honeycomb can run for extended times on quiet sites without problems. More testing is necessary to investigate the performance on busy sites, particularly with traffic that leads to increased amounts of connection state.

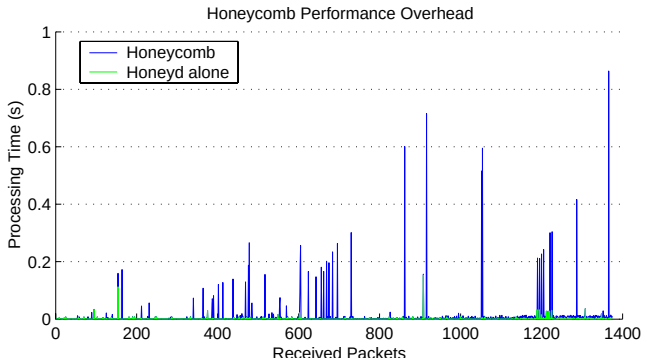

Fig. 9. Performance overhead when running Honeycomb. The packet processing times are almost entirely dominated by Honeycomb, so the `honeyd` part is hardly visible.

The system is rich in configuration parameters, which can have major influence on the performance of the system. Our tests were done with reasonable default values, which may not be the optimal values for the described environment.

VI. Summary

We have presented Honeycomb, a system that can produce NIDS signatures automatically by analyzing traffic on a honeypot. The system produces good-quality signatures on a typical end user's Internet connection. The system is particularly good at producing signatures for worms[5] — the signatures for Slammer and CodeRed II are extremely precise and were produced without any specific knowledge hardcoded into the system.

Honeypots are increasingly deployed in networks; however, they are mostly used *passively*: Administrators watch what happens and then manually perform forensic analysis once a machine has been compromised. Our results suggest that automated signature creation on honeypots is feasible and we believe our work is a first step towards integrating honeypots more closely into the security infrastructure.

Future Work

In the future, we want to expose Honeycomb to more aggressive traffic patterns to get a better understanding of its performance. We are working on reducing the amount of effort spent per arriving packet. Regarding the LCS algorithm, approximate matching schemes would allow us to create signatures that contain regular expressions. The system can currently be "distracted" by long, identical byte sequences that are inherent in some protocols like NetBIOS, potentially causing false positives. Future work needs to investigate means to mitigate this danger, for example using white-listing of known benign sequences. Better tools to analyze the created signatures and to ease their integration into the signature management toolchain is another interesting aspect.

[5]That is, as long as these worms are not polymorphic.

Generally, Honeycomb's approach could also be used for creating signatures of other traffic. On NIDS mailing lists, some of the most commonly asked questions are requests for signatures for a certain application or a particular exploit. Analysis of unsolicited email also comes to mind. Honeycomb could be a great tool for facilitating this — once a traffic stream with the traffic in question can be obtained, applying Honeycomb to this traffic could help provide the answer. We have in fact already used Honeycomb in order to verify existing signatures. We intend to provide a standalone application version of Honeycomb that performs signature generation on `libpcap` tracefiles instead of live traffic going through `honeyd`.

Acknowledgments

This work has been carried out in collaboration with Intel Research, Cambridge. We would like to thank Niels Provos for valuable feedback and for running the first `honeyd` competition, which was a major catalyst for the creation of this work. Thanks also to the reviewers for their helpful comments, to Alex Ho for providing his cable modem connection, and to the Castle Pub in Cambridge for hosting our brainstorming sessions.

References

[1] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 31, no. 23-24, pp. 2435–2463, 1998. [Online]. Available: http://citeseer.nj.nec.com/article/paxson98bro.html

[2] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th Conference on Systems Administration*, 1999, pp. 229–238.

[3] C. Stoll, *The Cuckoo's Egg*. Addison-Wesley, 1986.

[4] W. R. Cheswick, "An Evening with Berferd, in which a Cracker is lured, endured, and studied," in *Proceedings of the 1992 Winter USENIX Conference*, 1992.

[5] L. Spitzner, *Honeypots: Tracking Hackers*. Addison-Wesley, 2003. [Online]. Available: http://www.tracking-hackers.com/book/

[6] N. Provos, "Honeyd - A Virtual Honeypot Daemon," in *10th DFN-CERT Workshop, Hamburg, Germany*, February 2003.

[7] D. Gusfield, *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[8] P. Weiner, "Linear pattern matching algorithms," in *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[9] E. M. McCreight, "A space-economical suffix-tree construction algorithm," *Journal of the ACM*, vol. 23, pp. 262–272, 1976.

[10] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, no. 14, pp. 249–260, 1995.

[11] S. McCanne, C. Leres, and V. Jacobson, "tcpdump/libpcap," http://www.tcpdump.org/, 1994.

[12] M. Handley, C. Kreibich, and V. Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization, end End-to-End Protocol Semantics," in *Proceedings of the 9th USENIX Security Symposium*, 2000.

[13] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., Tech. Rep., 1998.