Chapter 1

VIRTUALIZATION AND PROGRAMMING SUPPORT FOR VIDEO SENSOR NETWORKS WITH APPLICATION TO WIRELESS AND PHYSICAL SECURITY *

Azer Bestavros,[1] and Michael J. Ocean[1,2]

[1] *Computer Science Department*
Boston University, Boston, MA
best@cs.bu.edu

[2] *Computer Science Department*
Endicott College, Beverly, MA
mocean@endicott.edu

Abstract Network Security Systems are heavily anchored in the digital plane of "cyber space" and hence cannot be used effectively to derive the *physical identity* of an intruder in order to prevent further malicious wireless broadcasts (*i.e.*, escorting an intruder off the premises based on physical evidence). Embedded Sensor Networks (SNs) can be used to bridge the gap between digital and physical security planes, and thus can provide reciprocal benefit to security tasks on both planes. Toward that end, we present our experience integrating wireless networking security services into snBench (the Sensor Network workBench). snBench provides an extensible framework that enables the rapid development and automated deployment of SN applications on a shared, embedded sensing and actuation infrastructure. snBench's extensible architecture allows an engineer to quickly integrate new sensing and response capabilities into the snBench framework, while high-level languages, compilers and execution environments allow novice SN programmers to compose SN

*This research was supported in part by a number of NSF awards, including CISE/CSR Award #0720604, ENG/EFRI Award #0735974, CISE/CNS Awards #0524477, #0952145, CNS/NeTS Award #0520166, CNS/ITR Award #0205294, CISE/EIA RI Award #0202067, and CISE/CCF Award #0820138.

service logic, unaware of the lower-level components on which their services rely. Concrete examples are provided to illustrate the power and potential of Wireless Security Services that span both the physical and digital plane.

Keywords: Wireless Sensor Networks, Video Networks, Intrusion Detection, Physical Security, Software Environments.

1. Motivation

A variety of Wireless Intrusion Detection Systems (WIDS) have been created to address Wireless Network Security concerns. WIDS employ wireless probes/sensors to monitor the Media Access Control (MAC) frames transmitted on the wireless medium and identify misuse by observing either suspicious characteristics of individual frames (*e.g.*, exhibiting characteristics imprinted by standard hacking tools) or a particular pattern in a sequence of frames (*e.g.*, sequences in violation of protocol standards). Wireless misuse includes illegitimate users attempting to gain access to the network (intrusion), man-in-the-middle attacks (*e.g.*, luring legitimate users into communication with a rogue access point), and various Denial of Service (DoS) attacks [5] (*e.g.*, spoofing a legitimate wireless Access Point (AP) and sending a disauthenticate beacon to legitimate users).

Wireless intrusion is often dealt with using Layer-3 mechanisms (*e.g.*, content based packet filtering, IP address isolation), essentially ignoring the option of Layer-2 detection and prevention. Layer-3 IDSs are likely popular because there is far more data available at Layer-3, making it straightforward to respond to attacks, and because detection and response at Layer-3 is independent of the Layer-2 connection medium. On the other hand, Layer-3 *response* to Layer-2 wireless DoS attacks is limited given that attackers will likely utilize fictitious or spoofed MAC addresses and may not have an IP address to retaliate against. Ultimately the only way to respond to these types of attack is to utilize information derived from the wireless medium (*e.g.*, received signal strength) to reconstruct physical location toward the goal of preventing further wireless transmissions from that user [8].

Wireless Intrusion Detection Systems provide mechanisms to identify, detect and locate DoS attacks, yet these systems are generally limited to logging or email alert response mechanisms. Many works ultimately recommend dispatching administration personnel to further analyze and respond to a detected attack – a costly and impractical solution in many situations. Instead, once the physical area of an attack has been derived it is possible to utilize automated responses from a variety of actuation hardware, if available; *e.g.*, embedded pan-tilt-zoom video cameras to capture an image, wireless detectors on pan-tilt motors to pin-point a signal, programmable robots to triangulate signal, a common message display (virtual bulletin board) in the environment informing users why their service has been interrupted and who is responsible. Additionally, there would be a clear benefit from including other, non-network centric *inputs* to the Wireless Network Security System (*e.g.*, a MAC whitelist from Bluetooth/RFID tracking, analysis of security camera images or passcard logs).

Generally, attaining such cross-modal interaction within the context of a Network Intrusion Detection tool would require the generation of highly customized, package and deployment specific software (modules, scripts, *etc.*) that are, by their very nature, cumbersome to maintain. Indeed, such an approach is wrong headed. We observe that Wireless Network Security Services are specific, narrowly focused instantiations of an Embedded Sensor Network wherein sensory data includes the output of such monitoring tools. Rather than "hack" a Wireless Security System to include Sensor Network functionality, we advocate the inclusion of Wireless Security within a Sensor Network. Thinking differently about Network Security, the integration of new sensory data (*e.g.*, motion detection, face detection) and actuation responses expand Network Security beyond the digital plane and into the physical plane.

This chapter details the inclusion of wireless network monitoring devices in our Sensor Network infrastructure, snBench (Sensor Network Workbench) to achieve precisely these goals. snBench provides a high-level programmatic interface to the resources of a Sensor Network (SN) and thus the inclusion of wireless network sensors enables intrusion detection and response services to be written quickly and easily. snBench has been designed with extensibility and modularity as a central tenet and therefore the changes required to include these new sensing modalities are quite modest. Moreover, the framework's modular nature allows a user to swap in any improved emergent wireless surveillance tool or technology (be it algorithmic or a physical turn-key device) with nominal effort and such changes would be transparent to their dependent services. We submit that our programmable, adaptable SN framework is the ideal foundation on which to compose Wireless Network Security services and physical security services alike, providing reciprocal benefit to each. The example programs given provide some insight into the highly customized, cross-modal Wireless Security behaviors that are possible in this context.

2. Related Work

While many Network Intrusion Detection (Security) Systems exist (both commercial and open-source), we are presently unaware of any other work that leverages a programmable Sensor Network framework toward joint physical and Wireless Network Security, and thus believe we are unique in this regard. We present works that are related in three major thrusts; We distinguish between works that provide detection on a single wireless source (probe) as *Wireless Intrusion Detectors* (WIDs), those works that detect events across multiple detectors simultaneously as *Intrusion Detection Systems* (IDSs) and finally those that determine attack location as *Wireless Intrusion Detection Systems* (WIDSs). Although WIDSs contain a WID component, these works are not necessarily proper subsets of each other, as IDSs may not provide wireless detection. The distinction that a WIDS must determine attack location is sensible, considering that MAC addresses are easily spoofed [5] and that Layer-2 DoS attack response generally requires physical intervention [8].

Wireless Intrusion Detection: Kismet [9] is the *de facto* open-source Layer-2 Wireless Intrusion Detector. Kismet passively scans 802.11 channels for activity and will generate alert events when suspicious frames are detected (among other uses). A Kismet deployment may consist of three distinct components, (1) a Kismet Drone that passively captures the wireless frames from its local interface and sends them to (2) a Kismet Server that processes the frames from drones to detect either fingerprint or trend based suspicious activity and (3) an optional remote Kismet client that connects to the Server to receive notifications and render the results. By writing a custom client (using the published client protocol) Kismet may drive "external" wireless event notification. Kismet may be configured as an IDS by associating several drones with a single server process to build a single, central wireless event log. Kismet has also recently been updated to track which physical drone is responsible for an alert, enabling ad-hoc spatial intrusion tracking; thus, assuming a custom client that processes this data, such Kismet deployments would be considered Wireless Intrusion Detection *Systems* by our definition. Other notable WID tools have existed prior to Kismet but have been unmaintained in recent years.
Intrusion Detection Systems: As Kismet is to Layer-2 WID and IDS, Snort [11] is the *de facto* standard IDS for Layer-3 (IP traffic analysis). Snort is a mature IDS with a large user base and comprehensive set of detection rules for detecting malicious content in IP packets for a wide

range of attacks. Snort also offers very basic response mechanisms (*e.g.*, logging or email alert mechanisms) and projects (*e.g.*, Barnyard) that claim to enable the creation and use of custom output plug-ins. As Snort is aimed at Layer-3, it offers no support for wireless-specific events; plans to integrate wireless frame capture appear to have been abandoned.

In many ways, our vision is similar to that of modular (or so-called "Hybrid") IDSs (*e.g.*, [14], [13]). These systems are designed to allow various Intrusion Detection Software packages to be integrated as "sensors" in the IDS. This modular approach is similar in spirit to the cross tool integration that we hope to provide to the Network Security community, yet these works are narrowly focused on issues of traditional Network Security. Our work enables sense and respond programs that manipulate both network and physical sensory data (*e.g.*, image processing on embedded video cameras) in a manner that would be impossible on these platforms without significant changes.

Wireless Intrusion Detection Systems: Many approaches to derive location from multiple sensors' Signal Strength Information (SSI) of RF transmissions have been undertaken, including addressing issues of transmission reflection, diffraction and interference (*e.g.*, [4], [15]). The WIDS architecture detailed in [2] provides detailed analysis of specific directional antennas (as opposed to the typical, omni-directional antennas) to form a sweeping perimeter around an access point and is able to accurately pinpoint wireless intruders. Not only would our work be compatible with the use of sweeping directional antenna, snBench could likely direct the servos that control antenna movement explicitly within the security logic (easing future changes).

Finally, commercial offerings provide turn-key detection and response systems for corporate wireless networks (*e.g.*, [3]). Responses to wireless attack detection in these systems are more proactive (*e.g.*, disauthenticating malicious users from the network), yet they do not provide integration with third-party tools or offer a programming interface to adjust the sense and respond behavior. Commercial sense and respond WIDSs lack the extensibility required to enable cross-modal monitoring (*e.g.*, utilizing video frames).

3. snBench Overview

To orient the reader to the platform to ease further discussion, in this section we briefly highlight the salient features of snBench. The vision, goals and high-level overview of the snBench infrastructure have been reported elsewhere [6] and implementation details may be found in [10].

snBench consists of programming support and a runtime infrastructure for Sensor Networks comprised of heterogeneous sensing and computing elements that are physically embedded into a shared environment. We refer to such a physical space with an embedded SN as a Sensorium. The snBench framework allows Sensorium users to easily program, deploy, and monitor the services that run in this space while insulating the user from the complexity of the physical resources therein. We liken the support that snBench extends to a Sensor Network to the support that higher-level languages and operating systems provide to traditional, single machine environments (language safety, APIs, virtualization of resources, scheduling, resource management, *etc*). snBench is designed such that new hardware and software capabilities may be painlessly folded into the infrastructure by its advanced users and those new capabilities easily leveraged by its novice users.

snBench provides a high-level programming language with which to specify programs (services) that are submitted to the resource management component which in turn disseminates program fragments to the run-time infrastructure for execution. At the lowest level, each sensing and/or computing element hosts a Sensor eXecution Environment (SXE) that abstracts away specific details of the host and attached sensory hardware. SXEs are assigned tasks by the resource management components of snBench; the Sensorium Service Dispatcher and Sensorium Resource Manager in tandem monitor SN resources, schedule (link) and deploy (bind) tasks on to available SXEs.

The Virtual Instruction Set Architecture of snBench is the Sensorium Task Execution Plan (STEP), a tasking-language used to describe complete programs and fragments alike. A STEP program is a graph of an SN program's data-flow and computational dependency, with the nodes of a STEP graph representing the atomic computation and sensing operations and edges representing data flow. In execution, demand for evaluation is pushed down from the root of the graph to the leaves, and values percolate up from the leaves back to the root. STEP nodes describe data, control flow (*e.g.*, repetition, branching) and computation operations that we refer to as STEP Opcodes, and the SXE maintains implementations of the Opcodes with which it may be tasked.

Opcodes do not directly manipulate sensors, but rather manipulate snBench typed data. Specific details of the sensor hardware of the SXE are abstracted away by a SensorHandler module that is capable of communicating with and reformatting the data from a specific sensor to produce to snBench typed data; support for new sensor device types require the addition of new SensorHandler modules[1]. In snBench there is a distinction between a SN Service Developer who uses high-level programming languages to compose Services by gluing together Opcodes and sensors (generally without regard for how the Opcodes are actually implemented beyond their type signature) and the snBench "engineers" who are responsible for expanding the Opcode and SensorHandler libraries to enable new functionalities.

4. Enabling Wireless Monitoring

snBench is extensible by design insofar as support for new sensing devices may be added to the Sensor eXecution Environment (SXE) by providing implementations of two relatively small interfaces; a *SensorHandler* translates snBench requests to interact with a specific device and a *SensorDetector* module must provide a facility to detect new devices of this type and inspect their state. The SensorHandler is akin to a device driver, abstracting away the specific idiosyncrasies of the particular device's interface and enabling the device to be accessed by higher-level programming constructs. As far as the snBench framework is concerned, the abstracted device becomes just another managed input device/event generator only different from a video camera or motion sensor insofar as the datatype of its output.

To enable wireless network security service composition on snBench, two new sensors and a new actuator were added; the WifiAlertSensor reports wireless alert detection events, the WifiActivitySensor reports MAC addresses and Received Signal Strength Indication (RSSI) for any passively observed wireless activity, and the WifiResponder actuator sends a disauthenticate flood to a particular MAC address. Rather than implement wireless Layer-2 tools from scratch, we opted to leverage several existing open-source software packages.

WifiAlertSensor: The WifiAlertSensor is a SensorHandler implementation that leverages the Kismet [9] wireless intrusion detector via a self-contained customized Kismet client. The Java based WifiAlertSensor class is hosted by a "non-lightweight" SXE and translates the proprietary Kismet client-server protocol into structured, typed snBench objects (tagged XML) that encapsulate notifications from the Kismet server. The decision to use Kismet stems from its passive scanning ability, wide range of hardware support, and modular design (described in Section 2). While the decision to use this package in particular may be debated, the inclusion of any another functionally-equivalent Wireless Intrusion Detector would be equally straightforward.

A Kismet client may request to receive several types of Kismet messages from a Kismet server/drone pair (client traffic, AP detection, suspicious activity alerts, *etc.*). In the case of the Alert Sensor, the client requests notification of all wireless alerts supported by the current stable build of Kismet. Whenever the Kismet server detects an alert condition from its corresponding drone's data feed, an alert is sent to the WifiAlertSensor client which translates and buffers the alert message. In addition to translating the Kismet protocol, the WifiAlertSensor adds additional fields to the alert message: a local timestamp to measure buffer service delay, a sensor source to identify the physical sensor (drone) that produced the message, and a severity field that indicates the relative threat of the particular attack.

The WifiAlertSensor's message buffer is configurable in length (where length is measured in either size or time) and alert messages are retrieved from the buffer by Opcodes requesting data from this sensor. Implementation of the retrieval Opcode may impose a blocking or non-blocking semantic, as needed. In our experimentation we implemented a single alert-centric Opcode, `sxe.core.wifi.get`, that performs a non-blocking read from the Alert Sensor's buffer to populate and return a WifiAlert. The WifiAlert data-type is a subtype of snStruct, with tagged fields corresponding to the fields populated by the WifiAlertSensor and thus accessing the data within a WifiAlert reuses the existing snStruct manipulation opcodes. A Service Developer retrieves WifiAlerts via the high-level function `DetectWifiAlert()` that is compiled into a call to the Opcode `sxe.core.wifi.get` with a WifiAlertSensor (or set of sensors) as a parameter. High-level service logic examples are given in Section 7.

WifiActivitySensor: The Activity sensor provides data regarding wireless transmissions that have been detected by a passive, promiscuous-mode wireless sensor. In particular, we are interested in the MAC address of a transmission, the observed signal strength (RSSI) and the mode of the transmission (*i.e.*, Access Points, Clients, Ad-hoc participants). While determining physical location from RSSI is imperfect (as RSSI readings themselves may not be entirely accurate depending on the driver implementation and other physical factors), the use of RSSI readings can better estimate the physical location of a MAC address beyond the simple cell-of-origin. WifiActivitySensor maintains a hashtable of the detected wireless activity (keyed by MAC address), which can be used either to report new/updated wireless activity (similar to the Alert Sensor) or to query the activity log to find information about a particular MAC address. Like the Alert Sensor, the activity sensor also communicates with a remote sensor "server" process responsible for gathering data.

As the Kismet drone/server cannot retrieve the RSSI on all hardware platform, two different physical implementations for the activity sensor server are supported. For Kismet's RSSI-supported hardware, the client, which is derived from the WifiAlertSensor implementation, requests and parses NETWORK and CLIENT messages from the Kismet server rather than ALERT messages. For the OpenWRT platform, a custom monitoring program sends ioctl's to the wireless device to put the device in passive monitor mode, accept frames, and retrieve data from the frames and device (including the RSSI). This program is based on code from the open source WiViz [12] package for OpenWRT, which contains the ioctl codes needed to achieve the proper device state and interaction. Like the Kismet server, this program provides notifications of activity messages which are received and hashed by the WifiActivitySensor.

The high-level Opcode `DetectWifiActivity()` is compiled into `sxe.core-.wifi.get` with a WifiActivitySensor as a parameter and blocks until a new activity message is available from that sensor. In addition `QueryWifiActivity()` (compiled into `sxe.core.wifi.find`) searches the WifiActivitySensor's hash table for the latest reading associated with the specified MAC address. As with the WifiAlertSensor, returned data is an snStruct derivative.

WifiResponder: In addition to the wireless network sensing described above, the Layer-2 wireless actuator (*i.e.*, output device) WifiResponder may be used as a retaliatory action against a detected attacker. The WifiResponder invokes a script on a trusted (whitelisted) device running Linux with a compatible 802.11 interface and the airreplay-ng [7] tool. The Opcode `APDeauth()` takes as arguments a WifiResponder that will send a flood of deauthenticate messages to a particular MAC address (the second argument) from a particular MAC address (the third argument).[2] An actuator is nearly identical to a Sensor in its implementation within snBench. The Handler for WifiResponder invokes the remote common gateway interface (CGI) script to initiate the deauthenticate "attack" against the specified host.

5. Deployment Environment

Our test-bed deployment contains several OpenWRT[1] Linux enabled Linksys WRT54GL Access Points (APs), each with the kismet-drone, airreplay-ng, and signal strength monitor packages installed. The APs are configured to use their wireless interface in client mode, and are connected to our gigabit research LAN by its 100Mbit Ethernet port.

To support the WifiAlertSensor, each of the APs run a Kismet drone process, while the Kismet server process runs on the same host as the SXE. Although the Kismet server process could also be run directly on the AP, the RAM and CPU limitations of these devices lead to a less responsive system in that scenario. As the Kismet server did not distinguish the results from different Kismet drones at the time of our experiments, one Kismet server process was required per drone, and each WifiAlertSensor connects to a unique Kismet server process thus allowing snBench to distinguish which drone generated a wireless event. Running one Kismet server per drone also carries the advantage of minimizing the impact of a Kismet server process hanging, or failing to process updates from its drones (admittedly a fairly uncommon occurrence).

In our tests of the WifiAlertSensor we were able to simulate and detect all relevant attacks detected by Kismet and were unable to measure any significant induced delay on event detection in the snBench infrastructure. Analysis confirmed the expectation that the amount of time a single Kismet message spent in the Sensor buffer was directly related to the computation load on the SXE host and the alert generation rate. In general the observed buffer service delay oscillated between zero and 15ms per alert under moderate load with unrealistically high message flooding arrival rates (in practice, Kismet can and will throttle alert notification rates, however this was disabled for our performance tests). Under heavy load conditions with alert message flooding, we experienced queuing delay as long as 300ms. This gives us a good indication as to the maximum acceptable workload for an individual SXE before it is no longer a viable host for wireless sensing tasks. Ultimately any response detection under one second is reasonable as it is unlikely that the attacker would, say, flee the premises (or video frame) within that amount of time.

6. Service Programming Primer

To understand the Wireless Security Services examples it is important to understand the key concepts and unique constructs of snBench programming.[3] The Sensorium Task Execution Plan (STEP) language has a functional-style, high-level sibling called SNAFU (Sensor Network Applications as Functions). SNAFU serves as a readable, accessible language that is compiled into the graph-centric STEP for execution. Broadly speaking, functions in SNAFU correspond to the computational nodes of a STEP graph while terminals represent nodes that convey sensors, actuators, and constant values.

SNAFU provides symbolic assignment and function definition, however it forbids explicit recursion by reference. Instead SNAFU provides iteration constructs called triggers. A trigger takes two arguments: a predicate and a response clause. The predicate is repeatedly evaluated until it evaluates to true, at which point the response clause is evaluated and returned as the result of the trigger expression. For example, consider the expression `Trigger(P,Q)` in which `P` is the detection of an AP intrusion and `Q` is an expression that shuts down the AP. The `While-Trigger(P,Q)` is similar to the previous trigger, except that it evaluates `Q` every time `P` evaluates to true and when `P` eventually evaluates to false returns the last value of `Q` (or *NIL* if `P` was initially false).

Persistent triggers extend the basic triggers in that they return a stream of values over their persistent evaluation. A `LevelTrigger` evaluates the predicate `P` indefinitely (or for some specified length of time or conditional termination) and evaluates and returns a value of `Q` every time `P` evaluates to true. In practice `P` may be the detection of a particular MAC address being used in the network and `Q` is the recording of an image at the detected locale. An `EdgeTrigger` continually evaluates the predicate, but will only evaluate and return the clause `Q` whenever the predicate `P` transitions to be true (*i.e.*, on the edge of the signal `P`). Consider, if the expression `P` represents detection of two deauthenticate beacons (indicating the start of a deauthenticate flood) and `Q` is an SMS pager alert, we do not want to generate a separate notification for every consecutive deauthenticate beacon for the duration of the flood.

SNAFU also allows a programmer to refer to an expression by symbolic reference (*e.g.*, `let X = Y in Z`, wherein `X` stands for the complete expression `Y` in the expression `Z`) or refer to a computational result by symbolic reference (*e.g.*, `let_const X = Y in Z`, wherein `X` stands in for the result of the expression `Y` in the expression `Z`). Finally the trigger construct begs for the creation of a unique reference that allows the symbol to be recomputed once per iteration of the trigger. The "`let_once`" binding (*e.g.*, `let_once X = Y in Z`) provides exactly that facility, ensuring the expression `Y` is evaluated once per iteration of the trigger (`Z`) at the first occurrence of the symbol `X`, while all latter instances of the symbol `X` in the same iteration of `Z` are evaluated by reference to the previous evaluation.

7. Wireless Security Services

An example SNAFU program that provides simple logging is given in Program 1. A `level_trigger` is used to assign an event handler to the detection of a high severity wireless alert. The `storage.append` Opcode modifies a named storage entity (*i.e.*, table) by inserting a data object and its corresponding unique key. The storage table is keyed by timestamp and includes entries for each detected violation containing the recorded MAC address, the sensor from which the alert was detected, and the type of alert. Unlike the logging provided by Kismet as an IDS, this service records which sensor has detected the event and is backed by an SQL server. The logged data is available programmatically via storage access Opcodes or direct SQL queries, or through a standard web browser via the SXE host's web service that performs XSL translations to render the local data storage.

SNAFU Program 1 Add to a central log on detection of a wireless alert.

```
let_once ALERT = DetectWifiAlert(sensor(WifiAlert,ALL)) in
   level_trigger(
      equals(ALERT."SEVERITY","HIGH"),
      storageappend("ALERTLOG",
            concat(ALERT."TIMESTAMP",ALERT."SOURCE"),ALERT))
```

This sample SNAFU program could easily be extended to establish a log of all observed wireless activity (not just attacks) by adjusting the predicate of the trigger from `DetectWifiAlert` to `DetectWifiActivity` and removing the severity check. Another simple example is given in Program 2, which automatically emails an administrator when a specific wireless attack is detected.

The previous examples are essentially the status quo for a response to the detection of a breach in a Wireless Network – an entry into a log file or an email alert. The advantage of employing the snBench in the wireless security domain is the wider range of responses possible. Nominally, the email operation in Program 2 could be replaced with any number of response mechanisms including sending an explicit deauthorization to the detected MAC address[4] using the WifiResponder and APDeauth opcode described in Section 4. Instead, we explore the unique cross section of the network plane (*e.g.*, wireless data frames) with the physical plane (*e.g.*, signal strength and signal loss of signal over distances). For example, an embedded, cross-modal Sensor Network such as the Sensorium can utilize both wireless network sensors (*i.e.*, network plane) and a pan-tilt-zoom video camera network (*i.e.*, physical plane) to catch an image of the attacker "in the act."

SNAFU Program 2 E-mail an admin when a specific wireless alert is detected.

```
let_once ALERT = DetectWifiAlert(sensor(WifiAlert,ALL)) in
    level_trigger( equals(ALERT."TYPE","DEAUTHFLOOD"),
            email("mocean@cs.bu.edu",
                concat("$NOW$",
                    ": Deauth flood detected from MAC ",
                    ALERT."MAC", " at time ", ALERT."TIMESTAMP",
                    " by sensor ", ALERT."SOURCE")))
```

Any user detected engaged in wireless network intrusion is clearly within a bounded distance from the detecting sensor. This coarse, cell-of-origin based physical location of wireless users is available, imprinted in all wireless data returned from the WifiSensors (determined by which sensor has detected the user). A very simple wireless cell-of-origin location example is specified in Program 3. The program's content is very similar to the previous examples and introduces some pan-tilt-zoom sensor (PTZCamera) specific opcodes, the function of which should be clear from context. This sample streams images of a region where an attack has been detected. The location estimation is explicit in the service logic, selecting an image from the camera that best covers the physical space within the signal coverage region of the relevant Wifi sensor, which (in this example) requires some knowledge of the specific physical layout of the sensor deployment. The `case` expression takes the same syntax as in StandardML and is used for readability as syntactic sugar (*i.e.*, a macro) for nested conditionals. Connecting this program fragment to either of the previous examples would log or email images that correspond to the attack location.

Alternatively user location reconstruction could be implemented within an Opcode, resulting in IDS logic that is agnostic to the particular location resolution mechanism used. Such an approach makes sense if the deployment environment already contains a wireless location infrastructure (*e.g.*, network appliance, all knowing oracle) that could be accessed from an Opcode call. An example of this approach is given in Program 4. `WifiLocateMac` encapsulates the physical location of MAC addresses and `PTZLocate` determines the best PTZ Camera (and corresponding angle) to capture an image of that location. The implementation of `WifiLocateMac` is functionally similar to `BestPTZForViewOf` in the example in Program 3, yet uses a received signal strength from multiple sensors to estimate the target's location between the sensors.

SNAFU Program 3 Whenever a wireless alert is detected, pan a PTZ camera to that region and return its image.

```
def BestPTZForViewOf(alert) = case APName(alert.SOURCE) of
    "CS Grad Lab West" => List(45,0,0,sensor(PTZCamera,"PTZ1"),
  | "CS Grad Lab East" => List(15,0,0,sensor(PTZCamera,"PTZ1"),
  | "CS Grad Lab Lounge" => List(0,0,0,sensor(PTZCamera,"PTZ3"),
  | "CS UGrad Lab" => List(0,0,0,sensor(PTZCamera,"PTZ4")

let_each ALERTSENSORS = sensor(WifiAlert,"ALL") in
  let_once ALERT = DetectWifiAlert(ALERTSENSORS) in
    level_trigger( not(isNull(ALERT)),
      PTZSnapshot(BestPTZForViewOf(ALERT)))
```

SNAFU Program 4 Equivalent to Program 3, but uses “black-box” opcodes.

```
let_each ACTSENSORS = sensor(WifiActivity,ALL) in
   let_each PTZSENSORS = sensor(ptz_image,ALL) in
      let_once ALERT = DetectWifiAlert(sensor(WifiAlert,ALL))
         in level_trigger( not(isNull(ALERT)), PTZSnapshot(
            PTZLocate(QueryWifiAlert(ALERT."MAC",ACTSENSORS)),
            PTZSENSORS)))
```

snBench not only eases the composition of such alert services, it also eases deployment by automating the re-use of existing computation/deployments to improve resource utility. All the examples given thus far share the same predicate logic and could share a single instantiation of that portion of the logic.

8. Future Work and Conclusions

Wireless Access Lists from Physical Data: With this infrastructure in place, programs may use information detected on the physical plane to (re-)configure the wireless network. For example, an embedded camera network and face detection Opcodes can be used to detect the identities of individuals entering or leaving as a trigger to enable the detected user’s wireless MAC address for service in a physical space. Put simply, when we see Jane enter the lab we want to enable Jane’s MAC address (added to the whitelist), and disable her MAC address when she leaves the lab. A dynamic whitelist would make it more difficult for a malicious user to abuse unused, authorized wireless MAC addresses for great lengths of time. Modification of the WLAN’s access control list in this way assumes the presence of a MAC whitelist; such an implementation is straightforward on OpenWRT enabled APs, using a CGI script to modify the device's configuration. In addition, other physical sensors could be used in tandem with face detection as the trigger predicate in this expression; *e.g.*, biometric sensors, magnetic card or RFID readers.

snBench as a Complete, Turn-Key Network Security Solution: The Network Security provided by snBench need not be limited to Layer-2 alone. Integrating Layer-3 detection (*e.g.*, Snort) as a sensor would enable the detection of misuse from IP contents that could be used to drive isolation or removal responses at Layer-2. Including port scanning or other fingerprinting tools as sensors could increase the accuracy of user identification thus further open the possibilities for more "severe" automated response.

Ideally we could imagine migrating our own campus IT departments to use snBench as their Network Security and Intrusion suite, a transition that could be eased by the development of a declarative/rule-oriented domain-specific language (and corresponding STEP compiler) that is similar to existing network rule specification languages. Finally, our work on lightweight Sensor eXecution Environments for embedded devices could be used to run the SXE directly on OpenWRT enabled APs to provide snBench as a turn-key solution for Wireless Network Security services.

In Conclusion: Network Security (specifically, wireless security) is not a problem that exists in a vacuum detached from the physical space in which the network is deployed. We promote an approach that unifies physical site surveillance and network security under the umbrella of snBench — our general purpose sensing infrastructure. In that regard, we have demonstrated how snBench enables the rapid development and deployment of cross-modal security services. We have shown that with snBench (1) detection of wireless anomalies can be correlated with other sensory inputs providing reciprocal benefit to merging security on the physical and cyber planes, (2) detection and response services may be easily composed and modified without technical knowledge of the specific protocols or implementations of the underlying sensory tools, and (3) adding new intrusion detection tools as input or other devices for response is straightforward given snBench's modular architecture. The illustrative example programs provided include the status quo (simple logging and email alerts) and hint at where we may go from here, in an attempt to spark the reader's imagination to consider what sensors, actuators and new hybrid services may be enabled by the snBench platform.

Notes

1. SXEs can retrieve Opcode implementations at run-time; however, support for loading new sensing devices at run-time is not currently supported. Such functionality is not difficult to support, and is analogous to dynamically loading device drivers to support new hardware.
2. Readers may readily note that this opcode is a loaded weapon and may gasp or recoil in horror. In fact, this is not the first Opcode that requires special user privileges to ensure correct use.
3. We refer the reader to [10] for a more thorough treatment of the SNAFU language and its evaluation.
4. A MAC address is far from the best way to uniquely identify an attacker as the attacker will likely use a fictitious MAC address or worse, clone a legitimate user's MAC during an attack.

References

[1] *OpenWRT Project Homepage*, http://openwrt.org/.

[2] Frank Adelstein, Prasanth Alla, Rob Joyce, and Golden G. Richard III, *Physically locating wireless intruders*, ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2 (Washington, DC, USA), IEEE Computer Society, 2004, p. 482.

[3] AirDefense, Inc., *AirDefense Enterprise Product Homepage*, http://www.airdefense.net/products/enterprise.php.

[4] Paramvir Bahl and Venkata N. Padmanabhan, *RADAR: An in-building RF-based user location and tracking system*, INFOCOM (2), 2000, pp. 775–784.

[5] John Bellardo and Stefan Savage, *802.11 denial-of-service attacks: real vulnerabilities and practical solutions*, SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium (Berkeley, CA, USA), USENIX Association, 2003, pp. 2–2.

[6] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Michael Ocean, *SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications*, IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets), October, 2005.

[7] Christophe Devine, *Aircrack-ng homepage*, http://www.aircrack-ng.org/.

[8] Jamil Farshchi, *Wireless intrusion detection systems*, http://www.securityfocus.com/infocus/1742, 2003-11-05.

[9] Mike Kershaw, *Kismet (version 2007-01-r1b)*, http://www.kismetwireless.net/documentation.shtml.

[10] Michael J. Ocean, Azer Bestavros, and Assaf J. Kfoury, *SNBENCH: Programming and Virtualization Framework for Distributed Multitasking Sensor Networks*, VEE '06: Proceedings of the 2nd international conference on Virtual execution environments (New York, NY, USA), ACM Press, 2006, pp. 89–99.

[11] Martin Roesch, *Snort - Lightweight Intrusion Detection for Networks*, LISA '99: Proceedings of the 13th USENIX conference on System administration (Berkeley, CA, USA), USENIX Association, 1999, pp. 229–238.

[12] Nathan True, *Wi-viz: Wireless Network Environment Visualization*, http://devices.natetrue.com/wiviz/.

[13] Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer, *Designing and implementing a family of intrusion detection systems*, SIGSOFT Softw. Eng. Notes 28 (2003), no. 5, 88–97.

[14] Yoann Vandoorselaere, et. el., *Prelude Hybrid IDS*, http://www.prelude-ids.org/.

[15] Moustafa Youssef, Ashok Agrawala, and Udaya Shankar, *WLAN Location Determination via Clustering and Probability Distributions*, March 2003.