Distributed WPA Cracking

Rodney Beede
University of Colorado
Boulder, CO 80309-0430
rodney.beede@colorado.edu

Ryan Kroiss
University of Colorado
Boulder, CO 80309-0430
ryan.kroiss@colorado.edu

Arpit Sud
University of Colorado
Boulder, CO 80309-0430
arpit.sud@colorado.edu

ABSTRACT
In this paper, we describe a distributed system we developed for doing lookups in a rainbow table for passwords to WPA-PSK 1/2 wireless encrypted networks. Our motivation for developing such a system came from the abundance of distributed systems for generating rainbow tables but a lack of any to do the lookup.

A rainbow table for even just a million passwords can be gigabytes in size. The problem we address is how a distributed system could be used to provide fast lookup of matching passwords in a table for some given captured wireless network data. We chose to focus on WPA versus other wireless encryption techniques (e.g. WEP) because it offers the best encryption that cannot currently be feasibly defeated via brute force methods.

Our approach involved writing new code to handle user submitted jobs of wireless data packets and modifying existing code from a project known as coWPAtty [5] which handled the actual table lookup. We used a cluster of 9 nodes running on virtual machines to handle the work load and compared the performance of our system to the original serial coWPAtty implementation.

Our results show a performance boost of a factor of 8 for our distributed system versus the serial coWPAtty. Testing data showed that the original serial coWPAtty code could return a single result within several seconds while our system could return results in under 1 second. While not a seemingly big difference, the scalability of our system provides much more usability especially if used as a web service.

We concluded that our results show that the importance of using a strong password for wireless networks is still very important. WPA is an improvement over previous encryption ciphers used in wireless networks, but it is still susceptible to weak passwords chosen by the end user. The most important outcome from our research shows that using large rainbow tables of pre-computed passwords can provide an easy, fast, and scalable method for finding weak passwords in encrypted wireless networks.

Categories and Subject Descriptors
C.2.1 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design– *Wireless communication;* K.6.5 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Security and Protection– *Unauthorized access (e.g., hacking, phreaking);*

General Terms
Security

Keywords
WPA, rainbow table, dictionary attack

1. INTRODUCTION
The use of wireless networks based on the 802.11 standards (also known as Wi-Fi) has become much more common place in a typical household. Early advents of these networks provided security to limit access and protect sensitive data with a protocol standard known as Wired Equivalent Privacy (WEP). Research and analysis into the cipher algorithms used in this protocol led to discovered weaknesses that could easily be exploited by a single computer with modest hardware using brute force. The deficiencies of this protocol lead to the development of a new one known as Wi-Fi Protected Access (WPA) [3] which was subsequently enhanced with WPA2. WPA1/2 can work in 2 modes:

- WPA-PSK or Pre-Shared Key also known as WPA-Personal
- WPA-Enterprise which is more secure but requires RADIUS authentication server

WPA-PSK utilizes TKIP for encryption and WPA2-PSK uses CCMP (which is based on AES). Thus, the newer protocols which utilize much more advanced and stronger encryption ciphers have made the possibility of using brute force attempts not feasible [4]. With the advent of these new security measures ways of attacking them have evolved as well.

One technique for deciphering encrypted data or accessing protected resources is to simply try a large number of different passwords in the hope to get one correct. The nature of wireless networks enables one to capture encrypted packets as they pass through the air and store them to disk in their encrypted form. Part of the wireless encryption protocol involves handshaking at the initial connection in order to authenticate the client to a wireless access point. This handshake includes some initialization vectors (IV's) that are used to allow the client to encrypt the password they will send for authentication. There are multiple types of WPA encryption some which use client certificates and server certificates for encryption of the data while others use simple passwords (WPA-PSK) that the user provides upon establishing the connection. The password has a minimum of eight characters and a maximum of 63 and is case sensitive. The encryption key is derived from this password and the case sensitive name of the wireless network (SSID). Attempting to brute force this key space would also prove to be computationally infeasible. A common alternative is to try a large dictionary of common passwords. This dictionary can also include slight

permutations on words such as appending numbers to the end in order to catch common passwords used by users.

The dictionaries used could contain millions of words which would require some significant amount of time to process for each wireless network someone wished to attack. A complimentary technique is to generate a large rainbow table of keys based on the dictionary and several common wireless network names. This would then enable an attacker to simply capture some wireless network authentication handshake data and do a lookup in the pre-computed rainbow table for a match. Such a tool already exists and is known as coWPAtty.

coWPAtty is a serial program that can generate a rainbow table with hash values for matching keys based on wireless networks names (SSIDs) and WPA-PSK passwords. The generation is done in a serial manner and written to a file that can later be used in conjunction with captured wireless data. All operations in the original coWPAtty code are done in a serial manner on one machine.

For our project we decided to create a distributed system that could perform this rainbow table lookup among a cluster of nodes in order to increase performance. We chose to not generate the rainbow tables as others have already done so [2]. Instead we chose to use an existing rainbow table that contained pre-computed keys for 1,000 wireless network names (SSID's) and divide it across multiple machines for doing lookup queries.

We developed a system with a single master node and multiple worker nodes that handled the job submission and work. The master node was written from scratch as a Java web application and is responsible for queuing jobs, sending them to the workers, and reading back the results. The worker nodes were created by modifying the original coWPAtty code to function in a distributed manner and are described later.

Our testing methodology consists of capturing wireless data from our own personal networks and submitting them to the distributed system. We compared the times for our system to find the correct solution to that of the original coWPAtty serial code running on a single node. We document our findings in the results and conclusions portion of this paper.

All the project code and testing data can be found on the project's website at http://code.google.com/p/distributed-wpa-cracking/

It should be emphasized that all testing data was gathered from the authors' own personal wireless networks. This software should not be used to infringe on the privacy rights of others.

2. ARCHITECTURE

2.1 Overview

The system is comprised of a master node which is responsible for coordination of work among a cluster of worker nodes. The worker nodes each hold a portion of the rainbow table in memory and listen for requests over TCP from the master node.

A network file system (NFS) is used as a shared location for the job input data and output results. In addition it serves as a central point for the binary code that will be run on both the master and the worker nodes.

The wireless network is not connected to the master or worker nodes nor must it even be anywhere in proximity to them. The end user captures wireless data from the wireless network using their computer and saves the result to disk. At a later time, after sufficient data has been captured, the user can then upload the captured data to the master node for processing.

Communication from the wireless network to the user's computer is done via standard 802.11 wireless protocols. The data capture can be done with already existing tools such as Aircrack-ng [1]. Communication from the user to the master is accomplished using the HTTPS protocol over a secure TCP socket. Communication internally occurs between the master node to a worker node. Worker nodes do not communicate with each other. The master node uses TCP connections with the SSH protocol for remotely starting the worker node binaries and a custom propriety protocol for sending running worker nodes jobs and checking their status.

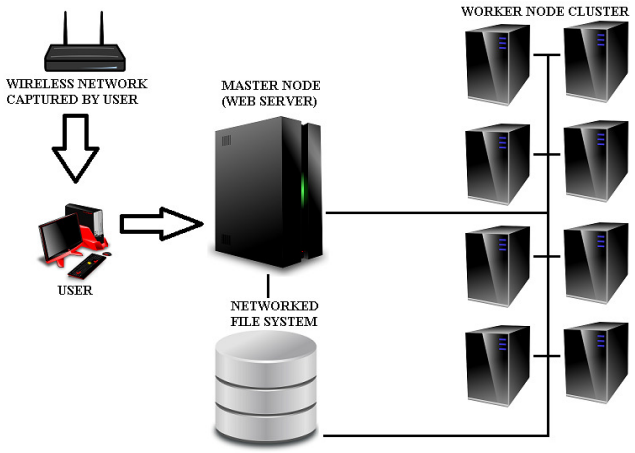

Figure 1. Architecture Overview

2.2 Network File System

In order to be able to share input data (the wireless capture file) with the worker nodes a networked file system was setup with common access between all worker nodes and the master node. This choice was made because it was a simpler configuration versus having the master transmit a copy over TCP to each worker node and have them store a temporary copy on their local disks. The master node was used to host the network file system, but a separate device could have been used as well.

When the user uploads a job request to the master it stores the uploaded capture file to a newly created folder on the network file system. The folder is given a name that matches the newly generated job id. This folder is visible to the worker nodes and will be used by them to later read the wireless capture file and also for storing the output results.

Common binaries for all the workers such as the actual worker binary executable are stored here as well. This allows for updated code to be published in one place and be applied to all workers at once. A simple restart of the worker process on each worker node will then load the new updates to any code.

The large rainbow table data, however, is not stored on the network file system, but instead a read-only copy exists on each node on local disks. Because gigabytes of data must be read from the rainbow table by each node at startup into memory having copies on local disk allow for shorter startup times since disk contention and network congestion are avoided. While this would add some burden to adding updates to the rainbow table this isn't done frequently and just requires more time to copy any new data

to each node's local disk. In addition our project doesn't generate new rainbow tables so no updates were really necessary.

2.3 Master Node

The master node is a Java web application written from scratch that is responsible for accepting and queuing user requests along with tracking the status of the worker nodes.

The master node web application provides a user interface as depicted in figure 2 that provides the user with the following:

- A HTML form for uploading a new job
- View of the jobs in the queue and completed jobs
- Status list for the worker nodes
- Controls for starting / killing the worker nodes
- View of the web application log

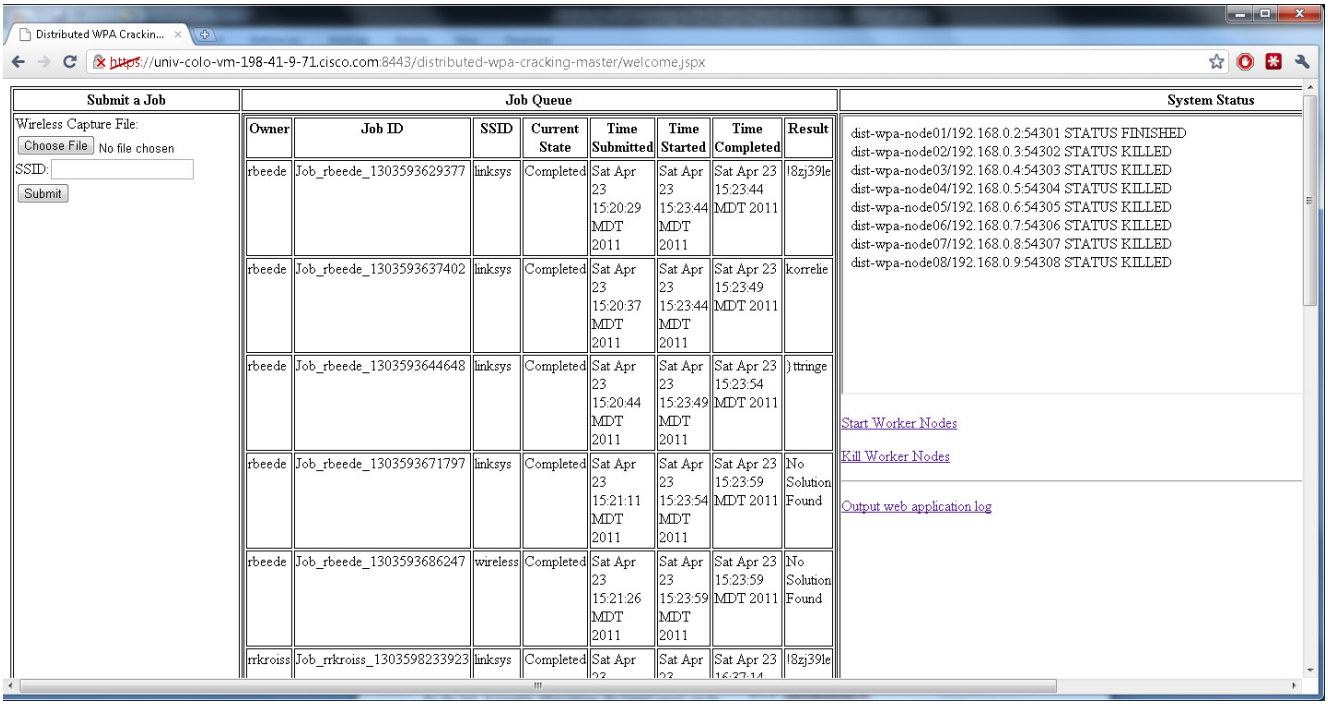

Figure 2. Master Node Web Application Interface

2.3.1 Configuration

All configuration is managed via an XML file on the master node that is loaded at the startup of the master node web application. It specifies the following:

1. The base directory pathname for where job data should be stored
    a. A network file system share that is accessible to both the master and worker nodes
2. The hostnames and port addresses for the worker nodes
3. The directory to the rainbow table for local use
4. Command templates for starting the worker nodes
5. Command for killing worker nodes
6. SSH credentials for accessing worker nodes

The template approach for the start command for worker nodes allows for the master to fill in necessary details such as directory paths and rainbow table offsets when starting a worker node. Details will be discussed in the section on worker nodes.

2.3.2 Job Submission

The web site itself is protected with a username and password along with HTTPS encryption. When a user submits a job via the interface the username, as reported by the web server container (Apache Tomcat), will be used for the name of the job owner and as part of the job id.

The user simply uploads a wireless network data capture and specifies the SSID of the desired network (a data capture could have more than one network in it so this must be specified). The upload and job submission could also be accomplished through automated means of a script by using the standard HTTP protocol as well.

The master node web application receives the request, generates a unique job id for it, creates the necessary directories on the network file system, and adds the job to the queue.

2.3.3 Job Queue

The queue is served in first-in first-out (FIFO) order and a history of all jobs is retained until the master node web application is restarted.

A job will not be started until all workers report that they are in a ready state and not currently busy with another job. Jobs are run one at a time.

Information such as the start and end time along with the solution found, if any, is listed in the interface. All users can view any job in the queue.

When the master is told that one worker has found the solution it will send a kill job signal to all other workers informing them to stop in case they haven't yet.

2.3.4 System Status

The master periodically queries each worker to get their current state. The list of workers and their state is listed to the user.

Details on the various state types are listed in the Worker Node section later.

2.3.5 Logging

System logging is performed on both the master node and the worker nodes. This was useful not only for debugging purposes but also for gathering more fine-grained results.

The worker nodes log messages to the network mounted file system so that they can easily be read from the master node. Each log outputs a millisecond scale time stamp along with the log message. These timestamps were used when collecting our experimental data. The web application did not provide sufficient resolution for our needs given that it only queries the workers every 5 seconds.

The master node log was made available via the web application. This was invaluable for testing purposes. It allowed us to verify both the data being sent to the worker and the messages being received by the master from the worker. The master node log was also the location where error messages from the worker nodes could be seen. To simplify the main web interface, error messages are kept to a minimum. More verbose error messages are easily visible from the master node log.

2.3.6 Start / Kill Worker Nodes

The master web application is responsible for initially starting each worker node and also provides a mechanism to terminate them all. A worker node is only started once before any jobs in the queue are processed. Each job that starts does not retrigger the start of a worker node. The master uses a SSH connection to remotely connect to the worker node specified by the configuration and issues a command to start the worker. The executable is accessible via a network file system share, and an appropriate pathname is used in the command to this binary executable file.

Before starting all the workers the master also verifies that the rainbow table SSID's are all consistent (the same size). It also calculates the byte offsets that each different worker node will use when they are to load their portion of the rainbow table.

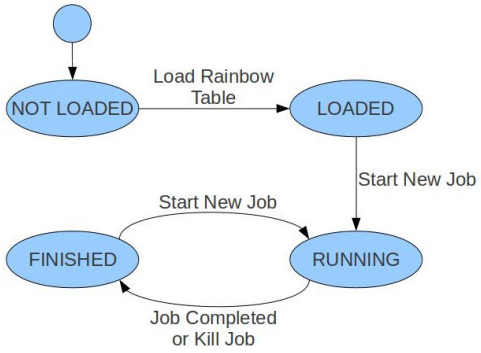

Figure 3. Worker node state diagram

2.4 Worker Node

The worker node is a modified C program based on code by Joshua Wright's coWPAtty. It has been modified to act as a type of service that listens on a TCP socket for instructions or status queries from the master node. The communication specifics will be discussed in the "Master to Worker Node Communication" section later.

2.4.1 Worker States

A worker node may be in any of the following states which are reported to the master node:

- NOT LOADED – The worker is not accepting TCP connections. It may not be running at all or it may still be busy loading the rainbow table into memory.
- LOADED – The worker is ready to accept jobs, has loaded the rainbow table, and has not yet run any jobs.
- RUNNING – the worker is currently busy running a job. It will refuse to run any additional jobs and return an error if asked to do so.
- FINISHED – The worker has finished a job and is ready for the next.
- ERROR – The worker has encountered an error and can no longer run any jobs. It must be restarted.

A worker node state diagram can be found in figure 3. This diagram omits the ERROR state as this state can be entered from any of the other states.

2.4.2 Rainbow Table

The master node web application will provide via command line arguments to each worker node with the relative offsets each should use for the beginning and end of the rainbow table data. In addition a command line option of where to find the rainbow table files is provided. A portion of each network SSID from the rainbow table file is loaded in an even distribution across all workers. This allows them to divide up the work of searching the rainbow table to reduce the time a lookup takes. In addition the entire rainbow table portion is loaded into memory so that performance is not bottlenecked by the disk.

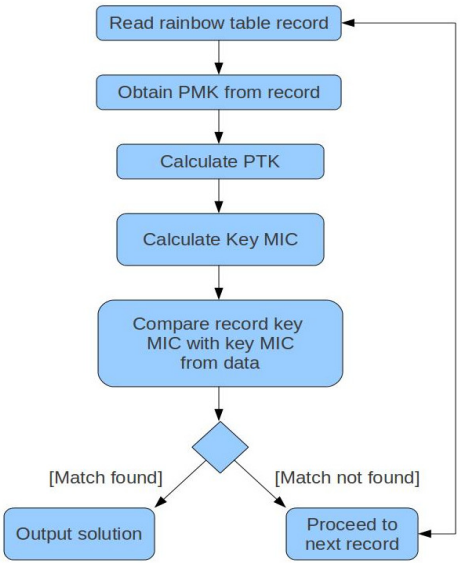

Figure 4. Record processing activity diagram

2.4.3 Computation

When a worker node receives a new job from the master, it first checks that no other jobs are running. An error is returned if another job is already running. Otherwise, the worker node proceeds to create a new thread to perform the rainbow table lookup.

This thread first opens the capture file and determines its validity. If the capture file does not contain the necessary information, an error is returned to the master and the job is terminated. Otherwise, if the packet is valid, the job can proceed.

The next step of the job is to locate the proper portion of the rainbow table. As mentioned previously each worker node stores a portion of the rainbow table in memory. The portion of the table loaded into memory corresponds to the byte offsets provided on start up by the master. When the start command is given the worker parses the SSID passed from the master. Based on the SSID the worker can then determine where in the rainbow table it should start its lookup.

After directing the lookup to the proper location in memory the worker node must read through the rainbow table to find the correct passphrase. The rainbow table is composed of a series of structured records where each record corresponds to a particular passphrase for a particular SSID. To find the correct passphrase the worker must read through each record in the rainbow table.

Each record contains the record length, the passphrase, and the pairwise master key (PMK) which is a hash of the passphrase using the SSID as a salt [3]. The structure used to represent a record can be seen in figure 5. After reading a record, the PMK of the record is used to calculate the pairwise transient key (PTK). This calculation requires using various data gathered when parsing the capture file. Then the message authentication code (MAC) is computed after calculating the PTK. After the MAC is computed the key message integrity code (MIC) can be found. The key MIC calculated for this record is then compared to the key MIC found in the capture data. If they are equal then the passphrase for this record is the solution. Otherwise, the worker moves on to the next record until all records in the rainbow table have been processed. The full sequence for processing the rainbow table records is illustrated in figure 4.

```
struct rainbow_table_record {
      uint8_t rec_size;
      char *passphrase;
      uint8_t pmk[32];
} __attribute__ ((packed));
```

Figure 5. Struct for rainbow table record. Note that this structure is packed so that we can be sure of the byte alignment. This is important so that we can read through the rainbow table efficiently.

2.5 Master to Worker Node Communication

Other than the SSH remote commands used by the master to start each worker node there is also a proprietary protocol we developed for communication between the master node web application and the worker nodes. This is accomplished through an unsecured TCP socket communication between the master and worker nodes.

The worker nodes listen on a configured port for connections from the master and never communicate to each other. The master connects as a client to the listening worker node server socket and sends a request. The worker node will then reply with the appropriate response. To signal the end of request data the master node also does a TCP half-close on the socket which signals an end-of-file marker for the worker node when it is reading data. When the worker node has finished sending its response, it closes the TCP socket. A new connection must be made each time by the master for any subsequent requests.

The request/response packets themselves are simple plain ASCII messages. Since all binary data like the wireless capture file is stored on an NFS share for common shared access binary data does not need to be transmitted.

The packet itself consists of multiple field values that are always null terminated as well as separated with the special control character "ASCII Unit Separator" (decimal code 31). In addition at the very end of the packet is a terminator "ASCII End of Transmission" (decimal code 4).

If a packet is corrupt or invalid (ex: missing a special control character) then the receiver of the packet must raise an error. In the case of the worker receiving an invalid packet from the master it can send back a special ERROR packet response assuming the TCP connection is still valid.

An overview of the different packet requests and responses is given below. Note that \31 or \4 are decimal values for single characters and that there are no line breaks in a packet:

2.5.1 Start Job

START\0\31**jobid**\0\31**/path/to/wifi.pcap**\0\31**/path/output/**\0\31**SSID**\0\31\4

- START signals the request type
- jobid is a uniquely generated id from the master for logging purposes
- /path/to/wifi.pcap is a variable length path that the worker can use to find the input file. Usually on a shared network file system.
- /path/output is where the worker should write output such as a SOLUTION file when a password is found. Usually on a shared network file system so the master can read it.
- SSID is the network wireless name

Possible responses:

1. **SUCCESS_START**\0\31**jobid**\0\31\4
2. **ERROR**\0\31**Message specifying exact error, such as another job is already in progress**\0\31\4

2.5.2 Worker Status Query

STATUS\0\31\4

Possible responses:

1. **STATUS**\0\31**LOADED**\0\31\4

a. Just started up and have already loaded rainbow table into memory

2. **STATUS**\0\31**RUNNING**\0\31**jobid**\0\31\4
    a. Currently running job with given id
3. **STATUS**\0\31**FINISHED**\0\31**jobid**0\31\4
    a. Last job finished was jobid
    b. Ready for next query
    c. Only remembers most recent job
4. **STATUS**\0\31**KILLED**\0\31**jobid**\0\31\4
    a. Job was killed before it could finish
    b. Ready for next query
    c. Only remembers most recent job
5. **ERROR**\0\31**msg**\0\31\4
    a. Worker is in unusable state

2.5.3 Killing Job

KILLJOB\0\31**jobid**\0\31\4

Possible responses:

1. **STATUS**\0\31**KILLED**\0\31**jobid**\0\31\4
2. **ERROR**\0\31**No job with jobid**\0\31\4

2.6 Typical Job Workflow

2.6.1 Packet Capture

The user uses a packet capture tool like aircrack-ng in order to capture Authentication Handshakes between the wireless client and the Wi-Fi Access Point (AP). Packet capture is a 3-step process which needs to be performed by the user as follows:

- Place wireless card in monitor mode ("listen all")
- Start packet capture
- Send a deauthentication packet to wireless client to induce authentication handshake

We have created a script that performs the above 3 steps thus making the task easier for user.

2.6.2 Job Execution

After all workers have been started and have the rainbow table loaded a typical job workflow (assuming no errors) precedes as follows (also summarized in figure 6):

1. User uploads the capture file along with the network SSID to the master node web application.
2. The user's request is assigned a job id and added to the queue.
3. Once any of the previous jobs have finished a thread in the master node web application picks up the user submitted job.
4. The master checks the current status of all the workers to ensure they are in a usable state.
5. The master sends a START request to all the worker nodes with the job input locations and desired output folder.
6. Each worker accepts the request and begins a lookup in the appropriate part of the rainbow table.
7. The master queries the status of the workers every few seconds and shows the job as running in the web interface.
8. Two things can happen
    a. One of the workers finds a solution
    b. None of the workers find a solution
9. If one of the workers found the solution
    a. The master tells all the workers to stop
    b. The master reads the SOLUTION file from the job output directory
10. If none of the workers found the solution
    a. The master doesn't have to take any action on the workers since they are all in a finished state
11. In either case the master
    a. Records the end time for the job
    b. Updates the display to show the solution or "NO SOLUTION"
12. The master goes on to the next job in the queue or waits for more.

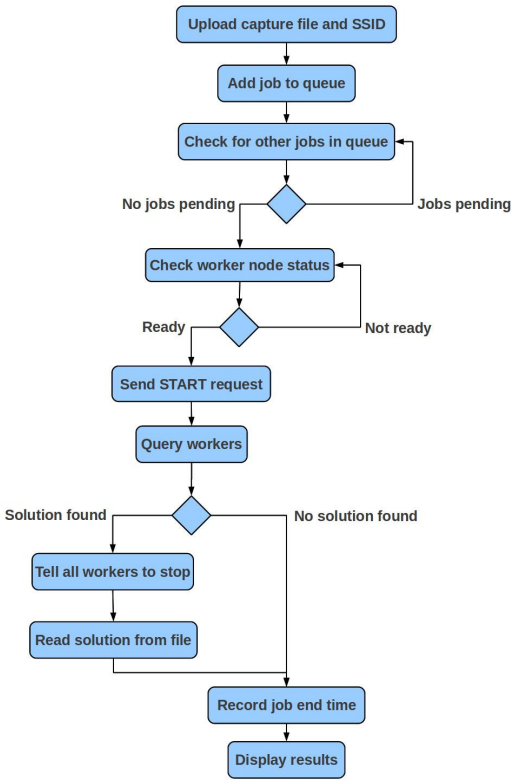

Figure 6. Summary of job workflow

2.7 Fault Tolerance

In the current implementation, failures are minimally handled. Failure of the master node causes the entire system to go down. Conversely, worker node failures are tolerable. As long as one worker is still alive jobs can still be run. However, the rainbow table diminishes in size for each worker failure so the set of possible solutions shrinks. Also since the master node maintains the job queue worker node failure will not cause jobs to be lost.

2.7.1 Master Node

The master node is the key component of this system. If it fails the web application will be unavailable since it is hosted by the master. In addition since the master is the NFS host the workers would no longer have access to anything on the NFS mount. As a result the workers would no longer be able to log messages, read any of the input data such as the capture file, or output any results even if one was found.

Replication of the master node is a clear solution to this problem. If the master node were to fail it could simply be replaced by one of the replicas. Another possible solution would be to create another NFS host separate from the master. This would alleviate some of the pressure on the master node. Replication of this new NFS host would further increase the robustness of this system.

2.7.2 Worker Node

If a worker node were to fail, the system as a whole would still be stable. The user would still be able to submit jobs from the master, but the section of the rainbow table loaded by the failed worker would be lost. As a result any solutions that might reside in that section of the rainbow table would never be found. This is one downside to the current distributed implementation where the rainbow table is loaded into memory. If all of the worker nodes were to fail, the system would no longer be useable.

Worker node failure is detected by the master node through heartbeat messages. When worker node failure is detected the workers can be restarted from the master node. The restart will cause the rainbow table to be loaded into memory again. After the table is reloaded the workers are again ready for new jobs.

In many applications check pointing is performed so that a job does not need to start from the beginning of the computation. It can simply load the state saved in the most recent checkpoint and start computation again at that point. For our purposes this was not feasible. Job completion time is relatively short so restart from the beginning of a job is not a problem. In addition the overhead of the checkpoint would be too expensive in terms of performance. As a result no checkpointing is performed by the worker nodes in this system.

3. TESTING

3.1 Overview

To test our system we did a performance comparison between our distributed version of coWPAtty and the original serial version of coWPAtty.

3.2 Test Data

Capture data was collected from our own personal home wireless networks using aircrack-ng. The networks were secured using WPA encryption. The networks were configured in a number of different ways using various passphrases and SSIDs. More specifically, one data set was captured using a particular SSID (linksys) with a number of different passphrases: the first, middle, and last passphrases in our rainbow table.

- First in Dictionary: !8zj39le
- Middle in Dictionary: korrelie
- Last in Dictionary: }ttringe

Also we capture another handshake using a passphrase that was not in our rainbow table, but a SSID that was in our table. Further, another data set was captured using a SSID that was not in our rainbow table.

3.3 Testing Methodology

Tests were run on all of the data sets using both the original serial coWPAtty on one worker node and the distributed coWPAtty on our test system. For the serial version timing data was simply collected using the *time* command line tool. For the distributed version jobs were sent to the workers using the web application. While the web application does include some timing information in its job output more fine grained information was collected from the worker logs. Each test was run 3 times.

Table 1: Testing results (milliseconds)

	Serial	Distributed
First passphrase	8	5
Middle passphrase	3056	742
Last passphrase	6014	767

3.4 Test Environment

We used a Cisco C210 M1 server with two Intel Xeon E5540 (2.5GHz) processors for a total of 8 logical CPUs (hyperthreading was turned off). The system had 72GB of RAM and sixteen 146GB SAS 6.0gbps drives in a single RAID5 configuration. VMware vSphere Hypervisor ESXi 4.1.0 348481 was the host operating system with 9 underlying virtual machines all running Ubuntu Server 10.10 64-bit Linux. The master node was allocated 2GB of memory while eight worker nodes were allocated 8GB of memory each. All nine virtual machines have 1 virtual CPU and 200GB of disk storage allocated. Access to the cluster from the Internet was limited to SSH and HTTPS to the master node only. All worker nodes had an internal IPv4 network on a private VLAN on the host machine only. The master communicated with the workers over TCP sockets and through SSH remote commands.

The master node had Oracle Java 1.6.0_24 and Apache Tomcat 7.0.11 installed. It also hosted an NFS4 network share to the worker nodes for sharing common code binaries. The actual rainbow table was hosted on local disk on each node to provide better performance during loading of the node software.

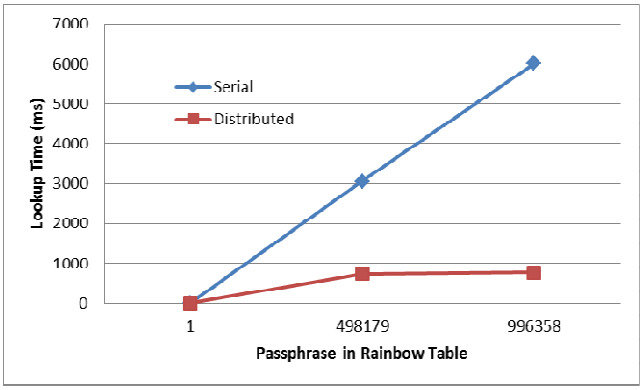

Figure 7. Graphical summary of test results

4. RESULTS

As described in section 3, testing was done to compare the original serial version of coWPAtty and the distributed version of coWPAtty. The results of these tests are summarized in table 1 and graphically in figure 7.

When the first passphrase in the rainbow table was used the difference between the serial and distributed versions was mostly negligible. Intuitively this makes sense. However, the performance increase between the serial and distributed versions becomes clear when the middle and last passphrases are used. For the middle passphrase there was a ~4x performance boost. For the last passphrase in the rainbow table there was ~8x performance boost.

For an embarrassingly parallel problem such as this these results are to be expected. The key idea here is that the distributed version of coWPAtty can be scaled to even further enhance the performance of the system. By increasing the size of the test system performance should theoretically improve even further. In addition, the size of the rainbow table could be increased with less of a performance impact on the system than in the case of the serial version of coWPAtty. The scalability of the system can be seen in figure 7. The lookup time for the serial version of coWPAtty scales linearly with the location of the passphrase in the dictionary. Scaling in the distributed version is much better.

In our case all the nodes actually resided on the same physical hardware, but running the same tests with nodes running on physically different hardware should not yield very different results. This is because there is very little network overhead introduced by our application. The communication that happens between the master and worker nodes is restricted to sending commands and requesting status alone. Worker nodes don't communicate with each other and the NFS share is only used to share the packet capture data, a few kilobytes in size, among the worker nodes.

In addition to the performance of the system for doing the rainbow table lookup the rainbow table load time should be taken into account. In this system the rainbow table is 40 GB in size. With eight worker nodes each node must load 5 GB of data into memory. This is a time consuming and very serial process. We found that it could take anywhere from 2 to 6 minutes for a single worker node to read its designated portion of the rainbow table. Although this process is time consuming it only needs to be performed once.

5. TECHNICAL IMPROVEMENTS

5.1 Offsets For Worker Node Reads

Initially it was the responsibility of each of the worker nodes to determine the offsets in the rainbow table that it needed to read and use during the attack. This in turn implied that each of the worker nodes had to read through the entire file in order to determine the size of the total number of records found in rainbow table i.e. (total file size – size of the header). This operation was being performed at each and every worker node for every rainbow table file (1000 in our case) as the distributed coWPAtty inherited this design from the original coWPAtty.

Upon analysis we realized that performing this operation at each and every worker node was wasteful of computational resources and a better design was to program the master to read through one of the rainbow table files and pass the offsets to workers which can simply fseek() to such offsets in the file.

5.2 Rainbow Table Load Sequence

While deliberating on the design for this distributed application we thought of randomizing the sequence in which each of the worker nodes load the rainbow tables into memory. Hence, instead of every worker node loading the rainbow tables in the same sequence (i.e. in alphabetical order) we felt that changing the sequence for this load operation (in some random fashion) would result in better performance since every worker node would not try to read from the same file. However an opposing thought did occur to us and we decided not to go along with this change. Our reasoning went as follows:

- Read does not require an exclusive lock hence the software(OS, drivers etc.) will not be a bottleneck.
- Reading from the same (or consecutive) blocks/sectors of the disk implies a higher probability of the data being cached (which means faster access)

Following this line of thought we decided not to make any change in the rainbow table load sequence at worker nodes.

5.3 32-bit to 64-bit migration

Given the vast memory requirements of this system we were restricted to 64-bit hardware. In addition to the hardware constraints caution had to be used when addressing memory. Integers were not sufficient in this usage scenario. We had to use the type `long long` to properly address our in memory data structures. In our initial testing this was not a problem because we used a scaled down version of the rainbow table. However, once we scaled the rainbow table up to its full size we soon ran into errors. This was fixed by using the proper data types for our memory offsets.

6. FUTURE WORK

Given more time for this project there are a number of aspects of our system that we could improve to enhance both the usability and performance of the system.

Currently captured data is gathered using the command line tools provided with aircrack-ng. Providing a graphical user interface to the capture process would increase the availability of the system to users possibly uncomfortable with the command line utilities.

The process of rainbow table generation is also currently a serial process. Distributing this task would not only improve performance, but it would also possibly improve the quantity and quality of the generated table.

In our system with 8 worker nodes and a 40GB rainbow table each worker requires at least 5GB of memory just to hold the rainbow table in memory. To lessen the need for memory we could possibly use a hybrid disk and memory approach. Performance would likely be negatively impacted, but that may be a necessary cost in certain systems.

Making the distributed version of coWPAtty amenable to a heterogeneous system would open up a variety of possibilities. One such possibility would be a volunteer computing environment, similar to SETI@Home [10]. One foreseeable problem in this environment would be data transfers. SETI@Home does not have this problem because of the small amount of data sent to the client for each work unit.

Currently the coWPAtty code is not well-designed. Porting the code to Java or C++ would likely make it more readable and open up the possibility for a more extensible design.

Currently both the serial and distributed versions of coWPAtty read through the records of the rainbow table one record at a time. The serial version of coWPAtty is restricted to this model since it reads the table from disk. However, since the distributed version of coWPAtty already loads the table into memory more intelligent data structures could be used. For example, a tree or hash table could be used to increase the speed of record look ups. This would likely require additional memory but could improve performance.

Fault tolerance is another major aspect of distributed coWPAtty that could be improved. Currently the master node is a single point of failure. By replicating the master node system down time could be reduced since one of the master node replicas could be swapped in to act as the new master. Additionally, another node could be added to act as the NFS host. Relieving some of the pressure from the master node could possibly reduce the likelihood of failure. Worker node failure is notably less critical than master node failure. However, there is still room for improvement regarding the fault tolerance of the worker nodes. Currently, the master node can detect worker node failure, but there is no automated system in place to deal with it. A system administrator must restart the worker nodes to bring them all back up. When the master detects worker node failure it could send the restart command itself rather than having a system administrator perform that action. Additionally, if the automated restart of the worker node by the master node failed the master node could try to redistribute the work load to some of the worker nodes that were still alive. Combining the automated restart of the worker nodes by the master node with the automated load redistribution would greatly enhance the fault tolerance in this system.

7. RELATED WORK

The distributed version of coWPAtty is built on top of the original serial version of coWPAtty [2]. The new version runs as a service in a distributed environment. The original version ran as one thread on a single machine. After the job was run the original coWPAtty would immediately exit. Also the rainbow table was read from disk each time the original coWPAtty was run. By leaving coWPAtty running as a service the rainbow table only needs to be read from disk once. This yields a great performance gain.

Before WPA the standard security protocol was Wired Equivalent Privacy (WEP). This security protocol was shown to have a number of design flaws that make it vulnerable to attacks. A number of these attacks have been chronicled by Martin Beck and Erik Tews [6].

WEP was first proven vulnerable in 2001 [7]. RC4 is a stream cipher that is used by WEP. Part of RC4 involves a key scheduling algorithm. This key scheduling algorithm has an identifiable correlation between the key and the output. This correlation can be used to determine the secret key used to authenticate in WEP. This attack is sometimes referred to as the FMS attack. The name is derived from the authors of the first published WEP attack.

After the first weakness of WEP was found [7], many others started to follow. Often they leveraged the same RC4 vulnerability (or the FMS attack). The improvements came in the form of reducing the number of packets that must be captured for a successful attempt to be mounted. One attempt was able to reduce the number of packets required to recover a 128-bit key from 4,000,000 to 1,000,000 packets [8]. This advance both reduces the time required to mount the attack and the storage needed for the packets.

Even after these first vulnerabilities were found WEP was still widely used as the security protocol of choice. This was the case even after WPA was introduced to replace WEP. As a result many other studies were conducted to find novel ways of breaking WEP. Some of these include the KoreK attack [11], the PTW attack [6], and the Chopchop attack [12].

Many have abandoned WEP saying that it is “completely insecure” [8]. WPA was its replacement. Eventually WPA was replaced with WPA2. So far these have remained relatively secure. The same types of vulnerabilities in WEP have not been found in WPA and WPA2. A number of attacks do exist though. Both WPA-PSK and WPA2-PSK are susceptible to password cracking attacks although no known attacks exist against the Enterprise mode of WPA1/2.

Possibly the first attack on WPA was presented in [6]. It demonstrated that a chopchop like attack (an attack used to crack WEP) could also be used to crack WPA. A variety of conditions must be met for this attack to work on WPA. However, it was shown that these conditions are not unreasonable in most wireless networks [6]. Additionally, this attack only works for WPA with TKIP not CCMP.

As the age of general purpose graphics processing units (GPGPU) computing dawned the extreme parallelism offered by the GPU became clearer and more widely used. GPGPU computing is now being used to crack WPA encrypted wireless networks. Pyrit is one such approach [9]. Using CUDA [13], Pyrit can compute up to 89,000 pairwise master keys per second. This technique could then be used as a brute force attack on the system.

Commercial solutions exist for distributed password cracking in the form of ElcomSoft Distributed Password Recovery [14] which claims to run over a distributed network of computers and can be used for cracking WPA-PSK keys. However it is designed to run over a local network and cannot expand onto the cloud. It also requires Windows to run and does not use rainbow tables, but instead relies on brute force and dictionary attacks.

8. SECURING WIFI

In this paper we have shown that wireless security is still imperfect. Even more sophisticated security protocols like WPA2

are still susceptible to attack. So how does the average person protect their wireless networks? There are three key things that can be done to improve security of wireless networks:

- Use WPA2-PSK (Enterprise if possible)
- Use a non-trivial network name or SSID
- Use a non-trivial passphrase if using PSK

As simple as these measures may seem, many people fail to do them. As a result personal wireless networks remain a key point of attack for many cybercriminals.

So what exactly is a non-trivial SSID and passphrase? First of all, many wireless networks will suggest a default SSID when they are first setup. An example of this is the "linksys" SSID. At a minimum this default SSID should be changed. Ideally it would be contain upper-case and lower-case characters as well as numbers and non-alphanumerical characters. The same principles apply to choosing a passphrase. By performing these simple tasks a user can significantly increase the security of their wireless networks. It should be noted that these precautions mainly guard against dictionary based attacks similar to the one presented here. Brute force attacks could still be possible. However, due to the computational requirements, brute force attacks are much less likely to be a significant threat at this time.

9. CONCLUDING REMARKS

Mobile devices are becoming more and more popular. As a result wireless security has become an increasingly important topic. WEP has been shown to be very insecure. As a result WPA and subsequently WPA2 were developed to enhance wireless security. However, WPA and WPA2 are still vulnerable to attack. One such attack is illustrated in this paper.

Most personal wireless networks use WPA-PSK. This security protocol relies on knowing a single passphrase. When weak passphrases are used the network is extremely susceptible to attack. Dictionary attacks can be very successful on weak passphrases. This paper discusses a distributed system for performing a dictionary attack on WPA1/2-PSK.

Building off of an existing system called coWPAtty we are able to crack a WPA encrypted network in less than second. This system uses a rainbow table. Currently this table contains 1000 SSIDs and 996358 passphrases. This table is distributed across a cluster of nodes. Each node loads a section of the table into memory.

By loading the rainbow table into memory and distributing the computation our system can achieve a performance improvement of nearly a factor of 8 compared to the original serial implementation of coWPAtty. In terms of world time this translates to an improvement of several seconds. While this may seem trivial, the value added here is that this solution is much more scalable and suitable to function as a service.

10. ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

Special thanks to Cisco for loading us the necessary hardware and Internet connectivity to run our cluster for testing.

Thanks to our professor Richard Han for guidance during our project for his class.

We give acknowledgment of the great work done by Joshua Wright, the original developer of coWPAtty, from which our work is based.

11. REFERENCES

[1] Aircrack-ng. 2011. http://www.aircrack-ng.org/

[2] Church of the Wifi. 2009. Church of Wifi Uber coWPAtty lookup tables. http://www.churchofwifi.org/Project_Display.asp?PID=90

[3] IEEE Computer Society. 2007. IEEE Std 802.11™-2007 New York, NY.

[4] Moskowitz, Robert. 1993. Weakness in Passphrase Choice in WPA Interface. http://wifinetnews.com/archives/2003/11/weakness_in_passphrase_choice_in_wpa_interface.html

[5] Wright, Joshua. 2009. coWPAtty http://www.willhackforsushi.com/?page_id=50

[6] Erik Tews and Martin Beck. 2009. Practical attacks against WEP and WPA. In Proceedings of the second ACM conference on Wireless network security (WiSec '09). ACM, New York, NY, USA, 79-86.

[7] Scott R. Fluhrer , Itsik Mantin, Adi Shamir, Weaknesses in the Key Scheduling Algorithm of RC4, Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography, p.1-24, August 16-17, 2001.

[8] Adam Stubblefield, John Ioannidis, Aviel D. Rubin, A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP), ACM Transactions on Information and System Security (TISSEC), v.7 n.2, p.319-332, May 2004.

[9] Pyrit. http://code.google.com/p/pyrit

[10] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. 2002. SETI@home: an experiment in public-resource computing. Commun. ACM 45, 11 (November 2002), 56-61.

[11] WEP Cracker. http://www.oxid.it/ca_um/topics/wep_cracker.htm

[12] Chopchoptheory. http://www.aircrack-ng.org/doku.php?id=chopchoptheory

[13] CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html

[14] ElcomSoft Distributed Password Recovery http://www.elcomsoft.com/edpr.html