

Improving Honeynet Data Analysis

Camilo Viecco

Advanced Network Management Lab
 Indiana University

Abstract—The honeywall’s hflow and walleye interface first introduced in[1] vastly improved honeynet data analysis by integrating different data sources and thus reducing the time required for analyzing honeynet data. However, there are some open architectural questions. This paper answers some of these questions by introducing a packet processing language that allows a modular architecture. This architecture not only solves the immediate problems but is also applicable to a wide range of problems.

We present data regarding the problems of the old architecture and present our solution. We also present some of performance envelopes of both architectures.

I. INTRODUCTION

Honeypots and Honeynets today are part of the tool-set of network administrators and security researchers. One of the easiest ways to deploy honeynets is the use of the Honeynet Project’s Honeywall[2]. The Honeywall is a Linux distribution that contains a prepackaged system for rapid deployment of honeynets. The honeywall functionality can be divided into four separate components: administration, data control, data capture, and data analysis. Administration includes all the system and honeynet configuration such as users, authentication and IP addresses; Data capture includes all the mechanisms for collecting the data from the wire; data control includes all the mechanisms to limit the effect of compromised honeywalls on the rest of the network; and data analysis includes all the tools that are used to refine, transform and access the data captured;

The current Honeywall uses a data model first introduced by Balas and Viecco[1]. This model is implemented by a set of Perl scripts that collaborate to generate a composite view of the state of the honeypots and the network behavior. This paper addresses open questions with respect to hflow and walleye. This paper also present solutions to some of the usability problems encountered by honeynet researchers when using the Honeywall as a production tool for network analysis.

A. Problems with Hflow

We find six problems with the current hflow implementation: (i) It is unsuitable for packet trace reprocessing, (ii) has many startup dependencies, (iii) uses a problematic flow model, (iv) has 'high latency' update process, (v)

lacks transparency in its operation, and (vi) does not generate correct process data in some circumstances. Each of this problems is explained in more detail below.

Unsuitable for packet trace reprocessing. Currently hflow works by reading data from several running processes. Each of these helper processes are implicitly synchronized due to the fact that all are reading from data from a live data stream. There is no explicit synchronization or unified timestamp. Thus, previously collected traces cannot be processed by hflow as there is no way for synchronizing all the helper processes.

Startup dependences. All the helper processes must be executed before hflowd.pl starts and must be restarted manually if hflowd requires restart. This is partially solved by the HwCTL scripts in the honeywall; however, this solution is particularly brittle and prone to errors. An ideal system coordinate of all the dependencies for startup and shutdown.

Problematic Flow Model. hflowd uses Argus[3] as the source of flow data. While this removes the requirement for hflow to make its own flow model, the flow information produced by Argus is not 'accurate'¹ from the honeywall perspective. Further, Argus’s flow model separates in-band data from out-of-band data. This separation makes sense from an IP performance perspective but generates analysis overhead for security research.

Delayed update process Another effect of using Argus or any other external flow generator is that the latency on the updates of the coalesced structure is at best equal to the latency of the slowest component. For hflowd, the update delay is dominated by Argus’s collector(ragator) updates. This generates two problems. First, new operators cannot tell if the system is working as expected until a latency period has expired. Second, the latency does not permit the use of the honeywall as a real time tool.

Lack of Transparency Hflow is not able to detect runtime failures of any of the dependent processes. Thus, the system will continue to operate after process failures giving operators a false sense of confidence. The desire to keep working even in the presence of errors is a common

¹Some of the problems include: incorrect determination of source IP address on syn-ack initiated flows and incorrect determination of 'source' IP address on error ICMP flows

problem that is present in many analysis tools. It is the author's view that this behavior is only appropriate for 'flight recorder' type applications where the data are recorded or lost forever. For applications such as hflow, where reprocessing is an option, the application should produce an interrupt that is difficult to ignore, highly visible and as informative as possible. The reasoning is that is much better to have no processed data than flawed processed data.

Incorrect Process One of hflow responsibilities is to generate process data derived from sebek data. This fails in two conditions: daemonization and process rollover. The process rollover is partially addressed by the current implementation by making the couple process id parent process id unique, but there is evidence where this heuristic fails.

B. Problems with Walleye

The Walleye user interface gave end users a simple way to traverse and analyze for data. However we recognize two limitations in particular: (i) uses a very constrained query language and (ii) no direct GUI access to process centric viewpoints.

Very Constrained query Language. The selection system of walleye consists of simple, unique and non-hierarchical operations over a set of variable-value pairs. Filters on the variables can only take one value. This allows operators to focus only in single type of incident in an accurate manner. This type of filtering is not suitable for exploration as in many cases we would like to query not only on single property but a set of properties. Further, it is impossible to use the aggregate view in an iterative fashion as there are no way to filter on one dimension using the GUI.

Lack of direct access to process centric viewpoints. Several 'bugs' have been reported into the honeynet bugtrack system about failure to observe sebek data when the system was working as designed. There are two causes for this perceived invisibility of the data: First, a lack of understanding of the behavior of hflow and the walleye interface; Second, the lack of a simple way to access at least a part of the sebek data directly from the honeywall without information about the flows.

II. PREVIOUS WORK

This paper is mainly concerned with the honeywall and honeynet data analysis. There are three areas of research related to the work presented here: Honeynet Data Analysis, Dataflow Processing Languages and Flow Classification.

A. Honeynet Data analysis

High interaction honeypot/honeynet data analysis is usually done in a two tier process. First some tool is used to transform the raw captured and generated data (packets and packet metadata) into a set of related packages. Then

each conversation or system capture is parsed via specialized tools.

The honeynet project is the source of most of the Free Open Source Software(FOSS) data analysis tools for honeynet/honeypot data. The 'official' honeynet data analysis tools have undergone several revisions. The second generation data analysis used a relational model only for sebek[4] data. The third and last version [1] uses a relational model for both network (flow) data and host (sebek) data.

Honeysnap[5] is another recent tool from the honeynet project. Originally devised as a tool to process IRC conversations it has evolved into an important tool for data analysis. While honeysnap generates interesting summaries and transformations for data, it does not generate process and flow models and thus its use is limited for detailed host data analysis.

On the commercial arena, the honeynet console[6] is an example of a honeynet data analysis. The console includes some interesting statistical tools, but lacks the ability to integrate related data from different sources.

B. Dataflow Processing languages

The dataflow execution model was first devised in 1966[7]. In this execution model, a program is represented by a directed graph where the nodes represent instructions and the arcs represent the data dependencies between these instructions. Dataflow programming languages represent their programs in the dataflow execution model.

This model initially devised as a way to easily represent parallelism. Starting in the 1970's several efforts went to create hardware using this execution model.

Dataflow visual programming languages started on the 1980's. An example of a successful project is National Instrument's Labview [8].

Dataflow languages are found to be particularly useful for control systems and signal processing. Examples in this area include Matlab's simulink, octave[9] and scilab[10].

In the network arena an example of dataflow processing is the Click router [11]. This is a software architecture for building configurable routers. Some of the ideas behind the click router, such as validating packets only once and let the other modules assume packets are valid, allow for high performance.

C. Flow classification

Another small improvement of hflow is the aggregation a flow type classification by application. This is particularly useful since it has been documented that up to 40% of the packets in traces cannot be classified by their well known port [12].

A very large body of research is found in this area, but in general three approaches are used. These are payload based analysis, packet statistical analysis and behavior based intra flow analysis. Examples of payload based systems in-

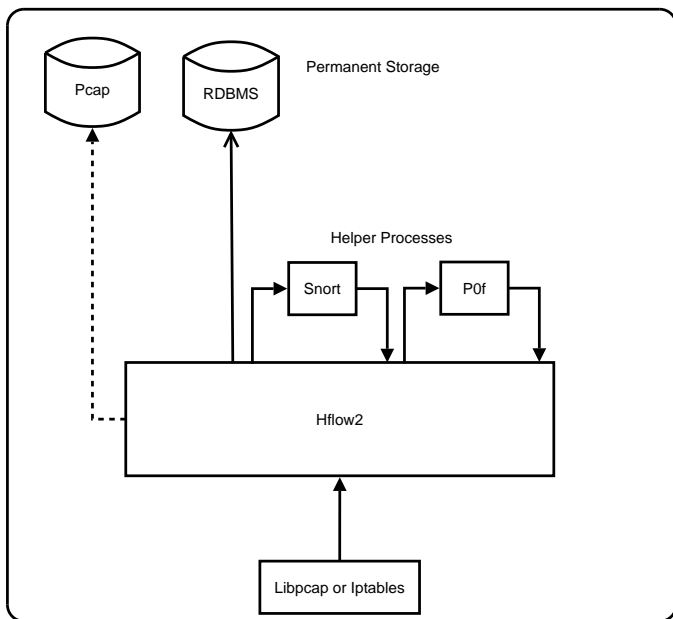


Fig. 1. Overall Hflow2 architecture.

clude `l7filter`[13], `hippie` and `pl0f`. Examples of statistical methods include the work by Crotti *et. al.*[14]. Their work uses packet size, inter-arrival time and arrival order to determine the application type. Thus they have access to the individual packets but do not require deep packet inspection.

'Network'² based methods include the works of Karagiannis *et al.*[15]. This method does not use deep packet inspection it uses netflow data. However this method does not classify each flow, but the behavior of Host-port pairs.

The block included in the current hflow implementation is a simple regular expression evaluator. The regular modular architecture implemented in this paper would allow for changing the classifier without changing anything else in the specification.

III. HFLOW2

Our research goal was to address all the challenges from the previous section, in particular the problem of how to make hflow capable of offline processing. Achieving this required a system that from the perspective of the helper tools was located at the libpcap level.

Figure 1 provides a high level view of the overall hflow2 architecture. Hflow2 is placed below the helper tools and is the only communication source/sink for the helper tools. Hflow synchronizes the helper applications and is in charge of their initialization and shutdown. Hflow generates and stores the composite view in a database, and potentially can also store the packet information in a pcap file.

²The Network here refers to the connection based generated graph

Thus hflow2 seemed to have not only a large complexity but also a large amount internal synchronization and interdependencies. We needed then to develop a solution that not only would be modular, to limit the complexity, but that also was self synchronizing and hopefully parallelizable. A Dataflow model was exactly what we needed.

A. A Dataflow Packet processing Language

Thus the solution became the challenge: how to define and implement a dataflow language for high performance packet processing. From the language perspective the only thing remaining was to define what are the data types for our language and determine if the language was to be data or demand driven. We decided to use a data driven model and have two data types: a tagged packet and a tagged flow. Packets can be tagged with flow information and flows can be tagged with packet information.

A.1 Data types and Intra Block Communication

There are two data types in the language: a tagged packet and a tagged flow. A tagged packet is made of the packet payload, a libpcap like packet header and a set of tags. A tagged flow is made up of a flow descriptor and a set of flow tags. The flow descriptor contains flow specific data such as source address, number of packets and type of flow.

Intra block communication is achieved by the use of the tags. Each block is free to look and modify the tags as seen by the block. Each block can also choose to modify the contents of the packets or not to forward any particular block.

There is, however one little difference worth mentioning, we will cheat: pure dataflow languages have no side effects, but we are using them. In particular, there is implicit communication via the database of the flow information generated by the flow maker. In a pure dataflow language, this information should be appended to the data passed, however as the size of the flow information is usually much larger than the information of a packet it is not reasonable to pass it to the next module. Our 'dataflow' language is thus not really pure.

A.2 Implementation

The question then came how to build an efficient dataflow language. Two things are of importance: performance and independence.

By performance we mean that the latency of each block in a string of blocks should not compromise the latency of the rest of the system. By independence we mean that modules should be able to operate independently of the rest of the system.

Each block is implemented as a C++ object. Hflow is a collection of prewired objects. An object oriented approach was selected to build the processing blocks as they follow

the object oriented approach. Further it solves potential names-space problems of other approaches.

Each block has a unique and common entry function that is called by the previous node. The execution patch follows a 'telescope' like approach. There is one constraint for each block, the entry function must have complexity class $O(1)$ or $O(\log n)$ and with no io blocking. This guarantees that for the complete telescope the complexity is $O(\log n)$ and thus a low latency. Blocks whose behavior cannot be satisfied need to create other threads to handle the blocking operations separately.

Another decision on the implementation is that on any unrecoverable error each module should fail and cause the application to exit. This is in conformance of our data analysis philosophy of :fail soon and noisily. The objective of this is that users can immediately know that something bad has happened and allow for reproduction of the failure.

B. Hflow Implemented Modules

For Hflow we developed several modules. The modules are grouped into several categories: input output modules, flow generators, interfaces to helper processes, flow classifiers, flow filters and packet filters. This section will describe each module in detail.

B.1 Input and Output modules

Two types of input modules are currently implemented: a libpcap input module and an iptables ULOG target input module. For output, only an libpcap module is implemented. The output modules are sink blocks, they have no output connection for any other blocks. The input blocks have no input connection, but only output connections. Further input modules 'self-fire' events from an external source, that being data from a file or live data from the system.

B.2 Flow module

There are many definitions and implementations of what constitutes a network flow. The reason is that not everyone agrees on what is a network flow and what data is required for a network flow to be useful.

An example on the large degree of differences is seen on the IETF draft IPFIX specification: there are 238 different potential flow fields (information elements) and not one required field. This means that being IPFIX compliant is not very useful in network security, because in network security the important aspect is not what could be represented but what is known to be represented and how this is useful for the particular analysis in mind.³

Another IETF working group also defines some different types of flows is the IP performance metrics group. Which has defined several and produced among others [16] which

³In other words, being IPFIX compliant does not give a lower bound on the richness of the data transferred

is the base of Argus[3]. Argus has gone under several revisions: two versions are worth mentioning: argus 2.0.6 and argus 3.0 rc. Argus however has some problems: version 2 had directionality problems and a flow model with some problems for icmp data. Version 3 removed the bidirectionality from the server and removed the helper application that we used: ragafor.

Thus the need for a causality generated directionality flows and the need to put both out of band data and in band data. In a the same flow record.

Our flows are bidirectional. Packets belong to a flow if: (i) they have a same 5 tuple: source IP, destination IP protocol, source port and destination port as a known flow. (ii) The 5 tuple reverse belongs to a known flow. (iii) they are end host ICMP errors that match a known flow 5 tuple.

ICMP related messages that are generated by intermediate routers are counted within a flow packets, but also create a separate router to end host flow.

For TCP and UDP packets the 'reverse' of a packet is found by swapping IP addresses and ports. For non TCP, UDP or ICMP packets the inverse is found by swapping IP addresses, the port values are kept at zero. For ICMP packets, the 'reverse' depends on the ICMP type and code. The current implementation only addresses 'reverse' for types specified in RFC792[17]. Directionality is given by the first packet that does not match any known flow given the rules above.

Besides argus like information, each flow also contains: a counter for the icmp related messages in each direction. An a set of flags to indicate what type of related icmp flags have been seen in each direction.

B.3 Sebek Module

The sebek Module is in charge of collecting only authentic sebek modules, interpreting them and inserting the sebek related data into the database. Thus its responsibility is similar to the previous *sebekd* and *hlfow* and *sbk_extract* programs.

Process ID (PID) rollover. A problem for long running honeywalls was the fact that the previous system assumed that each duple Parent Process ID (PPID) and PID were unique for the liveness of a system. This is not true for systems with rapid generation of children or for systems with large up times (more than one month).

Daemon processes: Another problem with sebekd was its inability to understand transitions from a regular process into Daemons (children of init).

We used a different approach: A process is new when: We have not seen the PID before or when the last PPID known than the new PPID and the new PPID is different than 1. This however implies having a record of what are the PID to PPID mappings.

This module is a two thread module. The main thread checks that the current packet is a sane sebek packet

(Source port, destination port, destination IP and magic number are what is expected) and if so inserts it into a queue. The second thread takes packets from the queue and processes them accordingly and inserts the appropriate data into a RDMBS.

To reduce the number of database transactions, the system keeps a memory record of the PPID, DBID and last time seen per each PID. And the database is updated or checked only when the internal data is stale or not found. To keep the size of the table finite, this state is only kept for $PID \leq 65536$ which is true for the default configurations of most OS.

B.4 P0f and Snort Modules

A crucial part of the system is the integration of Snort[18], [19] and P0f[20]. Since these are independent programs, they interface with hflow2 via two distinct threads. Both of these modules fork and execute an instance of the helper thread and pipe the stdin and stdout to unnamed pipes that are handled by hflow2. The forked processes are also setuid to mitigate the effects potential exploits in the helper applications

These modules consist of three threads. The main processing thread, makes a copy of the packet and places it in queue. The second thread reads from the queue and makes a libpcap packet that is written to helper application (Snort or P0f) stdin. The third and last thread reads the appropriate output from the helper applications, interpret it and communicates with the database to insert the interpreted value in the appropriate location.

It is worth noticing that hflow2's architecture objective is not to remove runtime dependencies on helper applications, but to remove the startup dependencies of helper applications.

B.5 Flow Classifier

One challenge when dealing with network data is that programs, in particular malicious programs, do not use well known ports to operate. This makes analysis harder. There are many approaches to application level classification, but we use the most simple: deep packet inspection. Our flow classifier identifies a flow when all the rules of a class are satisfied. Currently rules consist of: a PCRE regular expression to be matched against the packet payloads; a minimum and maximum packet number for the match to occur; a minimum and maximum byte number for the match to occur; and the direction of the classification: from client to server or from server to client.

B.6 BPF module and Fragmentation Module

One constant request for the honeywall was the inclusion of a Berkeley Packet Filter (BPF) filter at the hflow level.

While we do not agree on such filtering⁴ the addition of such filtering provides a simple way to provide inline filtering for other future modules.

The Fragmentation module is a specialization of the BPF module, it is actually just a BPF module that filters on IP fragments. Packets that are not the first section of a fragment get sent directly into snort all other packets are sent to the next module.

B.7 Class Filter

There are times when capturing the full packet payloads is of no interest because they cannot provide any additional information. The classic example is SSL traffic. As SSL provides perfect forward secrecy, the actual contents of the payload are not very useful. In such cases, storing only the packet and SSL headers is sufficient. When analysing data where this is the case, this type of filtering is extremely efficient at reducing the amount of data captured and thus needed for analysis.

C. The complete Hflow

With all the modules in place, the only remaining issue was the connection between modules and the initialization scripts. Figure 2 shows how the modules are connected in the current implementation.

This is not the only way to connect the components, in fact there are only two tight dependency sets: between the flow maker, the regexp classifier and the class filter; and between the input module, the bpf block and the IP fragmentation block. All other modules can be placed in many other configurations. Further, since all the modules are dynamic objects, all objects with only one input and output blocks can in theory be swapped at runtime by any other similar or different module.

D. Performance Envelopes

The final task remaining was to test the performance envelopes of the proposed system against the performance envelope of the previous generation. Two sets of experiments were done: one to test the effect on sebek rich data and one to test the performance of network data with no sebek data. The tests consisted of measuring the cpu usage of the system on steady state of previously generated packet traces replayed at different speeds.

⁴It is the author's opinion that a honeywall should collect everything as there is no *a priori* knowledge of what is good or bad in a honeywall. Some researches have suggested the use of such filters to reduce potential liabilities, due to privacy regulations, and to reduce some analysis overhead when other hosts are not of interest. These are well intended suggestions, but they miss one of the primary objectives of the honeywall: to collect the information exactly as it reached the honeypots so that accurate descriptions of the interaction of the outside world with the honeypots can be reconstructed. The addition of BPF filters allows for hidden information channels that can be potentially used by attackers.

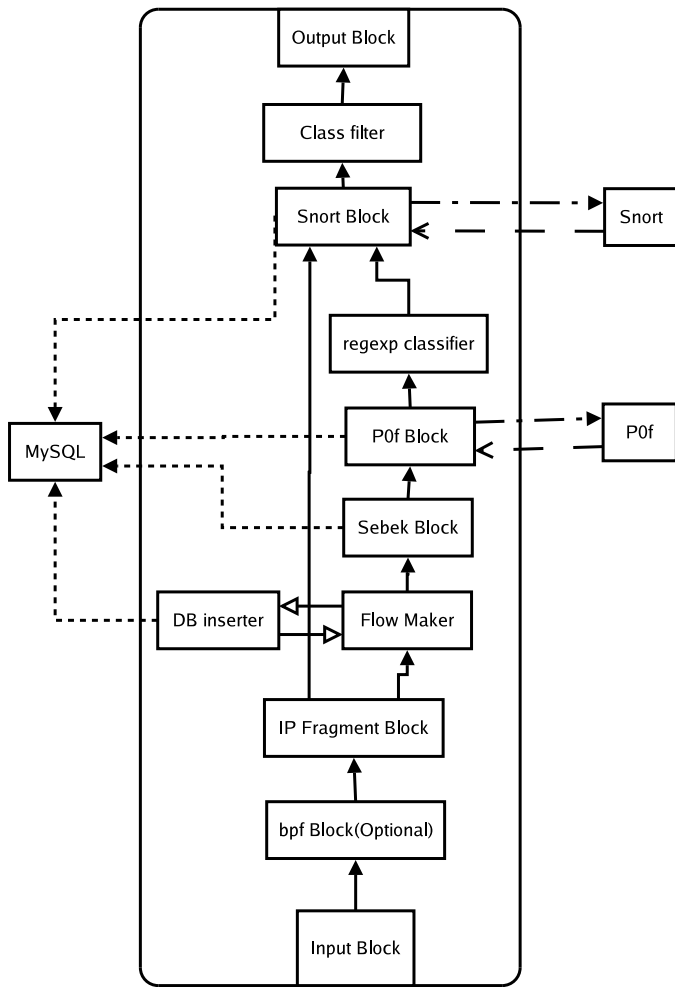


Fig. 2. Hflow2 Detail.

D.1 Equipment

To test the performance and limitations of our current and previous implementations we set up a roo honeywall version 1.0.hw-222. The only modification was the installation of MySQL 4.1.22 on the honeywall, as MySQL 4.0 or greater is required for hflow2. The Honeywall was installed in a 1.0 GHz Pentium III system with 512 MB of Ram with 100Mbps network interfaces. The test system was a similar system but with only one network card.

The tests consisted of different filtered versions of real honeypot data collected from an Indiana University honeypot. The pcap traces were injected back from the test system via tpreplay. It is worth mentioning that this packet injection is not packet reprocessing as the timestamps of the events are the timestamps generated by the packet injection and not the original timestamps of the recorded events.

The tests objective was to determine what are the performance envelopes of the coalescing system both old and

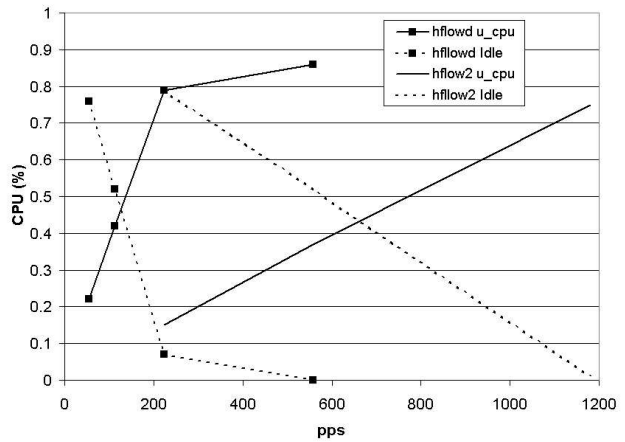


Fig. 3. Sebek trace CPU usages

new.

D.2 Sebekd Limits

This test consisted of injecting the known traces at different rates to the system. This test tries to simulate a DoS attack on the sebekd component of the data collection. The packet traces consisted of 97% sebek packets.

Figure 4 shows the percentile of user and idle CPU for both hflowd.pl and the new hflow. It can be seen that the original hflow starts to have problems at around 250 pps. The new system starts to have problems at around 1000 pps.

It is also noticeable that the user CPU for the original hflow is higher under saturation conditions (85%) than the saturation of the new hflow (75%). This translates that the system usage is higher, in particular the CPU usage of the database dominates both cases.

Further analysis indicates that the 1000pps limit is due to limitations on the insertion rate of the database. Recall that a portion of each sebek packet is inserted into the database. The system does provide a 5 times increase in the number of sebek packets required to DoS the data capture mechanisms. Thus it is important to deploy systems with filtered sebek[21] and with correct values for the magic number and ports so that the collection system is more robust to DoS attacks.

D.3 Non sebek limits

The second test was to determine the performance envelope for just network data (no sebek). The data contained on average continued 1 packet to be flagged as an alert every 300th package. Most of the CPU usage is caused by the insertion of alerts into the database. We tested only the new hflow as hflowd.pl contains a bug in the honeywall version that we used and no alerts were being inserted into the database.

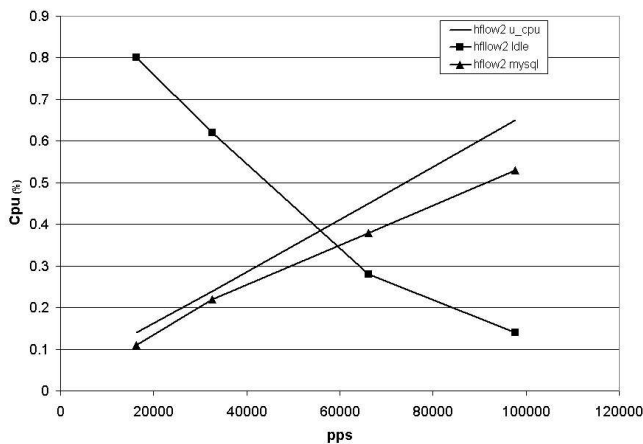


Fig. 4. No sebek trace CPU usages

We tested up to 97680 packets per second, and the system was able to keep up with no packet loss. The system also never reached 100% cpu usage.

IV. WALLEYE

Another part of the Honeywall that needed improvements was the data analysis section of the Walleye interface. Three problems are noted: First, The filtering abilities of Walleye are too coarse grained and not flexible enough; second, walleye did not included any direct way (by a set of user clicks) to access the sebek data and third, the system became less responsive as more data was inserted.

The first problem made many users redevelop tools similar to hflow to reprocess their data (even when the MySQL database was available). The second problem made many users believe that the system was not working properly as they tried to decode Sebek as a network flow and not access Walleye's process tools.

A. Reworking the filters

The filter remake included a two pronged approach: adding the ability to ask for multiple and negative selectors and modifying the user interface to access these changes. Multiple selectors allow to ask queries of the form: show me information about IP address x and IP address y. The addition of negative filters allowed the queries of the form: show me all data except for destination port 139. Combining these two allows for a much richer query interface.

The second part was to allow these filters to be controlled from the GUI. To do so two things were added: checkbox lines for the aggregate view and the ability to use the aggregate view on filtered data. The whole idea is to keep the user on the aggregate view as much as possible before entering the detailed view.

Figure 5 illustrates the changes in the GUI interface.

B. A simple Direct process view

The second change in walleye is the introduction of a direct process view. This view is similar to the aggregate view of the flows, but only has flow information. There is no sorting and the only ordering is by command name. The reason behind this are performance problems when building the aggregate view.

V. FUTURE WORK AND CONCLUSIONS

The biggest trade-off of the new hflow architecture is that we have a new single point of failure: the main hflow thread.

The current hflow model suffers from inserting too much data into the database. This is clear from the measures of the performance envelope of using sebek vs non sebek data. 100% cpu usage is found with only 1000 sebek packets per second. We should probably revisit the hflow model to move the sys_open data to a system with faster data rates.

More usability tests are required to determine other problems of the users of the system.

Improve each module, in particular the flow classification module. This module can be made not only more efficient but integrated with the walleye GUI.

The use of the dataflow language seems a promising way to create and develop new applications. It is an open question of how we can add new data types and connectors so that the system can be used also to display data directly. It is also promising to add a graphic interface to the system so that users could experiment with the modules in a graphic interface just like labview.

VI. ACKNOWLEDGMENTS

The author would like to thank Jean Camp and the anonymous reviewers by their comments. This work is sponsored by Indiana University's Advanced Network Management Lab and the Institute for Information Infrastructure Protection (I3P) research program.

REFERENCES

- [1] E. Balas and C. Viecco, "Towards a third generation data capture architecture for honeynets," in *Proceedings of the 6th Information Assurance Workshop*, IEEE, June 2005.
- [2] T. H. Project, "Know your enemy: Honeywall cdrom." <http://project.honeynet.org/papers/cdrom/index.html>, Feb 2005. Last access: March 2007.
- [3] "Argus project." <http://www.qosient.com/argus/>, 2004.
- [4] T. H. Project, "Know your enemy sebek." <http://project.honeynet.org/papers/sebek.pdf>, November 2003. Last access: Feb 2005.
- [5] D. Watson and A. Clune, "Honeysnap." <http://www.honeynet.org/tools/honeysnap/>, February 2007. Last Access: March 2007.
- [6] <http://www.activeworx.org/programs/hsc/index.htm>.
- [7] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," in *ACM Computing Surveys*, vol. 36, ACM, 2004.
- [8] N. Instruments, "Labview website." <http://www.ni.com/labview/>, 2007. Last Access: March 2007.

Data Analysis		System Admin		Customize CD-ROM		Logout									
March 2007 sun mon tue wed thu fri sat 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 (Prior Month) (Next Month) Hour Cons IDS 0:00 18 0 1:00 12 0 2:00 17 0 3:00 18 0 4:00 23 0 5:00 22 0 6:00 28 0		Aggregated Flows: Aggregated by dst_port Observed from Sensor 20 With IP Protocol undef Between Wed Mar 21 17:26:41 2007 and Wed Mar 21 18:26:41 2007 (Previous Page) Start 1 End (Next Page) 1 / 1													
Filter		Aggregate By		Aggregate Totals				Individual Flow Maximums							
Include	Exclude	Destination Port		Flows	Alerts	SRC Ports	DST Ports	SRC pkts	SRC bytes	DST pkts	DST bytes	SRC pkts	SRC bytes	DST pkts	DST bytes
<input type="checkbox"/>	<input type="checkbox"/>	24427		6	0	1	1	7	176	0	372	2	56	0	112
<input type="checkbox"/>	<input type="checkbox"/>	9001		1	0	1	1	3	84	0	168	3	84	0	168
<input type="checkbox"/>	<input type="checkbox"/>	1027		2	0	1	1	2	1,800	0	1,112	1	900	0	556
<input type="checkbox"/>	<input type="checkbox"/>	1026		2	0	1	1	2	1,800	0	1,112	1	900	0	556
<input type="checkbox"/>	<input type="checkbox"/>	ntp		12	0	1	1	12	672	12	672	1	56	1	56
<input type="checkbox"/>	<input type="checkbox"/>	0		4	0	1	1	4	214	4	214	1	58	1	58
<input type="checkbox"/>		Apply checkbox filters													

Fig. 5. Filter check-boxes in Walleye

- [9] J. W. Eaton, "octave website." <http://www.gnu.org/software/octave/about.html>, March 2007. Last Access: March 2007.
- [10] INRIA, "Scilab website." <http://www.scilab.org/>, 2007. Last Access: March 2007.
- [11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, August 2000.
- [12] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic application-layer protocol analysis for network intrusion detection," in *Usenix Security*, 2006.
- [13] <http://17-filter.sourceforge.net/>. Last access: March 2007.
- [14] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through statistical fingerprinting," in *ACG SIGCOMM Computer Communication Review*, vol. 37, January 2007.
- [15] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "Blink: Multilevel traffic classification in the dark," in *SIGCOMM '05*, ACM, August 2005.
- [16] M. Paxson, Almes and Mathis, "Framework for ip performance metrics," 1999.
- [17] J. Postel, "Internet control message protocol darpa internet program protocol specification," 1981.
- [18] M. Roesch, "Snort—lightweight intrusion detection for networks," in *Proceedings of LISA '99 Systems Administration Conference*, 1999.
- [19] S. Community, "Snort the open source network intrusion detection system." <http://www.snort.org/>, 2005. Last access: Feb 2005.
- [20] M. Zalewski, "passive os fingerprinting tool." <http://lcamtuf.coredump.cx/p0f.shtml>, 2004.
- [21] E. Balas, G. Travis, and C. Viecco, "A dynamic filtering technique for sebek system monitoring," in *Proceedings of the 7th Information Assurance Workshop*, IEEE, June 2006.