Saluki: a High-Performance Wi-Fi Sniffing Program

Keren Tan and David Kotz
Department of Computer Science
Institute for Security, Technology, and Society
Department of Computer Science, Dartmouth College
Hanover, NH, USA

Abstract—**Building a campus-wide wireless LAN measurement system faces many efficiency, scalability and security challenges. To address these challenges, we developed a distributed Wi-Fi sniffing program called Saluki. Compared to our previous implementation and to other available sniffing programs, Saluki has the following advantages: (1) its small footprint makes it suitable for a resource-constrained Linux platform, such as those in commercial Wi-Fi access points; (2) the frame-capture rate increased more than three-fold over tcpdump with minimal frame loss; (3) all traffic between this sniffer and the back-end server was secured using 128-bit encryption; and (4) the traffic load on the backbone network was reduced to only 30% of that in our previous implementation. In this paper, we introduce the design and the implementation details of this high-performance sniffing program, along with preliminary evaluation results.**

I. Introduction

As enterprises increasingly depend on wireless local-area networks (WLANs) for mission-critical applications, it is important to monitor such networks to recognize usage patterns, diagnose malfunctions, and detect any abnormal behaviors that would disrupt or degrade the network operation. To support these activities, we have built a campus-wide, distributed WLAN monitoring system, the Dartmouth Internet Security Testbed (DIST) [1], at Dartmouth College. In this paper, we introduce the design and implementation of a distributed, high-performance, and secure Wi-Fi sniffing program, Saluki,[1] which is a core component of the DIST system. This sniffing program is now running on over 210 Air Monitors (AMs) across the Dartmouth College campus to monitor WLAN network activities.

While Saluki shares many of the same features as other passive network sniffing software tools, its design was driven by our past experience and the special needs of the DIST project. In the MAP project [3], the predecessor of the DIST project, we implemented a building-wide WLAN monitoring system. However, when we scaled the deployment from a *building* to a *campus*, and when we needed a higher level of security, our previously developed sniffing program no longer fit the needs. Our new sniffing program, Saluki, was designed to address these efficiency, scalability, and security challenges. Compared to our previous implementation and to other available sniffing programs, Saluki has the following advantages: (1) its small footprint makes it suitable for a resource-constrained Linux platform, such as those in commercial Wi-Fi access points; (2) the frame-capture rate increased more than three-fold with minimal frame loss; (3) all traffic between the sniffer and the back-end server was secured using 128-bit encryption; and (4) under the same frame-capture rate, the traffic load on the backbone network was reduced to only 30% of that in our previous implementation.

[1]*Saluki* is the name of a speedy hunting dog. As one of the earliest breeds to diverge from wolves, it is known for its beauty, speed and endurance [2].

A. Passive network sniffing programs

Many passive network sniffing programs have been developed for either wired or wireless network measurement; the best-known are tcpdump [4], Wireshark [5] and Kismet [6]. tcpdump is a command-line network sniffing and parsing tool ported to several platforms. Wireshark is similar to tcpdump, but with a graphical user interface and many advanced sorting and filtering options. tcpdump and Wireshark can work on both wired and wireless networks. Kismet is a sniffing program (and an intrusion-detection system) dedicated to wireless LANs. Unlike tcpdump and Wireshark, Kismet supports a "server/drone" configuration in which the drone captures the wireless data and forwards it to a Kismet server via a secondary connection, such as Ethernet. We developed a sniffing program called dingo [7], [8] in our previous MAP project [3]; it supports several advanced features, such as channel sampling, data aggregation, dynamic filtering, and refocusing. However, dingo's performance deteriorates quickly when dealing with high-volume traffic. Jigsaw [9] is a sniffing program developed specifically for some Atheros chipsets that use an old version of the madwifi driver [10]. Jigsaw patches this specific madwifi driver to grab some statistics about physical layer events, at the expense of loss of portability. As with Kismet, both dingo and Jigsaw can work in a distributed client/server mode, that is, they run on an array of remotely deployed AMs and forward the captured traffic to central servers.

B. DIST

Table I compares our new sniffing program with the above-mentioned programs. In this table, Saluki provides the most complete feature set; these features were chosen to address the efficiency, scalability and security challenges we encountered in building the DIST system. Although designed for wireless sniffing, Saluki can also support wired network measurement.

DIST is one of the largest wireless network measurement systems installed for research: 210 AMs deployed, 10 buildings covered and more than 5,000 wireless network users monitored. Based on our preliminary measurements, more than 3.5 terabytes of frame headers will be captured by 210 AMs

TABLE I
COMPARISON OF PASSIVE NETWORK SNIFFING PROGRAM

		tcpdump	Wireshark	Kismet	dingo	Jigsaw	Saluki
Features	wired/wireless network	Both	Both	Wireless	Wireless	Wireless	Both
	client/server mode	No	No	Yes	Yes	Yes	Yes
	data aggregation	No	No	No	Yes	Yes	Yes
	data encryption	No	No	No	Yes	No	Yes
	data compression	No	No	No	No	Yes	Yes
	Wi-Fi channel sampling	No	No	Yes	Yes	No	Yes
Supported platform		Unix, Linux, Windows	Unix, Linux, Windows	Linux	Linux	Linux (some Atheros chipsets)	Linux

each day. We use the Aruba AP70 [11] (flashed with OpenWrt Linux [12])[2] as our AM's hardware platform. The advantage of the Aruba AP70 is that it fully complies with IEEE 802.3af standard for Power over Ethernet (simplifying installation), provides diverse interfaces (USB, serial, and Ethernet), and has a compatible appearance to other devices in our deployment environment. Because the Aruba AP70 was originally designed to be a commercial AP (Access Point) instead of a wireless AM, its processing capability is limited; indeed, just to put it in context, even smart phones have more memory and CPU power, as shown in Table II.

Because DIST is distributed across campus, Saluki works in a client/server mode: a front end runs on the remote AM, capturing and forwarding traffic to our servers via the campus backbone network. Two concerns drove many design decisions. First, since DIST is monitoring Dartmouth's production wireless network, the collected traces may contain sensitive information related to human privacy, such as MAC/IP address and TCP/UDP payloads. There is a risk that an adversary could intercept the traffic between the AM and server and jeopardize user privacy. To protect privacy, we limit what we capture and we encrypt all the traffic between the AMs and the server. Second, the DIST servers are located in the computer science department. Since these servers share a 1Gbps link with more than 200 other machines in the department, 3.5 terabytes per day through this link will negatively affect other machines' network performance. To efficiently use the bandwidth, we must use data compression to alleviate the pressure on the shared link.

Given that the Aruba AP70 is a resource-constrained hardware platform (Table II), our sniffing program faces great challenges in supporting encryption, compression, and traffic forwarding functions while maintaining high sniffing performance.

II. APPROACH

We took a divide-and-conquer approach in designing and implementing Saluki. In this section, we present it in the same manner.

[2]Saluki does not depend on a particular Linux kernel or OpenWrt distribution.

TABLE II
COMPARISON OF ARUBA AP70 TO OTHER PLATFORMS

	Aruba AP70	Linksys WRT54GL [13]	iPhone 3Gs [14]
CPU	266MHz MIPS 4Kc	200MHz MIPS	600MHz ARM
Memory (RAM)	28MB	16MB	256MB
Storage	8MB	4MB	16/32GB

A. Capture interface

We use a raw socket with PACKET_MMAP enabled as the capture interface. The raw socket lets us bypass the protocol stacks (the link layer and above) inside the Linux kernel, and the PACKET_MMAP provides an efficient way for communication between the kernel space and the user space. We adopted this interface instead of using the popular libpcap library [4] for efficiency reasons. The advantage of libpcap is that it provides a portable and architecture-independent interface for packet capture on several operating systems. Since we plan only to support the Linux operating system, this extra level of abstraction was unnecessary and indeed cost precious memory space and CPU time on the Aruba AP70.

In the Linux kernel, PACKET_MMAP is specifically designed to facilitate the network traffic capturing task. Without this socket interface, capturing each network packet requires a system call. PACKET_MMAP minimizes the number of system calls by implementing a configurable circular buffer between the user and the kernel space, and thus capturing a packet in the user space becomes a simple read operation on the shared circular buffer.

The raw PACKET socket (with PACKET_MMAP) turned out to be a highly efficient capture interface on the AP70's wireless NIC. In one test run (simply capturing frames and counting, nothing else), this interface was able to capture 7,063 frames per second with 25%-35% CPU usage and 3.3% frame loss. As a comparison, when we ran tcpdump with libpcap 0.9.8 under the same traffic load, it made the system freeze.

It is worth noting that since version 1.0.0 (released in 2008) libpcap has added partial support for PACKET_MMAP, although useful parameters to configure the circular buffer are not exposed to library users, and (as of this writing) it has not been ported to the Aruba AP70.

TABLE III
UDP THROUGHPUT FOR DIFFERENT DATAGRAM SIZE

Datagram size (bytes)	Throughput (KBps)
10	44.8
50	222.9
100	443.9
200	879.8
1000	4155.6
1500	5326.7
2500	7443.3
3000	7875.5

TABLE IV
DIST COMBO FRAME HEADER DEFINITION

	Size	Meaning
ver	16 bits	combo frame format version number
amid	16 bits	which AM sent out this combo frame
seqn	16 bits	combo frame sequence number
frn	16 bits	number of frames inside this combo frame
dsz	32 bits	size of uncompressed data in bytes
mgc	32 bits	magic number to check frame corruption

B. Data aggregation

Saluki uses UDP packets to forward the captured traffic back to our central servers. We observed that if we pack only one frame in each UDP packet, the 100Mbps Ethernet connection on the Aruba AP70 could not keep up when there was a high volume of wireless traffic. We measured the maximum throughput under different UDP datagram sizes as shown in Table III. We can see that small UDP packets degrade the Ethernet throughput greatly. Given that small frames, like a 14-byte ACK frame, are widely used in the IEEE 802.11 MAC layer, it is much more efficient to aggregate multiple frames and then send them as a "combo" frame. A DIST combo frame has two sections: the header section and the data section. The header section contains meta information about this combo frame as listed in Table IV, and the data section holds multiple captured frames. When a new frame is captured, Saluki appends the frame size and the frame content to the DIST combo frame's data section.

It is worth noting that there is a trade-off between the size of the combo frame and the frame-receipt delay at the server side. While a bigger combo frame will use the Ethernet connection more efficiently, bigger is not always better, especially for time-critical applications, like wireless-network intrusion detection. For this reason, we defined two adjustable criteria to decide when a combo frame should be sent: when the payload size of a combo frame exceeds a size threshold, or when the time difference between the first enclosed IEEE 802.11 frames and the current system clock exceeds a time threshold. In our current implementation we set these two parameters to 14KB and 1 second, respectively.[3]

[3]In this paper, 1KB = 1024 bytes, 1Kb = 1024 bits, 1B = 1 byte = 8 bits, and 1b = 1 bit. 1KBps = 1024 bytes per second, and 1Mbps = 1024×1024 bits per second.

C. Data compression

The DIST combo frame increases Saluki's network efficiency, but we need to do better. To more efficiently use the bandwidth on the backbone network, we compress a combo frame before sending it. Given the Aruba AP70's limited processing power, instead of pursuing the maximum compression ratio, we aimed to find a lossless compressor that has a good balance between processing speed and compression ratio.

After some background study, we focused on two variants of the Lempel-Ziv (LZ) compression method [15]: QuickLZ [16] and FastLZ [17]. Compared to the standard LZ compressor, these two variants trade compression ratio in favor of speed. It is important to note that a compressor's performance (compression ratio and speed) may vary when dealing with different data. We chose QuickLZ because it had a more consistent performance on our captured network data. In our experiments, a 14KB combo frame was compressed to 2.8-3.6KB by QuickLZ. The use of compressed combo frames saved more than 70% of the load on the backbone network, compared with sending individual uncompressed frame headers in each UDP packet.

D. Data encryption

As a basic security measure to protect the privacy of the network users whose traffic we capture, we encrypted all traffic between each AM and the central servers.

Encryption ciphers can be classified into two categories based on their operation mode: block ciphers and stream ciphers. A block cipher operates on data blocks, usually of fixed size, and a stream cipher operates on a continuous stream of data. We chose a stream cipher over a block cipher for two reasons: speed and security. First, a stream cipher generally will be much faster than a block cipher. Second, when using the same encryption key, there is a strict one-to-one mapping between the plaintext and the ciphertext for a block cipher, whereas there is no such one-to-one mapping for a stream cipher [18]. For DIST, this property of block ciphers could be a potential security flaw, because all possible values in many fields of Radiotap header and IEEE 802.11 header can be easily enumerated, and thus a block cipher may facilitate attacks by providing a much smaller search space than a stream cipher.

We evaluated all stream ciphers from the eSTREAM project [19] and the SNOW 2.0 cipher [20]. The best two ciphers were Rabbit and SNOW 2.0. Both support 128-bit encryption.

We evaluated an assembly-language implementation of the Rabbit cipher optimized for the MIPS 4Kc processor, whereas SNOW 2.0 is implemented in the C language and was not specifically optimized for this processor. Since our goal was to transmit the protected data most efficiently, we tried the ciphers both without compression and in combination with compression. We observed the following.

1. For stream ciphers, Rabbit emerged as a winner on the Aruba AP70, surpassing SNOW 2.0. When executing 5000

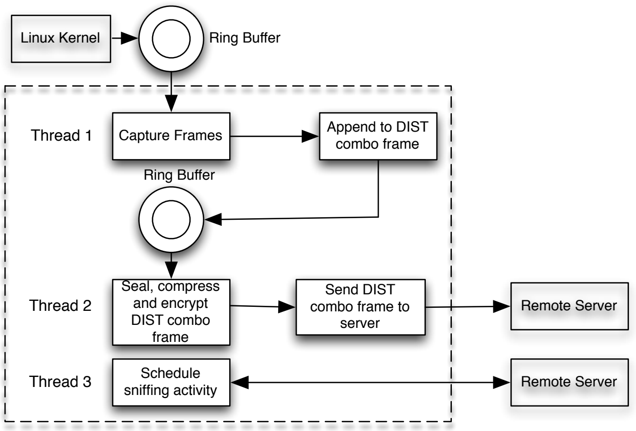
Linux Kernel
Ring Buffer
Thread 1
Capture Frames
Append to DIST combo frame
Ring Buffer
Thread 2
Seal, compress and encrypt DIST combo frame
Send DIST combo frame to server
Remote Server
Thread 3
Schedule sniffing activity
Remote Server

Fig. 1. The Saluki architecture.

loops on 14KB data, Rabbit took 5.33–5.55 seconds, whereas SNOW 2.0 took 7.42–7.73 seconds.

2. Adding compression decreases the total processing time, because there were fewer bytes to encrypt. In effect, compression was computationally "free".

Securely transmitting 5000 14KB combo frames (each combo frame may contain tens to hundreds of captured Radiotap and IEEE 802.11 frames) to a DIST server took 6.2–6.4 seconds, which encompassed two operations: encryption and UDP forwarding. The load on the network averaged 14KB per combo frame. If we compressed these combo frames first, however, handling them took *less* time, namely 5.3–5.4 seconds for *three* operations: compression + encryption + UDP forwarding. The required network bandwidth was also reduced by more than 70% (from 14KB per combo frame to 2.8-3.4KB per combo frame). This result illustrates that an efficient compression not only saves network bandwidth, but also reduces CPU time needed for encryption and UDP forwarding. If needed, we could set the size of the uncompressed DIST combo frame to be larger than 14KB. Although this change may improve the network throughput (Table III), it comes at the expense of increased delay at the server side (Section II-B).

E. Multithreading

So far we have introduced four core components of the Saluki sniffing program: capture interface, data aggregation, compression, and encryption. The final important task is to assemble them efficiently. Each of these components is relatively self-contained and can work independently from other components. For example, capturing frames from the Wi-Fi interface and forwarding DIST combo frames via Ethernet are I/O-intensive operations, while data compression and encryption are CPU-intensive operations. This observation inspired us to fit these components into a multithreading pipeline. We experimented with several schemes about which component should go to which thread, and Figure 1 shows the final and optimal configuration.

From Figure 1, we can see that Saluki has three threads. Thread 1 and Thread 2 are in charge of data capturing, processing, and forwarding. Thread 3 is the control thread that deals with scheduling and channel-hopping tasks (as in dingo [7], [8]). Two ring buffers are used in this program. The top ring buffer is responsible for mapping the captured frames from kernel space to user space and is emptied by Thread 1. The second ring buffer connects Thread 1 with Thread 2. From the perspective of multithreaded programming, the communication through these two ring buffers follows the classic writer/reader programming model.

Instead of putting compression, encryption and UDP forwarding all in Thread 2, we had planned to divide them between two threads: compression and encryption in one thread, and UDP forwarding in another. In the test run, we observed that Thread 1 and Thread 2 had similar CPU usage. Due to this observation, we did not split Thread 2 further.

III. Evaluation

In this section, we evaluate Saluki in terms of memory usage, CPU usage, frame-capture rate, and frame-loss ratio. Because tcpdump, Kismet and dingo are all built on libpcap, and tcpdump is the simplest (and should also be the fastest) among them, we used tcpdump as the baseline for comparison. To release tcpdump's maximum potential [21], we let it output to /dev/null instead of the screen or a file[4]. We set the capture size for tcpdump and Saluki to 192 bytes.

We set up two laptops (each a Thinkpad T42 with 1.6GHz Pentium M CPU and 1.5GB RAM) to act as the IEEE 802.11g Access Point and the client respectively. These two laptops were placed about 2 meters (6 feet) from each other, and one Aruba AP70 sniffer was placed halfway between them. We used Iperf [22] as the traffic generator running on two laptops.

A. Memory usage

We used the Linux command "top" to query memory usage. During execution, Saluki occupied 660KB RAM, and tcpdump used 740KB RAM. Note that, since tcpdump is dynamically linked with libpcap, its actual memory usage was larger than 740KB if the memory used by libpcap were counted. Of the 660KB RAM consumed by Saluki, much of it was allocated to various buffers for better performance. For example, the size of the second ring buffer (connecting Thread 1 and Thread 2) was about 90KB, and the size of the compression and encryption buffers were about 30KB each. If needed, one can reduce Saluki's memory usage by shrinking these buffers.

B. Capture performance

Figures 2, 3 and 4 show the performance in terms of frame-capture rate, frame-loss ratio and CPU usage. The frame-capture rate measures the speed that a sniffing program captures frames in the unit of *frames per second* (fps). The frame-loss ratio is the ratio of *the number of lost frames reported by the OS kernel* to *the sum of the number of captured frames*

[4]That is, `tcpdump -i ath0 -n -s 192 -w /dev/null`

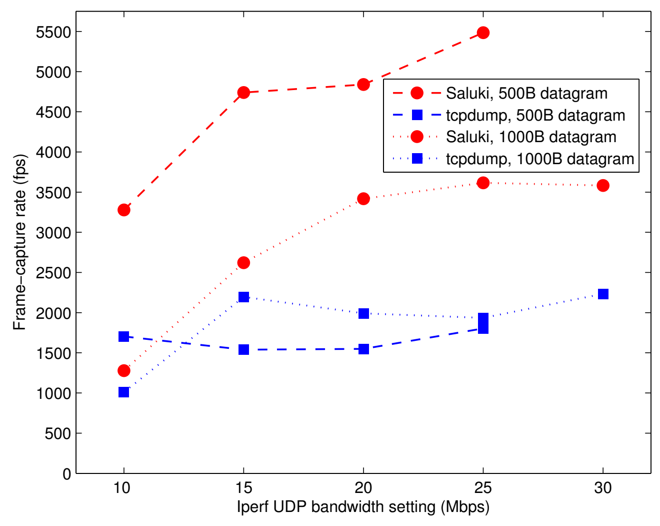

Fig. 2. Comparison of frame-capture rate.

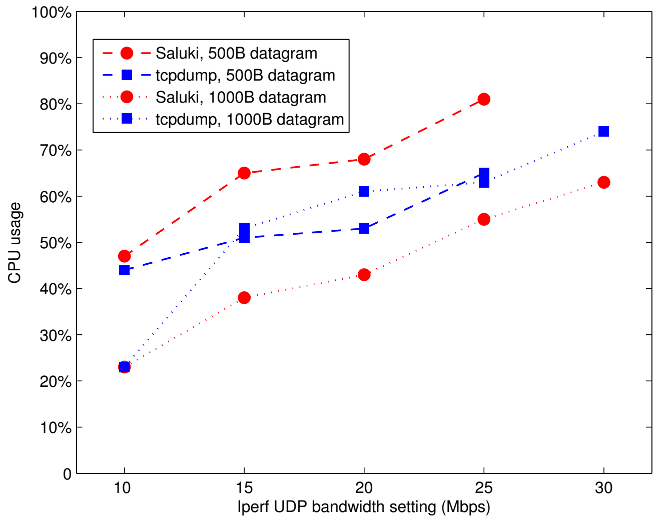

Fig. 4. Comparison of CPU usage.

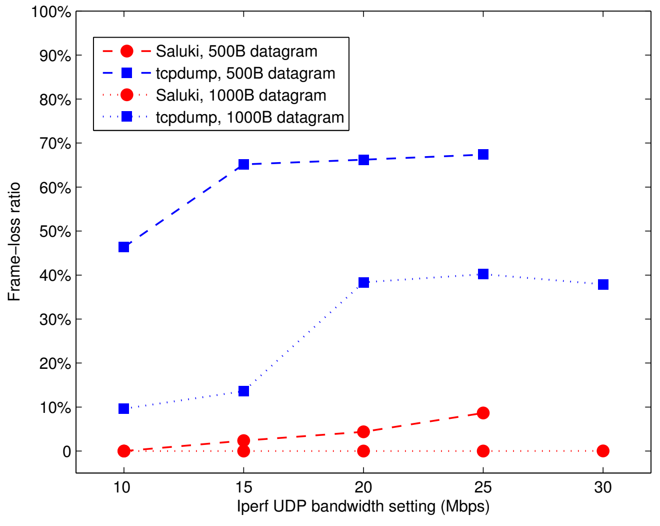

Fig. 3. Comparison of frame-loss ratio.

and lost frames. Since Saluki is a multithreaded program, its CPU usage in Figure 4 is the sum of all its threads' usage.

We used Iperf to generate UDP traffic with 500B and 1000B datagrams under four UDP bandwidth settings: 10Mbps, 15Mbps, 20Mbps, 25Mbps and 30Mbps. Each experiment setting ran for 200 seconds. Two things are worth noting. First, these five bandwidth settings are the parameters we gave to Iperf; however, in reality, the actual bandwidth could be a bit lower than the setting. Second, for a given bandwidth setting, Iperf must generate many more small-size packets than large ones to achieve that bandwidth. Due to the limited CPU power on the laptop, we could not generate enough 500B UDP packets to reach 30Mbps. Thus we did not provide a result for this setting.

Figure 2 shows that Saluki captured frames much faster than tcpdump under all settings even though Saluki needed to complete much more work (data compression, data encryption, and UDP forwarding) than tcpdump. Saluki's advantage became more obvious when dealing with high-speed traffic. When Saluki captured 5,488 frames per second, tcpdump only captured 1,802 frames per second. In this case, Saluki captured more than three times as many frames as tcpdump did.

Figure 3 demonstrates that Saluki's frame-loss ratio was significantly lower than tcpdump's. For UDP traffic with 1000B datagrams, Saluki's frame-loss ratio was nearly always zero (except for 0.028% under 30Mbps), while tcpdump could lose around 40% of frames. For UDP traffic with 500B datagrams, the disparity was more obvious (8.6% vs. 67.4% in the worst case here).

Here is an interesting phenomenon: by comparing "tcpdump, 1000B datagram" to "tcpdump, 500B datagram" in Figure 2, we can see that tcpdump usually captured 500B frames at a lower rate than it captured 1000B frames, even though Iperf sent them at a higher rate. In Figure 3 one can see that tcpdump lost a much higher fraction of 500B frames. Our hypothesis is that tcpdump dropped many "half-processed" frames when new frames came in.

Figure 4 summarizes Saluki and tcpdump's CPU usage. When there was not too much traffic, their CPU usages were comparable. When traffic volume was high, Saluki's CPU usage was higher than tcpdump's. Considering Saluki captured more than three times as many frames and included other work, this amount of increased CPU usage, however, is reasonable.

IV. Summary

We introduce Saluki, a high-performance and secure Wi-Fi sniffing program. Its design was driven by our past experience and the special needs of a large-scale wireless network measurement system. Compared to our previous implementation and to other available sniffing programs, Saluki has the following advantages: (1) its small footprint makes it suitable for a resource-constrained Linux platform, such as those in commercial Wi-Fi access points; (2) the frame-capture rate increased more than three-fold with minimal frame loss; (3) all traffic between the sniffer and the back-end server was secured using 128-bit encryption; and (4) under the same frame-capture rate, the traffic load on the backbone network was reduced to only 30% of that in our previous implementation. Saluki's source code will be released at CRAWDAD [23] in the future.

Acknowledgments

We thank Bennet Vance and Chris McDonald for their help and valuable suggestions, and the anonymous reviewers for their helpful feedback. This paper results from a research program in the Institute for Security, Technology, and Society (ISTS), supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001 and by the NetSANI project at Dartmouth College, funded by Award CNS-0831409 from the National Science Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security or the National Science Foundation.

References

[1] S. Bratus, D. Kotz, K. Tan, W. Taylor, A. Shubina, B. Vance, and M. E. Locasto, "Dartmouth Internet Security Testbed (DIST): building a campus-wide wireless testbed," in *Proceedings of the Workshop on Cyber Security Experimentation and Test (CSET)*. USENIX Association, August 2009.

[2] "Saluki," 2010. Available online: http://en.wikipedia.org/wiki/Saluki

[3] Y. Sheng, G. Chen, H. Yin, K. Tan, U. Deshpande, B. Vance, D. Kotz, A. Campbell, C. McDonald, T. Henderson, and J. Wright, "MAP: A scalable monitoring system for dependable 802.11 wireless networks," *IEEE Wireless Communications*, vol. 15, no. 5, pp. 10–18, October 2008. DOI 10.1109/MWC.2008.4653127

[4] "tcpdump/libpcap public repository," 2010. Available online: http://www.tcpdump.org

[5] "Wireshark," 2010. Available online: http://www.wireshark.org

[6] "Kismet," 2010. Available online: http://www.kismetwireless.net

[7] U. Deshpande, C. McDonald, and D. Kotz, "Refocusing in 802.11 wireless measurement," in *Proceedings of the Passive and Active Measurement Conference (PAM 2008)*, ser. Lecture Notes in Computer Science, vol. 4979. Springer-Verlag, April 2008, pp. 142–151. DOI 10.1007/978-3-540-79232-1_15

[8] ——, "Coordinated sampling to improve the efficiency of wireless network monitoring," in *Proceedings of the Fifteenth IEEE International Conference on Networks (ICON)*. IEEE Computer Society Press, November 2007, pp. 353–358. DOI 10.1109/ICON.2007.4444112

[9] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: solving the puzzle of enterprise 802.11 analysis," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 39–50, 2006. DOI 10.1145/1151659.1159920

[10] "MadWifi Project," 2010. Available online: http://madwifi-project.org

[11] "Aruba Networks Air Monitors," 2010. Available online: http://www.arubanetworks.com

[12] "OpenWrt," 2010. Available online: http://openwrt.org

[13] "Linksys WRT54G series," 2010. Available online: http://en.wikipedia.org/wiki/Linksys_WRT54G_series

[14] "Apple iPhone technical specification," 2010. Available online: http://www.apple.com/iphone/specs.html

[15] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984. DOI 10.1109/MC.1984.1659158

[16] "QuickLZ," 2010. Available online: http://www.quicklz.com

[17] "FastLZ," 2010. Available online: http://www.fastlz.org

[18] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[19] M. Robshaw, "The eSTREAM project," in *New Stream Cipher Designs: The eSTREAM Finalists*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–6. DOI 10.1007/978-3-540-68351-3_1

[20] P. Ekdahl and T. Johansson, "A new version of the stream cipher SNOW," in *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography (SAC)*. London, UK: Springer-Verlag, 2003, pp. 47–61.

[21] "libpcap with MMAP," 2010. Available online: http://public.lanl.gov/cpw/

[22] "Iperf," 2010. Available online: http://sourceforge.net/projects/iperf/

[23] "Community Resource for Archiving Wireless Data At Dartmouth (CRAWDAD)," 2010. Available online: http://www.crawdad.org/