An FPGA architecture for the recovery of WPA/WPA2 keys

by

Tyler Blaine Johnson

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Joseph Zambreno, Major Professor

Phillip Harrison Jones

Thomas Earl Daniels

Iowa State University

Ames, Iowa

2014

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

Wi-Fi protected access (WPA) has provided serious improvements over the now deprecated wired equivalent privacy protocol (WEP). WPA, however, still has some flaws that allow an attacker to obtain the passphrase. One of these flaws is exposed when the access point (AP) is operating in the WPA personal mode. This is the most common mode as it is the quickest and easiest to configure. It requires the attacker to capture the traffic from four-way handshake between the AP and client, and then provide enough compute time to reverse the passphrase. Attackers quickly noticed that by investing the compute time in advance, and storing their work, they could decrease the time-to-crack for an AP. This caused attackers to start compiling large lookup tables based on dictionaries of common passwords and common SSIDs. The attackers are required to compile a separate lookup table for each SSID, making this style of attack most feasible against APs with a common SSID and password.

The work in this thesis will focus on creating an FPGA based architecture to accelerate the generation of the lookup table, given a dictionary of possible Pre-shared Keys and an SSID. The application of this work would be most useful for attacking one-off SSID's. This is because most common SSID's already have a generated lookup table that can be downloaded much faster than it could be generated, so this regeneration would be wasteful. The application will also provide a manner to check for a valid Pairwise Master Key during the table generation phase.

CHAPTER 1. Introduction

Wi-Fi Protected Access (WPA) was created to replace the Wired Equivalent Privacy (WEP) protocol after it was proven to be insecure. WPA fixed many issues with the design of WEP, but still has a few issues of its own. The work presented in this thesis will focus on creating an FPGA architecture to attack the WPA authentication process, using a previously known attack vector. The goal of the architecture in this thesis is to provide more efficient method to reveal the passphrase required to authenticate with an access point, while operating in WPA personal mode.

The remainder of Chapter 1 will provide an outline of the history of wireless LAN. Chapter 2 will provide the details of how the attack is carried out, and will provide an analysis on the time requirements of this style of attack. Chapter 3 will introduce the four-way handshake that the authentication process relies on, and provide details of each algorithm that the handshake uses during the authentication process. Chapter 4 will first introduce the Covey platform used to implement the architecture, and will then describe the full architecture developed in detail. Chapter 5 will provide an analysis of the results achieved in this work. Finally, Chapter 6, will provide a summary work in this thesis and where the work can be continued in the future.

1.1 A BRIEF HISTORY ON THE EVOLUTION OF WIRELESS LAN

In the late 1960s, researchers at the University of Hawaii started on a project to allow the university to share computing resources across its main campuses, which were spread across several islands [4]. The project required that the communications were made via radio link, to avoid building a physical infrastructure between the islands. Out of this research project arose the ALOHA system, also known as the ALOHAnet.

The ALOHA systems main contribution to modern wireless networking was the concept of sharing a transmission medium between multiple clients [5]. The ALOHA system used two radio channels, one for broadcasting from the main computer center to its clients, and the other for the clients to broadcast back to the center. Since there were many clients, the client to center channel would become noisy when many clients had information.

This introduced the idea of a Medium Access Control or MAC. By using acknowledgment packets when messages were sent from the center to the clients, and random back offs by the clients when an acknowledge packet went missing, the noisy channel was made usable and reliable. This concept would also be borrowed and improved upon by the developers of Ethernet [19]. The idea of a MAC is still in use today by many networking protocols.

In 1985 the FCC decided to open up three radio frequency bands for civilian use, with the three bands designated as industrial, scientific, and medical [19]. These bands became know as the ISM bands, and were available for use without license, given certain restrictions were followed. This decision by the FCC opened up the playing field for companies to start researching the application of wireless communications for consumer devices, while keeping the investment risks lower.

After the ISM bands were released, several companies started to take in interest in developing a wireless LAN technology. One of these companies was the National Cash Register Company. The NCR realized there was an opportunity for developing wireless cash registers. This opportunity was driven by the recent opening of the radio bands and the constant remodeling of department stores, which were the main customers of NCRs devices [19]. NCR then funded a feasibility study for this application, and when it showed positive results, they invested more into the project. Inside the Institute of Electrical and Electronics Engineers (IEEE), a group had been created known as the 802, which focused on the development of LAN standards. This group had already been developing standards under the 802.3 Ethernet and 802.4 Token Bus groups, along with other LAN standards. The idea of a developing a wireless LAN specification within the 802 was gaining steam, in part due the NCRs interest in wireless LAN technologies. In 1990 the IEEE 802.11 group was officially created, with the goal of creating a wireless LAN standard. In 1997 the first 802.11 wireless LAN standard was officially approved by the group [16].

Even though manufactures were following the 802.11 standards, there were still issues with cross-manufacturer compatibility of their 802.11 enabled devices. It soon became evident that there needed to be a group to control the testing of 802.11 devices with the goal of helping to ensure the reliability and assure consumers their devices were going to work. The 802.11 group was only created to design the specifications for wireless LAN technologies, not to test and verify manufactured devices.

The need for this missing testing body brought the Wi-Fi alliance to life. In 1999 the Wi-Fi alliance was officially created by several industry leaders of the time, and the term Wi-Fi was coined. The Wi-Fi alliance began officially certifying devices in 2000, and those that passed were allowed to use the Wi-Fi Alliances branding on their devices. This branding is what started to increase the consumer demands for wireless LAN devices [6].

Today Wi-Fi has become one of the most successful globally used technologies, and has become the most popular method to provide internet connectivity to devices. The 802.11 group has since released many amendments to their first specification, for both performance and security enhancements. Wi-Fi has improved from a 1-2 Mbps data link all the way up into the hundreds of Mbps. To top off these improvements, a new specification with several Gbps is expected to be released in the near future. It is clear that this technology has come a long way since its first conception, and has become an integral part of our daily lives.

CHAPTER 2. REVIEW OF LITERATURE

2.1 Overview of the Attack

The attack carried out in this paper is a dictionary attack, a style of brute force attack, whose attack time is restricted by the size of the chosen dictionary. The attack can only be successful against an AP that has chosen a password that exists in the dictionary. To search the entire possible password space, the attack time would be prohibitive, as discussed in section 2.2, thus the dictionary approach is used.

This attack has several phases, starting with the first phase of capturing the WPA 4-way handshake traffic. See section 3.1 for details of the handshake. This capturing can be accomplished in one of two ways. The first is to simply monitor the Wi-Fi traffic in the area, until a device attempts to authenticate with the targeted access point. This method is a passive approach and is completely undetectable by the access point. The second option is an active approach and involves forcibly disconnecting a current client from the targeted access point. This is done by sending a deauthentication packet to the client with spoofed address from the attacker. This causes the client to disconnect from the AP and reconnect. As the device reconnects the 4-way handshake can then be captured. Forcibly disconnect a client can speed up the capture time, but is possible to detect.

After the 4-way handshake is captured, a program such as coWPAtty [2] can be used to extract the required components of the handshake to begin the attack. These components are listed in table 2.1. During the handshake process both parties will exchange the un-encrypted nonce values. The MAC address can be extracted from the packets, which require a MAC address to be properly routed. The MIC value is also exchanged between both parties to verify the other party does know the PSK, and is what actually ends up being attacked.

Table 2.1 Four Way Handshake Extracted Components

Component	Description
ANonce	The nonce of the authenticator
SNonce	The nonce of the supplicant
AMac	The Hardware MAC of the authenticator
SMac	The Hardware MAC of the supplicant
SSID	The identity descriptor of the AP
MIC	The message integrity code from the PTK

Finally, the SSID is required to generate the PMK, but can be extracted from the APs beacon frames. After capturing the handshake the attack process can begin. The attacker will select a dictionary of PSK values to try, and start creating the PMK values associated with the each PSK in this dictionary. After each PMK is generated, the attacker can generate the respective PTK by providing the PMK and data from table 2.1 to the PRF-512 function, figure 2.1. Once the PTK is generated, the captured MIC value is compared to the computed MIC, which is taken directly from the PTK. If the two MIC values are equivalent, then the attacker has found the correct PSK. If the values differ, the attack will continue. This phase is considered the searching phase, and is the most time consuming part of the attack. This phase requires 4096 iterations of the PBKDF2 function section 3.2.3 to generate a single PMK.

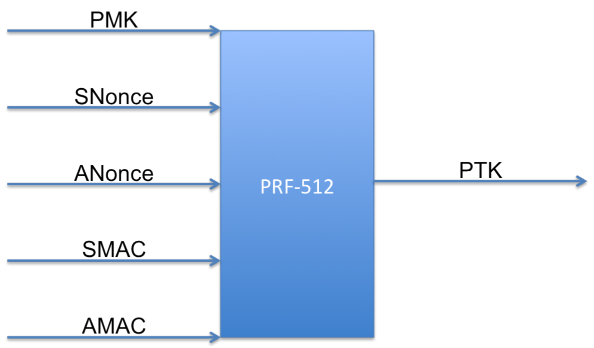

Figure 2.1 Diagram of PTK Generation

The attack time can be decreased in terms of "time to passphrase reversal from time of handshake capture", by pre-computing large lookup tables of PMKs. The PMK is generated by using the PBKDF2 function, and takes as an input the passphrase (PSK) and the SSID of the AP. Thus by using common PSK and SSID values, the work can be done in advance, and the time can be cut down significantly during the attack. The caveat being that the SSID being attacked must match for one of the tables that have been previously generated. Tables for common PSK and SSID combinations are publicly available at [23] and other locations.

There are currently a number of programs that attempt to provide this attack for WiFi auditing available as open source. A few of these are coWPAtty [2], Aircrack [1], and pyrit [3]. coWPAtty is credited for providing the first software to implement this attack, but is considerably slower than the following two programs. Aircrack provides a quicker implementation of the attack along with some other known WEP attacks. Finally, Pyrit provides a subset of this attack, specifically the PMK generation, which is capable of running on Nvidia CUDA and Open CL supported platforms. The use of GPUs greatly enhances the PMK generation speeds. Pyrit even has the capabilities to use resource sharing of GPUs so that multiple parties can cooperate on the generation of PMK lookup tables.

2.2 Analysis of the Attack Space

Knowing the size of the attack space is an important aspect of any attack. By calculating the size of the possible space, you can find the upper and lower bounds for the time consumption of the attack that is being carried out. In a brute force attack the entire key space is searched, one by one, until a valid key is found. This causes a brute force attack to always yield the worst upper bound on attack time, since the brute force attack needs to exhaust the entire possible key space in the worst case. By searching the entire key space, a brute force attack provides a one hundred percent probability that the passphrase can be recovered provided enough attack time. The attack time, however, often excludes brute force methods from being used since it is generally prohibitively large.

The size of a brute force attack space is simply a permutation based on the available digits or characters, and length of the key. For example, if only a 1 or 0 could be selected, there are only two possible ways to create a key. If two digits of either a 1 or 0 could be selected then

Case	Number of Printable ASCII Characters	Permutation	Space Size
Best Case	8	95^8	$6.63E+15$
Worst Case	63	95^{63}	$3.95E+124$

Table 2.2 Best and Worst Case Brute Force Attack Spaces

there would be

$$2 * 2 = 4$$

possible ways to create a key. Again if three digits of either 1 or 0 could be chosen, there would be

$$2 * 2 * 2 = 2^3 = 8$$

ways to select a key. This pattern continues for the length of the key and the number of options for each spot.

In WPA/WPA2 the attack space is limited by the 95 printable ASCII characters. For a passphrase the user must choose between 8 and 63 of these 95 ASCII characters, or the user also has the option to choose a 64 digit hexadecimal number. The most common case will be the users selecting an 8 to 63 character passphrase, with most users selecting closer to the minimum of 8 characters range.

With this information, the attack space of WPA/WPA2 is calculated in table 2.2. This table shows that even for the best case, an 8 character password, the size of the brute force attack space is still a large number. To demonstrate the total attack time, consider if the attacker could generate and test 1,000 passphrases per second, it would take

$$time_{hours} = (6.63E+15)phrases * \frac{1sec}{1000phrases} * \frac{1min}{60sec} * \frac{1hr}{60min} = 1842834531.35851_{hours}$$

This time complexity prohibits the effectiveness of a brute force attack on this space.

The brute force attack space, however, assumes that the user has selected a random passphrase from the available character set. If the passphrase is not generated at random, it is possible to remove a large part of the attack space, effectively lowering the attack time. There are several common ways to reduce the attack space in this manner but the dictionary attack is probably the most simple to implement.

Number of Phrases	$Time_{Seconds}$	$Time_{Minutes}$	$Time_{Hours}$
(235,886)	235.87	3.93	.07
(500,000)	500	8.33	.14

Table 2.3 Time Complexity of Dictionary Attacks Assuming 1,000 $Passphrases_{\text{second}}$

The dictionary attack is effective by limiting the attack space to known words from the dictionary of a given language, a list of names, common terms, etc.... The attacker can then compile a list of these possibilities, which provides a much smaller attack space than the brute force method does. The tradeoff of this approach is that it turns the one hundred percent success probability of the brute force attack into an attack with a much lower success probability, which is based on the selection of the dictionary. It also requires that the passphrase is actually present in the attacker's dictionary to reverse the passphrase.

To determine the attack space for the dictionary attack, the size of the dictionary must first be determined. According to the Oxford Dictionaries [13] and Merriam-Webster [20], there are currently around a half of a million entries in the English dictionary. A more modestly sized dictionary is provided by most Unix/Linux based systems at "/usr/share/dict/words". At the time of this writing, 235,886 words were counted by the wc command on a Mac OS X 10.7.5 based system with Darwin 11.4.2 as the kernel version. The time complexity to attempt each of these dictionaries, with the same rate of 1000 passphrases/sec, can be found in figure 2.3. It can be seen in figure 2.3, that by limiting the attack space with the dictionary approach, the time complexity becomes much more manageable. This comes at a cost of lower attack success rates. The dictionary attacks success rate can be improved by using programs such as John The Ripper [22], to alter each word in the dictionary to contain common prefixes and postfixes. For example, the common passphrase password would exist in the dictionary. A common postfix for this passphrase is 1 to yield password1, which would not be in the English dictionary. John The Ripper will generate attempts such as password1, password2, etc for each entry in the dictionary. Doing this, however, will add back to the time complexity of the attack. In table 2.4 the complexities added by using John The Ripper are analyzed for some common cases,

Configuration	Example Phrase	Time(Hours)
Any dictionary word + 1 digit (1-9)	word1	1.5
Any dictionary word + 2 digit (1-9)	word12	13.5
Any dictionary word + 3 digit (1-9)	word123	121.5
Any dictionary word + 4 digit (1-9)	word1234	1093.5
Any dictionary word + 5 digit (1-9)	word12345	9841.5
Any dictionary word + 6 digit (1-9)	word123456	88573.5
Any dictionary word + 7 digit (1-9)	word1234567	797161.5
Any dictionary word + 8 digit (1-9)	word12345678	7174453.5
Any dictionary word + 9 digit (1-9)	word123456789	64570081.5

Table 2.4 Time Complexity of Dictionary Attack Using John Assuming 1,000 $Passphrases_{\text{second}}$

again assuming the 1,000 passphrases/sec rate.

CHAPTER 3. The WPA Authentication Process

3.1 WPA Four Way Handshake

WPA depends upon a four way handshake process to authenticate with a client to the network. In WPA pre-shared key mode, there are only two parties involved in the authentication process, the authenticator (access point) and the supplicant (mobile client). Both parties must prove to each other that they know the pre-shared key to ensure a secure connection. It is important to note that the pre-shared key is never sent between the supplicant and authenticator, it is up to the user to program this key into each party. This is also important because there is no secure channel of communication before the authentication process has completed, so sharing the pre-shared key would be done unencrypted and easily discovered by outside parties. The four way process is broken down into message A through D in the following section. Figure 3.1 provides a demonstration of this process through time.

3.1.1 Message A *Authenticator* ← *Supplicant*

The first step is for the authenticator to generate a nonce value. The nonce value is a pseudo random value generated by a publicly known and repeatable process. In a perfect implementation a Nonce value would never be used more than once. Since this property is not feasible, it is enough to provide a very high probabilistic guarantee that the value will not be chosen again. The pseudo random value is generated by the Pseudo Random Function 256 or PRF-256, as defined by the WPA specifications. The nonce value that the authenticator generates is denoted as the ANonce. Upon generation of the ANonce, the authenticator sends a message to the supplicant containing the ANonce value.

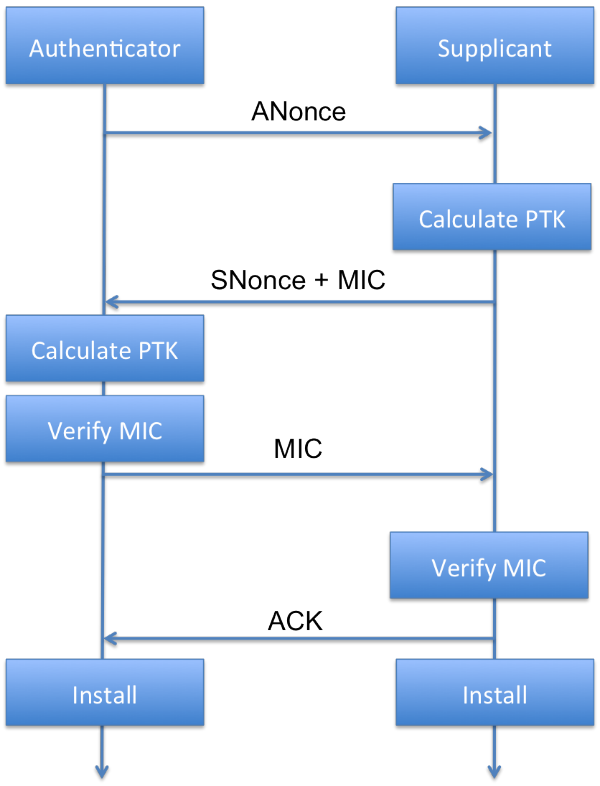

Figure 3.1 Diagram of the EAPOL 4-Way Handshake

3.1.2 Message B *Supplicant* $\leftarrow$ *Authenticator*

The supplicant then generates a nonce value using the same process as the authenticator. This nonce value is denoted as the SNonce. Upon the supplicant receiving message A from the authenticator, the supplicant can generate the transient key. This key is required to be generated by both parties, and allows each party to verify that the other has the correct PSK. To generate the transient key, the supplicant needs to have the data listed in table 3.1. Part of the transient key is known as the message integrity check, or MIC. This value, along with the SNonce is then transmitted back to the authenticator by the supplicant.

Data	Derivation
SNonce	Generated by the supplicant
ANonce	Received from authenticator in message a
SMAC	MAC address of supplicant and is known by supplicant
AMAC	MAC address of authenticator and is extracted from message a
PSK	Known by both parties

Table 3.1 List of requried data for transient key generation

3.1.3 Message C *Authenticator* $\leftarrow$ *Supplicant*

Once the authenticator receives message b it has all the values required to generate the transient key. The authenticator then generates the transient key, and checks that the MIC value in message B matches the MIC value that it has just generated. This proves that the supplicant knows the value of the PSK. Assuming that the MIC value is correct the authenticator then sends message C to the supplicant. Message C allows the supplicant to ensure that the authenticator is a trusted party, if the authenticator did not have a matching PSK, the MIC would differ. Message C also informs the supplicant that the communication channel is about to be encrypted.

3.1.4 Message D *Supplicant ← Authenticator*

The final part of the handshake serves to allow the supplicant to acknowledge that the authenticator is now going to use encryption for the communication. After the supplicant transmits message D, it will install the encryption keys on the channel. After the authenticator receives message D, it will install the encryption keys. After this point, all further unicast communication is protected by this encryption, until the client disconnects from the access point [14].

3.2 WPA Authentication Components

3.2.1 SHA1 - Secure Hash Algorithm

SHA-1 is a secure one-way hashing function that is used in supporting the Pre Shared Key (PSK) architecture of WPA/WPA2. It was developed by the National Security Agency (NSA) and later published by the National Institute of Standards and Technology (NIST). The SHA-1 is considered a secure hashing algorithm by the NIST, and has been published as an acceptable hashing function for use in sensitive data by government agencies [21].

Hashing algorithms are used to process a message and produce a condensed representation of the message. The representation is called a message digest, and for a perfect hashing function, is a unique digital signature of the message. For a true one-way hashing function, the original message cannot be recovered from the resulting hash, but can be used to compare against other message digests. If two of these message digests are equal, then it is highly probable that the two digests are from the same original message. In this manner, hashing functions are useful for verifying digital signatures and performing fast comparisons between digital contents.

The SHA-1 hashing algorithm is valid for messages with a size less than 2^{64} bits, operates on blocks of size 512 bits, uses a word size of 32 bits, and has a resultant message digest of 160 bits. SHA-1 produces the smallest message digest of all the algorithms in the Secure Hash Standard (SHS) [21], and is considered the least secure of these. Table 3.2 shows the SHA1 size properties, along with other SHS algorithms published by the NIST.

SHA-1 is can be separated into two stages of processing. The first stage is the pre-processing

Algorithm	Message Size	Block Size	Word Size	Message Digest Size
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Table 3.2 Secure Hash Algorithm Properties [21]

stage, where the message is manipulated in preparation of being hashed. This involves padding the message and parsing the message into blocks. The second stage is where the message is processed by the various logical functions for 80 rounds, to produce a resultant hash.

3.2.1.1 Pre-processing

Stage 1 of the SHA-1 hashing algorithm is used for preprocessing of the message, to prepare it for the hashing stage. The preprocessing stage has three sub-stages, padding the message, parsing the message into blocks, and finally setting the initial hash values. Padding ensures that the message is a length multiple of 512. Parsing the message involves dividing the message into blocks to operate on. The final step is to set the initial hash values.

3.2.1.2 Padding The Message

The message is padded to ensure that it is a multiple of 512, without this padding stage, the message would not divide into an integer number of blocks. This is required, since the hashing stage is only able to perform work on a full block. The message is padded in the following manner

- Append a binary 1 to the end
- Solve the following equation for k: $1 + k + mlength = 448 mod 512$
- Append k binary 0's to the message

- Append the binary value of the original message length represented as 64 bits in the last two words of the block

The number 448 is derived from $512 - 64 = 448$, where 512 is the block size and the last 64 bits are reserved to store the length of the message. If the message ends in the last two words then an extra block is required. This block is entirely zero padded, and the message length is set in the last two words of the extra block.

3.2.1.3 Parsing

The message and the newly added padding are then parsed into N 512 bit blocks. These blocks are denoted as

$$M^1, M^2, \ldots M^N$$

Since each block size is 512, and a word size is 32, each block can be represented as 16 words. These words are denoted as

$$M_0^i, M_1^i, ..., M_{15}^i$$

where the superscript represents the block number, and the subscript represents the word location in the ith block.

3.2.1.4 Set The Initial Hash Value

The final hash value is 160 bits long, and is created by concatenating 5, 1 word intermediate hash variables H0,. . . ,H4. These intermediate hash variables are initialized before the hashing begins, using the constants in table 3.3.

Intermediate Hash Variable	Constant Value
H^0	0x67452301
H^1	0xEFCDAB89
H^2	0x98BADCFE
H^3	0x10325476
H^4	0xC3D2E1F0

Table 3.3 Initial Hash Values [21]

3.2.1.5 Hashing

The hashing phase takes as an input the message blocks derived in the parsing stage of the pre-processing phase. Each of these blocks, $M^0, \ldots, M^N$, is processed in sequential order to produce the final hash value. The hashing process consists of 4 sub stages, preparing the message schedule, initializing the five working variables, performing logical operations, and computing the ith intermediate hash value H^i. This process is repeated N times, once for each block. The full algorithm can be found in algorithm 1.

The SHA1 algorithm defines the $ROTL^n(x)$ function and the $f_t(x, y, z)$ functions in addition. The $ROTL^n(x)$ function takes the input x and does a circular left shift n places. It operates on the word size of the respective SHA class, or 32 bits in the case of SHA1. The $f_t(x, y, z)$ function is a combination of three other logical functions, and the result is selected by t. These functions are described in algorithm 2 and algorithm 3. The constant K is defined in table 3.4.

Range of t	K Value
$0 \leq t \leq 19$	0x5A827999
$20 \leq t \leq 39$	0x6ED9EBA1
$40 \leq t \leq 49$	0x8F1BBCDC
$60 \leq t \leq 79$	0xCA62C1D6

Table 3.4 Constant K Values [21]

Algorithm 1 SHA1

```
for $i = 0$ to $N$ do                                         ▷ For all message blocks

    for $t = 0$ to 79 do                                      ▷ Prepare message schedule
        if $t < 16$ then
            $W_t \leftarrow M_t^i$
        else
            $W_t \leftarrow ROTL(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}, 1)$
        end if
    end for

    $a \leftarrow H_0^{i-1}$                                  ▷ Initialize working variables a,b,c,d, and e
    $b \leftarrow H_1^{i-1}$
    $c \leftarrow H_2^{i-1}$
    $d \leftarrow H_3^{i-1}$
    $e \leftarrow H_4^{i-1}$

    for $t = 0$ to 79 do                                      ▷ Preform logical operations
        $T \leftarrow ROTL^5(a) + f_t(b, c, d) + e + K_t + W_t$
        $e \leftarrow d$
        $d \leftarrow c$
        $c \leftarrow ROTL^{30}(b)$
        $b \leftarrow a$
        $a \leftarrow T$
    end for

    $H_0^i \leftarrow a + H_0^{i-1}$                          ▷ Compute the $i$th intermediate hash value
    $H_1^i \leftarrow b + H_1^{i-1}$
    $H_2^i \leftarrow c + H_2^{i-1}$
    $H_3^i \leftarrow d + H_3^{i-1}$
    $H_4^i \leftarrow e + H_4^{i-1}$
end for

$Return \leftarrow H_0^N \| H_1^N \| H_2^N \| H_3^N \| H_4^N$       ▷ $\|$ is the concatenation operation
```

Algorithm 2 $ROTL^n(x)$ function

```
$Return \leftarrow (x << n) \vee (x >> w - n)$
```

Algorithm 3 $f_t(x, y, z)$ function

```
if $0 \leq t \leq 19$ then
    $Return \leftarrow (x \wedge y) \oplus (\neg x \wedge z)$

else if $20 \leq t \leq 39$ then
    $Return \leftarrow x \oplus y \oplus z$

else if $40 \leq t \leq 59$ then
    $Return \leftarrow (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$

else if $60 \leq t \leq 79$ then
    $Return \leftarrow x \oplus y \oplus z$

end if
```

3.2.2 HMAC

Hash Based Message Authentication Code (HMAC) provides a mechanism to calculate a message authentication code (MAC) based around a cryptographic hashing function, provided a key. HMAC may use any cryptographic hashing function as the underlying calculations of the final MAC, for example SHA-1 in section 3.2.1 or MD5. The output size of the resulting MAC and the cryptographic strength of HMAC are dependent upon the underlying hashing algorithm as well, making this selection an important part of the design.

WPA uses an HMAC-SHA1 implementation, where SHA1 is chosen as the underlying hashing algorithm. The resultant MAC is thus 160 bits in length. HMAC iterates on two rounds of hashing, with certain logical operations performed on the key and text between each round. The function definition of HMAC is as follows:

$$H(K \oplus opad, H(K \oplus ipad, text))$$

Here K is the key , opad and ipad are constants in table 3.5, text is the salt, and H is the underlying hashing algorithm. In round 1, the text is appended to the end of the result of K xor ipad, and then hashed. Upon completion of the first hashing process, the result of the hash is then appended to K xor opad, and the resulting value is hashed one final time to produce the final result [18].

Table 3.5 HMAC ipad and opad Constants [18]

Constant	Value
ipad	byte 0x36 repeated B times
opad	byte 0x5C repeated B times

3.2.3 PBKDF2

WPA/WPA2 depends upon the PBKDF2 function to generate the PMK values. The PBKDF2 is specified in [17], and can use any underlying pseudo random function. However, the WPA/WPA2 specification requires that HMAC-SHA1 is used for this random generator, see sections 3.2.2 and 3.2.1. The PBKDF2 function requires as an input, a PSK passphrase, the SSID of the network, the required size of the derived key, and finally the number of times the underlying pseudo random function should be iterated. For the application of WPA/WPA2, the iteration count is fixed at 4096 and the derived key length is fixed at 256 bits.

The result from the PBKDF2 is created by concatenating as many underlying pseudo random generation results that are required to generate a derived key the length of the dklen value. Since HMAC-SHA1 produces a 160 bit value, two HMAC-SHA1 values are required to achieve a derived key of 256 bits. Notice that the two 160 bit results produce a 320 bit value, so the last 64 bits of the second HMAC-SHA1 output are discarded. The PBKDF2 function is defined as follows

$$PBKDF2(P, S, c, dkLen)$$

with the parameters identified in table 3.6. A full function definition can be found in algorithm 4.

Table 3.6 PBKDF2 Function Parameters

Parameter Name	Parameter Usage
P	The PSK of the AP
S	The SSID of the AP
C	The number of rounds to apply the PRF
dkLen	The size of the derived key

Algorithm 4 PBKDF2

```
if $dlLen > (2^{32} - 1) * hLen$ then
    $Return \leftarrow$ "derived key too long"

end if

$L \leftarrow \lceil dkLen/hlen \rceil$
$R \leftarrow dkLen - (L - 1) * hLen$

for $b = 0$ to $L$ do                                    ▷ For 0 to L

    $T_b \leftarrow F(P, S, c, b + 1)$                   ▷ Function F is defined in algorithm 5

end for

$Return \leftarrow T_0 || \ldots || T_{L-1}((Hlen) downto (Hlen - R))$    ▷ Concatenate Blocks
```

Algorithm 5 PBKDF2: $F(P, S, c, i) function$

```
for $j = 0$ to $C$ do                                    ▷ For 0 to C
    if $j = 0$ then
        $U_j \leftarrow PRF(P, S || Int(i))$             ▷ Append 64 bit representation of i to S

    else
        $U_j \leftarrow PRF(P, U_{j-1})$

    end if
end for

$Return \leftarrow U_0 \oplus U_1 \oplus \ldots \oplus U_{C-1}$
```

CHAPTER 4. The Architecture

4.1 Introduction to the Convey HC-2 Platform

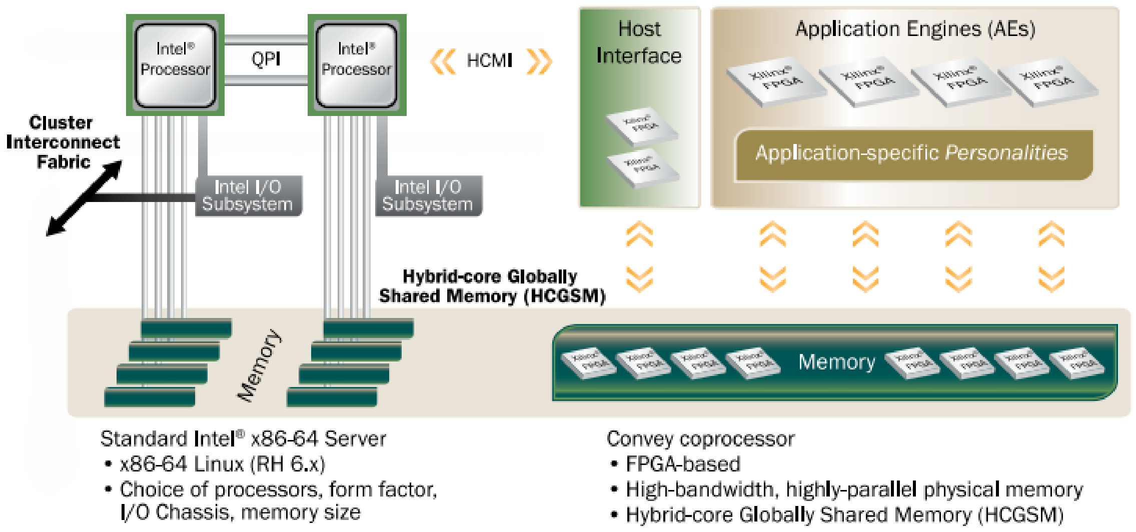
Intel® Processor
QPI
Intel® Processor
HCMI
Host Interface
Application Engines (AEs)
Xilinx® FPGA
Application-specific Personalities
Cluster Interconnect Fabric
Intel I/O Subsystem
Intel I/O Subsystem
Hybrid-core Globally Shared Memory (HCGSM)
Memory
Memory
Standard Intel® x86-64 Server
• x86-64 Linux (RH 6.x)
• Choice of processors, form factor, I/O Chassis, memory size
Convey coprocessor
• FPGA-based
• High-bandwidth, highly-parallel physical memory
• Hybrid-core Globally Shared Memory (HCGSM)

Figure 4.1 Diagram of Convey's HC-2 Platform [7]

The architecture in this paper will be designed and implemented on a Convey HC-2 platform. The HC-2 platform is a hybrid-core platform which combines the versatility of an x86 based host platform and the configurability of an FPGA based coprocessor. The system provides a high memory bandwidth for memory bound applications, and a large reconfigurable fabric for compute bound applications. In addition the memory system is globally addressable by both the host processor and coprocessor, and uses a globally coherent cache [12]. A diagram of the platform can be found in figure 4.1.

The Convey HC-2 platform allows the designer to write software in a common Convey supported programming language, such as C. The designer can then develop the required hardware architecture for the co-processor's reconfigurable fabric in a hardware definition language such as VHDL. The Convey development kit then ties the two components together by allowing the custom coprocessor routines to be called from the software.

The HC-2 platform used in this work contains 4 Virtex-5 LX330 FPGAs that are available to the developer for custom hardware. Each of these available FPGAs are known as an application engine or AE. The designer can have the same architecture duplicated across all 4 of the AEs or a unique architecture on each AE. Each AE has 8 memory controllers operating at 300 Mhz with each memory controller providing two interleaved 150 Mhz interfaces to the memory system for a total of 16 ports. The platform provides an 80 GB/s maximum bandwidth to the coprocessor memory system. The platform provides a memory crossbar system to direct memory requests to the proper controller, from any other controller. It also provides a read order queue interface, allowing all read requests to return in the same issue order.

Data can be transferred to and from the coprocessor in two ways. Firstly, since the co-processor and host processor memory is globally available, data can be exchanged using a commonly known memory location. Secondly, each AE has Application Engine General Registers, or AEGs. The host processor can pass values to an AE using these AEG registers when the coprocessor routine is called. This is useful for passing pointers to memory, size of data to process, or other values of up to 64 bits in length (each AEG is a 64 bit register). If the coprocessor routine is expected to provide a return value, the return value should be stored into the AEG specified as the return register. The coprocessor can also store any 64 bit result into any other available AEG, and then the host processor can access that AEG after the coprocessor routine has completed. Generally large data should be passed using the memory system, and small data can be passed more easily in an AEG [8] [9] [10] [11].

4.2 Implementation

The architecture developed in this paper is specifically designed to generate WPA/WPA2 Pairwise Master Keys as quickly as possible. This process is compute bound, in which the speed of the application is limited by the number of processing resources available. A compute bound application differs from a memory bound application, in that the speed of the system is limited by the throughput of the available memory bandwidth. To achieve the best performance in a compute bound application, the reconfigurable fabric provided by the HC-2 platform will need to be fully utilized and used in an efficient manner. An overview of the steps to the PMK generation process is listed below.

- Read the next PSK to create the PMK
- Using the memory interface request the next PSK
  - Wait for the memory request to complete
  - Proceed
- Start computing the PBKDF2 of the SSID and PSK
  - Requires 4096 HMAC iterations
  - The HMAC process requires 4 SHA1 iterations for this application
  - SHA1 is an 80 round operation per 512 bits of input
- Upon completion store the PMK to memory for later verification
- Repeat this process from step 1

As can be concluded from the previous steps, the PBKDF2 calculations are where the majority of the time will be spent. This warrants an architecture that can provide as much PBKDF2 computation resources as can be fit into the fabric, while allowing enough space to include the controlling logic.

4.2.1 Software

The software that supports the architecture needs to prepare the co-processor for the custom instruction before the custom call can be made. The supporting software is written in the C++ language and takes advantage of the Convey development kit. The software provides configuration information for each of the 4 AEs on the co processor, and performs the validation of PMKs to find the correct passphrase.

The first responsibility of the software is to process the provided dictionary of passphrases. This dictionary is provided as a simple text file format with each line representing an entry in the dictionary. The software runs verification on each passphrase, ensuring that the passphrase meets the length requirements and is a valid ASCII character. Each valid passphrase is then moved to the co processor memory, starting at a base memory location allocated for the Dictionary.

Once each passphrase has been verified and loaded into co processor memory, the software splits the dictionary into 4 blocks. This is done by dividing the size of the dictionary by 4, and assigning the remainder to the first AE. Memory offsets into the dictionary are then calculated, so that the hardware can be provided with the proper start location. Memory for the PMK storage is allocated in the coprocessors memory as well. This storage is divided into 4 in the same manner as the dictionary is divided.

After allocating the required memory, loading the dictionary, and calculating the offsets the software is ready to make the call to the custom coprocessor instruction. This is done by using a function provided by the Convey platform. All of the data that needs to be passed to the coprocessor is provided to this function call. This data is a memory address to start reading the dictionary for each AE, a PMK store address to start storing the generated PMKs, the number of passphrases each AE will process, the SSID, and the length of the SSID. This high level call will be translated to an assembly function that the developer is responsible for writing. Each parameter is transferred to a register in a soft processor on the co processor, and the assembly routine is called. This routine then moves data from the soft processors registers into the AEG registers onto each AE. After the required information has been fully passed to each AE, the soft processor makes the call to the custom instruction on the coprocessor.

The software on the host processor the polls on the PMK store and can start checking for PMK matches. Initially each location of the PMK store is zeroed out, so polling on a non-zero value will indicate to the checker when the next PMK is ready to check. This can be split into 4 separate threads, one thread to check each of the 4 PMK addresses assigned to each AE.

4.2.2 Top Level Attacker Architecture

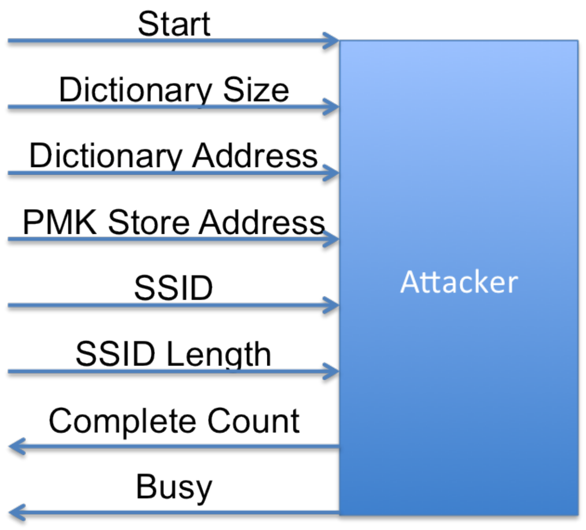

Figure 4.2 Diagram of Attacker Input and Output

In this section the top level view of the attacker architecture will be introduced. Figure 4.2shows the required input and output to the module. A full top level architecture diagram can be found in figure 4.3. The attacker modules inputs and outputs are described in detail in table 4.1. These inputs are provided from the software and passed via the AEGs mentioned in section 4.1. The attacker module will be duplicated four times, once on each available application engine on the Convey platform. This requires that the dictionary and pmk store is divided into four equal parts. If the size of the dictionary is broken into four equal parts, the first application engine handles the remainder.

It can be seen that the data traverses through the architecture in a right to left manner. Firstly, the data starts at the memory controller interfaces MCIF_0 through MCIF_7 as read operations. This data is passed through the hashing modules to produce the PMK result. Finally the PMK result is saved to the PMK store memory location via the memory controller interfaces MCIF_8 through MCIF_{11}. In order to be able to verify the generated PMKs against the handshake capture, it is important we know the passphrase that produced each PMK. This is done by maintaining a one to one mapping from the dictionary to the PMK store.

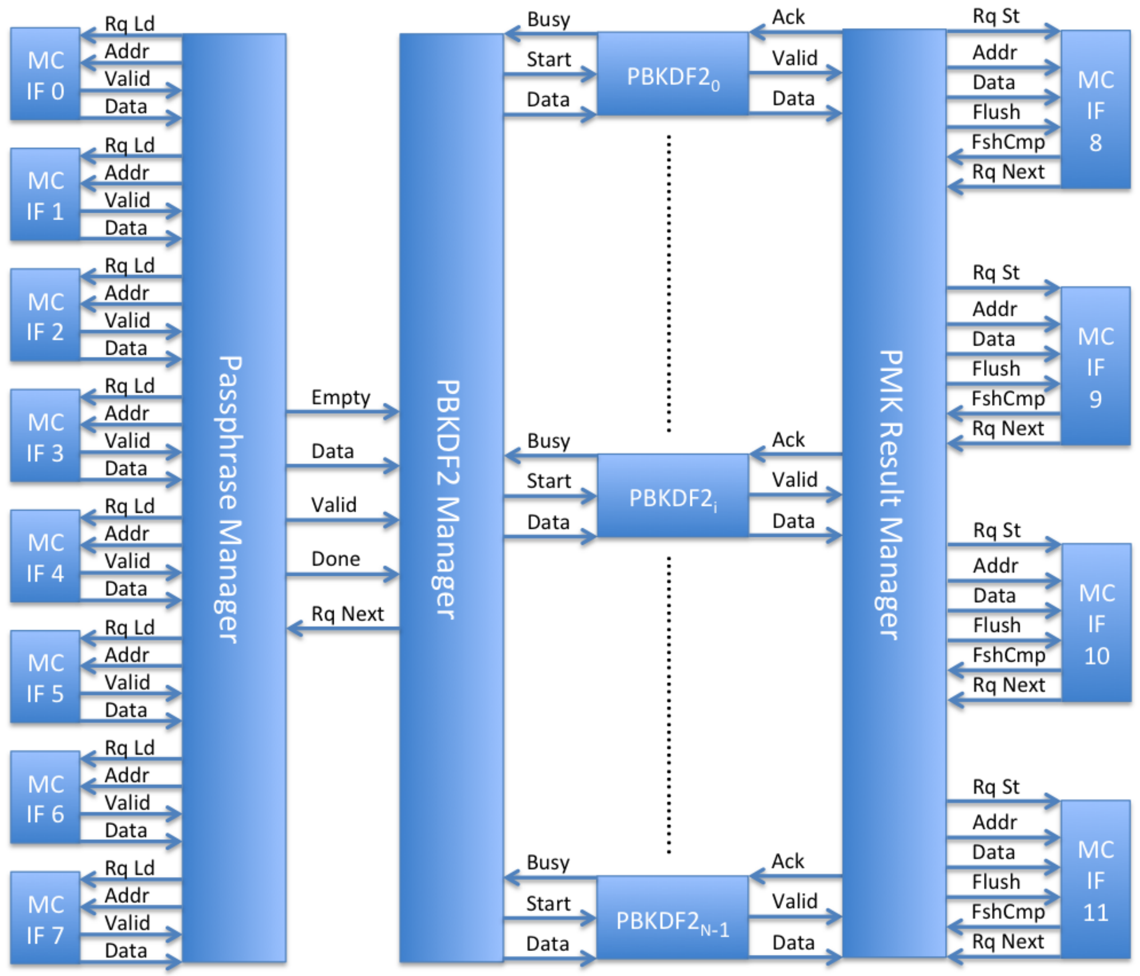

Figure 4.3 Diagram of Top Level Attacker Architecture

Port	Description
Start	This signal notifies that the software has finished setup and the hardare should start the attack
Dictionary Size	This allows the attacker module to know how many elements it is responsible for procesing, and allows it to know when it is complete
Dictionary Address	This is the base address to the dictionary that has been moved to coprocessor memory. It allows the attacker module to read the passphrases
PMK Store Address	This is the location that the attacker should start storing PMK results
SSID	This is the Secure Set Identifier of the network being attacked
SSID Length	This is the length in bits of the SSID. The hardware needs to know this length to properly pad the SHA1 message blocks
Complete Count	This is an output which allows the host processor to query the number of saved PMKs
Busy	This signal is high during the attack phase and low otherwise

Table 4.1 Attacker Input and Output Descriptions

E.g. dictionary[0] $\rightarrow$ PMKStore[0] or dictionary[N] $\rightarrow$ PMKStore[N]. Thus it is important that the data maintains an order respective to the location in the dictionary as it progresses through the hardware. The architecture guarantees this ordering by using two techniques. The first method is the use of the Convey provided read order queue [9], which keeps all memory responses ordered in the same manner issued. The second method is ensuring that all finished hashes are saved in the same order that they were dispatched. Thus, if every passphrase is dispatched to a hashing module in the same order as it was entered the dictionary, and saved in the order it was dispatched, all entries will be saved in the same order.

The first part of the attack process is to request the next passphrase from the dictionary. This is controlled by the passphrase manager module which can be seen in the diagram. This module has access to 8 memory controller interfaces. Each memory controller has a read/write width of 64 bits. Since a passphrase will contain a maximum of 512 bits, 8 memory controllers will allow an entire passphrase to be requested on the same cycle. The passphrase manager will make a request for the next passphrase, and increment a counter which tracks the total number of dictionary entries that have been read. The manager contains 8 FIFOs to handle the responses from the memory controllers, which can provide responses on different moments. By using the optional read order queue provided by Convey [9], all responses will be returned in the same order they were requested. Thus, when all 8 FIFOs are non-empty, the data at the front of each FIFO can be pieced together in the proper order to form the 512 bit passphrase. When a passphrase is ready, the empty signal is de-asserted, and the PBKDF2 Manager will be able to start issuing passphrases to open PBKDF2 modules. The passphrase manager will continue to request the next available passphrase until the FIFOs become full, or the read count is equal to the size of the dictionary.

The second part of the attack is where the hashing takes place. The PBKDF2 manager is responsible for dispatching each passphrase to an open PBKDF2 module to generate the PMKs. The module dispatches each passphrase in a round robin order to each hashing module. This allows the architecture to maintain the correct order. Each PBKDF2 module has a busy flag, which allows the PBKDF2 manager to verify if a module can accept a new passphrase. When the next hashing module to start indicates it is not busy, the manager dispatches the passphrase and asserts the start signal for this module. The manager will then proceed to the next available passphrase and hashing module, until all modules indicate they are currently busy. Each hashing module will finish in the same order that they were started, since the runtime to generate each PMK is constant.

Finally, the result manager is used to store each PMK to the coprocessors memory using the memory controller interfaces MCIF_8 through MCIF_{11}. As each PBKDF2 hashing module completes and drives the valid signal high, the manager will store the result in a round robin manner. The round robin operation allows the architecture to keep the required ordering, as previously mentioned. The result manager then increments the stored PMK count and sends an acknowledge to the finished hashing module. The acknowledge is used to inform the hashing module that it may overwrite the result register if needed. The hashing module will still be able to proceed hashing the next passphrase, it is just important that the hash result and valid signal remain unchanged until the acknowledge is provided. This is because any of the memory controller interfaces used for storing the results, may non-deterministically request a stall for all writes to that interface. A stall here will also cause the result manager to also halt its round robin operations. Thus, if the hashing module modified the valid or result signals before the acknowledge occurs, the PMK would be lost. In general, a memory interface stall will only last for a fraction of the time required to generate the next PMK. So a stall here should not cause many stalls of the hashing process, which would degrade the performance.

4.2.3 PBKDF

This section will focus on the architecture of the PBKDF2 component. PBKDF2 algorithm can be found in section 3.2.3. The PBKDF2 architecture is duplicated across the design with the goal of utilizing all available resources on the FPGA. Figure 4.3 shows where the PBKDF2 architecture is placed in the global architecture and figure 4.4 shows the PBKDF2 implementation itself.

The PBKDF2 module requires the inputs shown in table 4.2. Notice that there are two HMAC computations that are run in parallel. This is possible because the first N bit block of the final 256 bit result can be computed without dependencies on the rest of the N bit blocks. In this case, two 160 bit HMAC results are computed and appended together, to form the final 256 bit PMK value.

Table 4.2 PBKDF2 Module Parameters

Parameter Name	Parameter Usage
Passphrase	Passphrase under test
SSID	SSID of network under attack
SSID Length	Lenght of the SSID required for preparing the message
Start	Informs the logic to begin the PBKDF2 computation

The first round of the algorithm needs the value of the SSID concatenated with a one byte value as an input for the T port. This one byte value represents which part of the 256 bit output will be generated. Since a 160 bit underlying hash function was selected, this value will be 0x01 or 0x02. This concatenation is represented by the blocks labeled SSID || 0x0i. The actual implementation of this is done by shifting the byte representation of i, by the length of the SSID, and then using an N bit or operation to combine the two, see figure 4.5. For the other 4065 rounds, the T input is taken from the previous hash result.

The computation time for each HMAC operation will be static, since the message size to each HMAC module is fixed. This allows one of the two HMAC valid signals to be used to enable the round counter. When the valid signal is asserted, the count is increased, the U_0 and U_1 registers are updated with the new HMAC result. The round count is then checked to determine if the logic should report a valid result, or continue with the hashing process. If all rounds are completed, the entire 160 bits of U_0 and the top 96 bits of U_1 are appended together to form the final result, and the valid signal is then driven high.

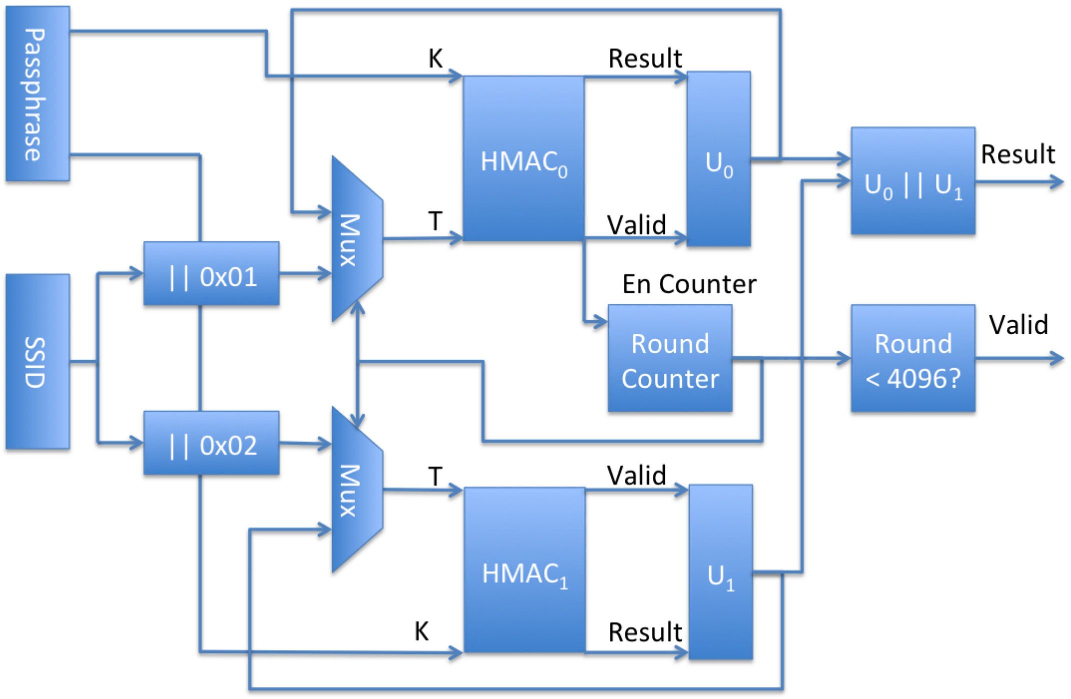

Figure 4.4 Diagram of PBKDF2 Module

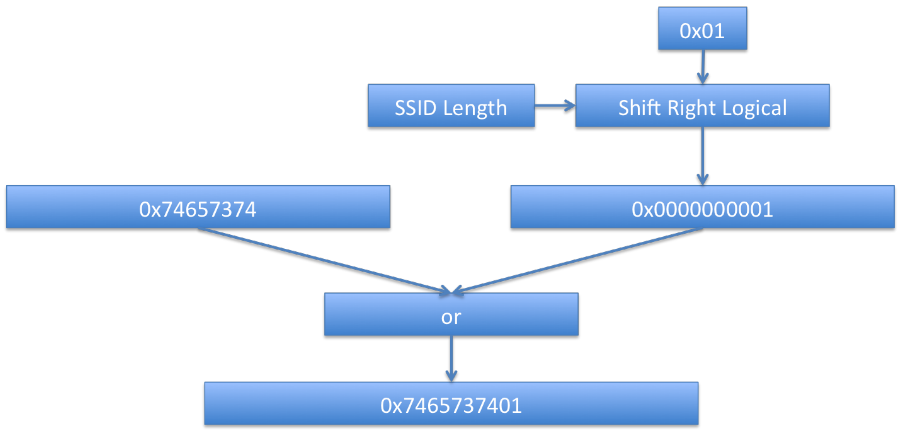

Figure 4.5 Diagram of Round One T Generation Using "test" as the SSID

4.2.4 HMAC

This section will introduce the logic for the HMAC computation, section 3.2.2. The HMAC module relies on SHA1 as the underlying hashing algorithm. The attack will only ever require a maximum of 841 bits to be hashed for any SHA1 message. This value is computed by considering that the input to the HMAC function will either be a 160 bit HMAC result concatenated with an 512 bit value (iPad/oPad xor passphrase), or it will be an 512 bit value (iPad/oPad xor passphrase) concatenated with the SSID concatenated with the 1 byte representation from the PBKDF2 requirements. Both cases will also include a binary 1 bit appended to the end of the messages data, and require the last 64 bits of the message to contain the entire length of the message. This bit distribution can be found in table 4.3.

Table 4.3 Size of SHA1 Message Constructed by HMAC

Case	Bit Distribution	Total Size
Pad + SHA1 + Binary 1 + Length	512 + 160 + 1 + 64	737 bits
Pad + SSID + 1 Byte + Binary 1 + Length	512 + 256 + 1 + 64	841 bits

Since the SHA1 must operate on block of 512 bits, both of the above cases require the message to be zero padded out to 1024 bits before the hashing operation begins. This allows us to design an HMAC module that only needs to handle preparing two blocks of data for the SHA1 module. The HMAC module relies on a finite state machine to control the flow of the hardware. A diagram of this state machine can be found in figure 4.6. The diagram flows in a clockwise direction, with different shades to represent what each state is used to compute. The purple shaded states represent the states that the HMAC module will enter when either waiting on an instruction to start the next computation, or when the module has completed a computation. The orange states are used to compute the first round of the HMAC computations. The red states are used to compute the second round of the HMAC computations.

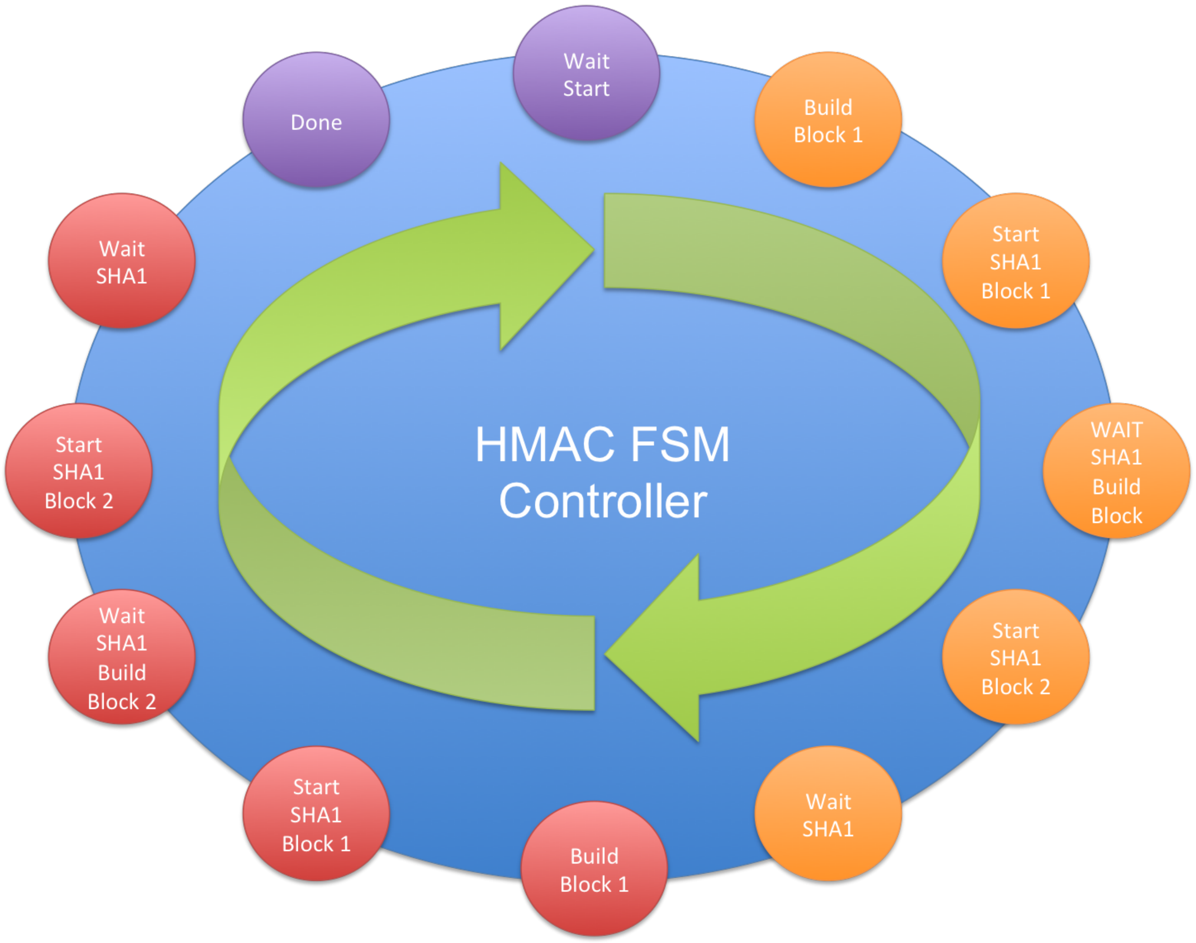

Figure 4.6 Diagram of HMAC FSM

The first block is created from the operation of $iPad \oplus k$. The 512 bit result is then used as the first block to the SHA1 module. While the first block is being hashed, the second block of the first HMAC round is computed. This block is assembled in a separate register, and then used as input to the SHA1 module when the SHA1 computation is complete. The second block is assembled by first inserting the SSID into the upper bits of the assembly register. A binary one is then shifted right by the length of the SSID, and combined into the assembly register using an or operation. Finally the lower 64 bits are assigned the value of the total length of the message. The message length is computed by adding the the 512 bits of the $iPad \oplus k$ operation to the length of the SSID. The bits added to the message block through the inclusion of the binary 1 append and the 64 bit message length, are not considered part of the original message, and thus not included. During the second round, it is easier to generate the two blocks, as the sizes of these blocks are fixed. The first block is just the 512 bit result of $oPad \oplus k$. The second block is the 160 bit SHA1 result from the first round appended with the binary 1, and the length set into the lower 64 bits. Thus the length will always be 672 bits, derived from 512 bits + 160 bits, and as a result no shifting or addition operations are required to assemble this block.

4.2.5 SHA1

The SHA1 implementation will be broken into an 5 stage pipeline in order to meet timing requirements of the reconfigurable fabric. However, each stage of the pipeline will not produce a valid result on each clock cycle, like most pipeline are designed to do. This is due the data dependencies of SHA1, which has a dependency on the previous rounds result before the next round may begin. The implementation can be found if figure 4.7.

During rounds 0-15, the Wt value is simply computed by choosing the Nth 32 bit word from the 512 bit message block. In rounds 16-79 the computation requires that the previous 16 Wt values are also made available. In order to achieve this, the Wt values are sent through a 32 bit wide by 16 word deep shift register. This shift register provides read ports on the 3rd, 8th, 14th, and the 16th word of the register. These values are then passed to an XOR gate, and then rotated 1 bit to the left to produce the next Wt value. The new Wt value is then routed to the input of the Wt shift register and used in the addition operation.

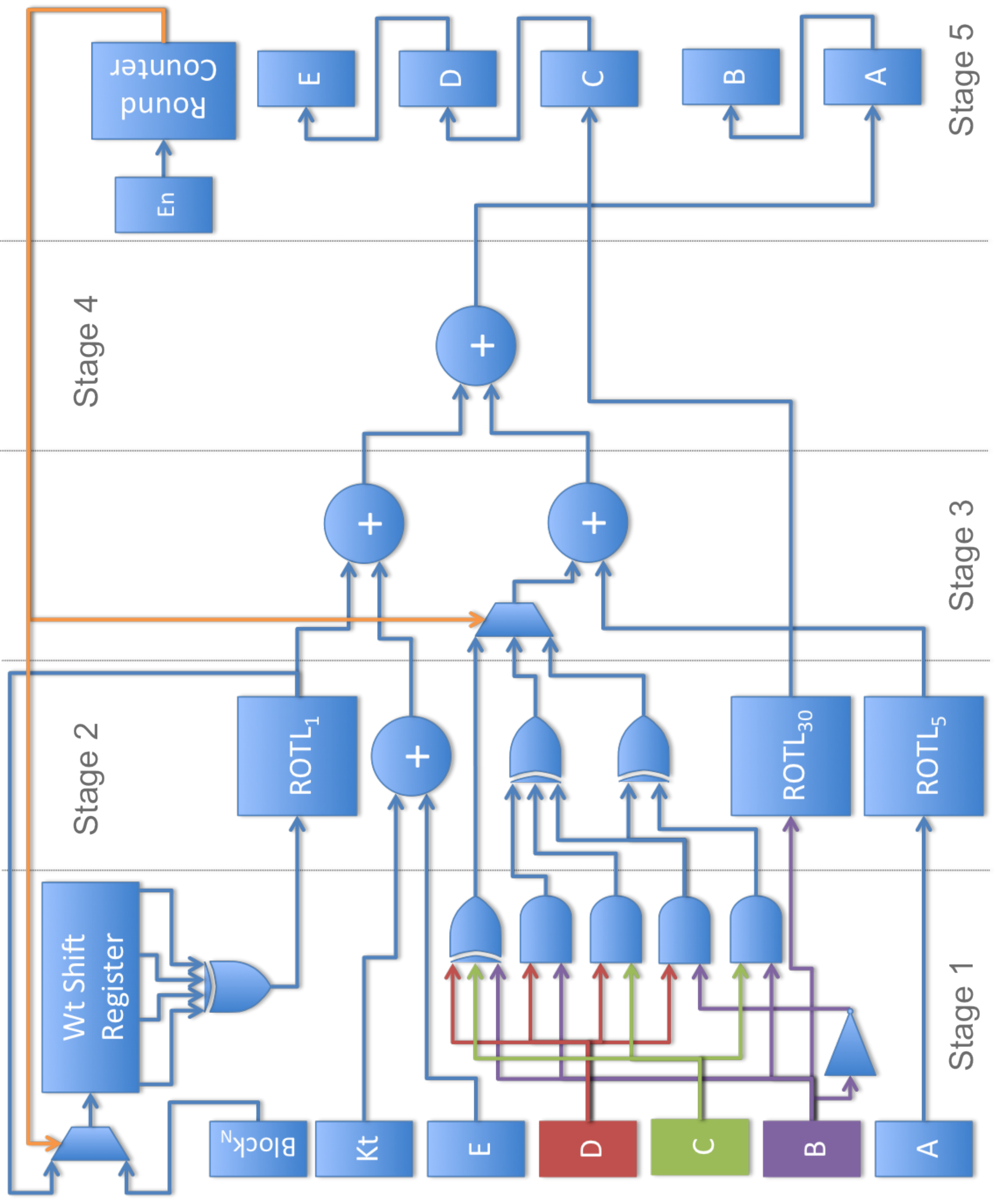

Figure 4.7 Diagram of the SHA1 Module

The Kt value is a constant base off of the values in table 3.3. The Kt register is maintained by assigning one of the 5 constant values, based on the current round count. The Kt driving logic is replaced by just the Kt register in the diagram, for reasons of simplicity.

The F function, algorithm 3, computes three different values based on the current round count. All possible outputs are computed, and then the required output is selected by evaluating the round count. The F function logic is spread across two stages. The first stage is used to calculate the following:

- $B \wedge C$
- $B \wedge D$
- $C \wedge D$
- $\neg B \wedge D$
- $B \oplus C \oplus D$

The second stage then calculates:

- $ROTL(A, 5)$
- $ROTL(B, 30)$
- $(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$
- $(B \wedge C) \oplus (\neg B \wedge D)$

Stage 3 and stage 4 are used to add the intermediate values required to compute the next A register value. Finally, stage 5 is used to set the next values of the A,B,C,D, and E registers. The A register is assigned the summation results calculated in stage 3 and 4. The B register is assigned the previous A register value. The C register is assigned the previous B register value rotated 30 bits to the left. The D register is assigned the previous C register value. The E register is assigned the previous D register value. Stage 5 also increments the round counter by one and enables the Wt shift register in order to shift in the new Wt value and to shift the 16 words one position.

The SHA1 controlling logic implemented as an FSM is not shown in the diagram. This logic provides the following functionality. Firstly, the module will wait to start computing until the containing logic informs the module to start computing the hash. Once the module is instructed to start, it will start computing the hash. Upon completion of the 80 rounds, the A-E registers are added to the H0-H4 registers to produce the both the new A-E register values and H0-H4 values. If the the completed block is the first block of the message, the module will indicate that it now requires the second block of the message, and wait to receive the new block to proceed. If it was the second block of the message, the module will provide the SHA1 hash result and inform the container it is complete. The module will then proceed to wait for the next start signal.

CHAPTER 5. Results

5.1 Results

This chapter will provide the results that were achieved with the architecture introduced in this work. We will start by providing the raw results for number of PMKs generated by our architecture. We will then provide some theoretical analysis on what could be achieved based on various factors such as clock rates, reconfigurable fabric resources, and the selected SHA1 implementation. We will the provide analysis of our design against current implementations.

The speed at which a PMK can be generated is directly dependent on how long it takes to compute a PBKDF2 result. The time requirement for the PBKDF2 computation is dependent on the speed of the HMAC computations. Finally, the HMAC run time is dependent on the SHA1 computation requirements. As presented in section 4.2.5, the SHA1 is broken into 5 separate stages per each round. Thus, to compute a single SHA1 block, it takes 80 rounds times 5 cycles per round for a total of 400 cycles.

In the application of WPA/WPA2 PMK generation, the size of the HMAC message will be equivalent to the size of two SHA1 blocks. This is calculated by considering that a SHA1 block is 512 bytes, and our longest HMAC message is 833 bits as described in section 4.2.4 and table 4.3. Each HMAC message, after being zero padded, will be 1024 bits in length to meet the SHA1 block requirements. Thus, for each HMAC message, there are 2 SHA1 blocks to be hashed times the 400 cycles per block, for a total of 800 cycles per HMAC message.

The PBKDF2 computation uses an HMAC of an HMAC computation for each round. So each round of the PBKDF2 algorithm will require 800 cycles times 2 HMAC computations, resulting in a total of 1600 cycles. Finally, the PBKDF2 algorithm is repeated 4096 times, requiring a total of 1600 cycles times 4096 rounds which results in a total of 6553600 cycles for a single PMK computation. Using this calculation, table 5.1 provides the PMK per second calculations for the architecture, using the two possible AE clock frequencies.

Table 5.1 PMKs per Second

Clock Rate	Cycles/PBKDF2 Calculation	PBKDF2 Cores	AEs	PMK/sec
150 Mhz	6553600	10	4	915.5273438
300 Mhz	6553600	10	4	1831.054688

We will now provide a comparison of our implementations performance against other current implementations of the key recovery. We have selected the implementation Pyrit [3] to provide context with regards to CPU and GPU performance. And we have select the Open Ciphers project [15] to provide context against an FPGA platform. The comparisons can be seen in table 5.2. As can be seen, the current state of our implementation does not directly compete with the GPU options, and only marginally performs better than the CPU and FPGA implementations. In the following discussion, we will provide further analysis of our implementation to see if it can become more competitive.

Table 5.2 Comparison Against Other Implementations [3] [15]

Implemntation	PMK/sec
Convey Based Architecture	1831
Pyrit on CPU	1300
Pyrit on 1x GPU	19500
Pyrit on 4x GPU	89000
OpenCiphers	1000

To show the importance of the SHA1 computation time, figure 5.1 shows the overall impact on the PMKs per second that the SHA1 component has. The cycles per round for the SHA1 module assume implementations that range from 1 to 5 cycles for each round of computation. The 5 round version is what is in used in this architecture, but is a naive implementation of the SHA1 algorithm. An implementation requiring 1 cycle per round can be found in [24]. There are also commercial implementations available for purchase claiming 1 cycle per round. The implementations considering 2 cycles per round to 4 cycles per round are assumed and inserted for additional reference.

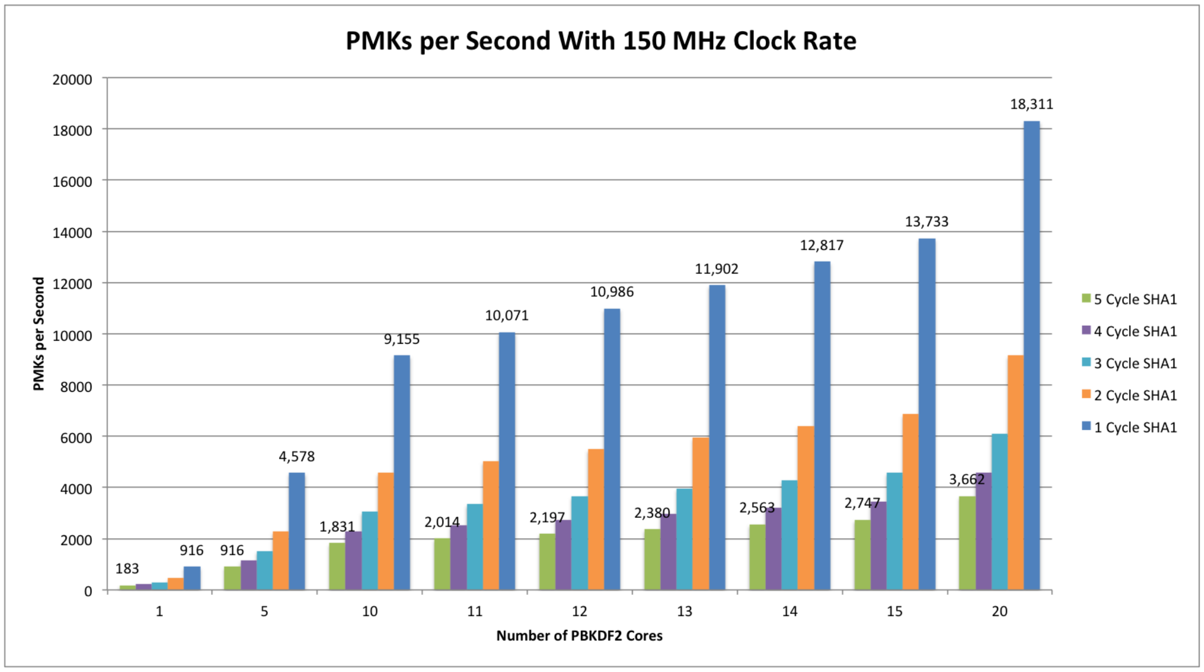

Figure 5.1 Graph of SHA1 Implementation vs PMKs per Second at 150 MHz

It can be seen that seen in figure 5.1 that by moving from a SHA1 implementation requiring 5 cycles per round to an implementation that requires 1 cycle per round, we can provide a 5 times greater improvement in performance. The architecture is currently targeting the 150 MHz clock offered by the Convey platform. The Convey platform also offers a 300 MHz clock rate for the user design. Let us now look at the effect of doubling the clock rate in figure 5.2. As can be seen, doubling the clock rate for the user design will provide a 2x improvement in the overall PMK/second figure.

Finally, let us analyze the efficiency of the designs fabric usage. The usage reports for the designs current state can be found in table 5.3. This table shows that there is about of the reconfigurable fabric consumed. Out of this 2/3 usage, the Convey platform is consume 1/3 for the memory interfaces. This indicates that our design is consuming about 1/3 of the available space as well. In turn, these numbers show that there is about 1/3 of the overall

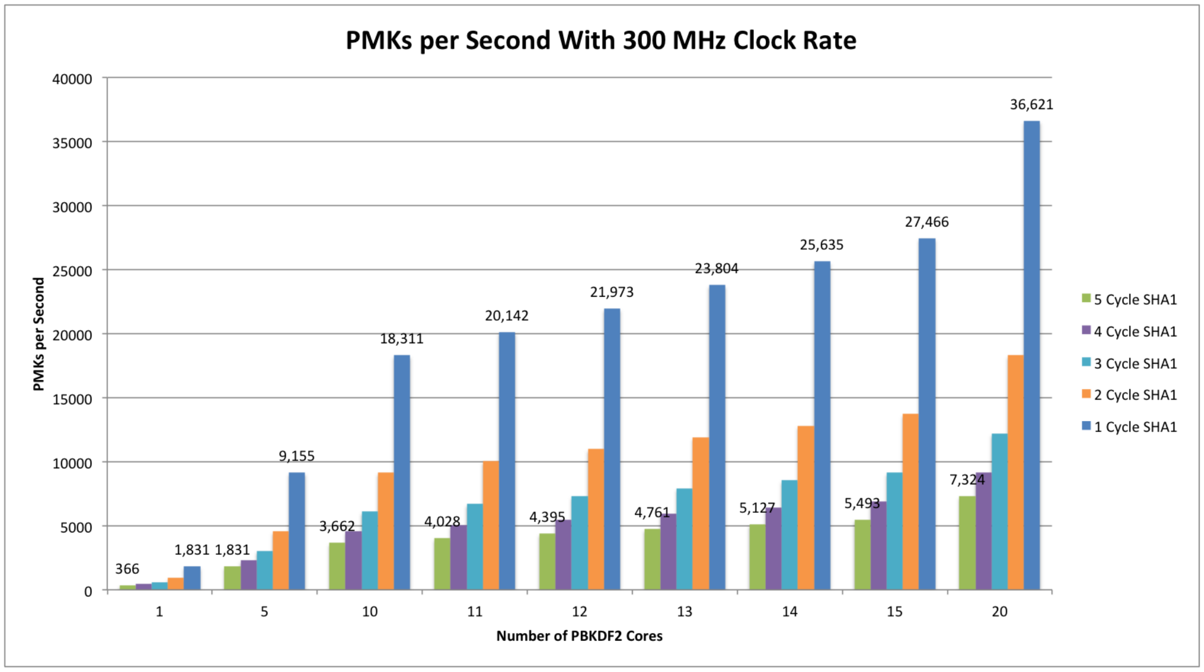

Figure 5.2 Graph of SHA1 Implementation vs PMKs per Second at 300 MHz

fabric remaining. Ignoring routing and timing issues that become more difficult to deal with as a design becomes denser, we could theoretically double the current density of compute cores. This speculation is also shown in figure 5.1 for the 150 MHz clock rate, and again in figure 5.2 for the 300 MHz clock rate. See the analysis with the 20 PBKDF2 computation cores.

Table 5.3 Usage Report

Term	Percent Usage
Number of Slice Registers	68
Number of Slice LUTs	61
Number of occupied Slices	92
Number with an unused Flip Flop	17
Number with an unused LUT	25
Number of fully used LUT-FF pairs	56

Now let us compare the available fabric for a Virtex-5, which is an older technology, to a newer Virtex-7. The Convey system is using the most dense Virtex-5 chip offered by Xilinx, so let us also choose the most dense Virtex-7 chip for this analysis. Table 5.4 shows the number of slices that each chip has to offer along with the number of flip-flops and look up tables for each device.

Table 5.4 Virtex Slices by Chip [25] [26]

Model	Slices	FF per Slice	LUT per Slice
Virtex-5 LX330	51840	4	4
Virtex-7 2000T	305400	8	4

As can be seen in table 5.4, the Virtex-7 offers a 5.8 times larger reconfigurable fabric for our design to occupy. The Virtex-7 slice also has twice the number of flip-flops as the Virtex-5, taking us from being flip-flop bound to lookup table bound. Let us now analyze the gains that could be acquired by moving to the newer Virtex-7 technology. Figure 5.3 and figure 5.4 show the 150 MHz and 300 MHz clock rates respectively, with the calculations of a 5 times larger fabric in terms of number of compute cores.

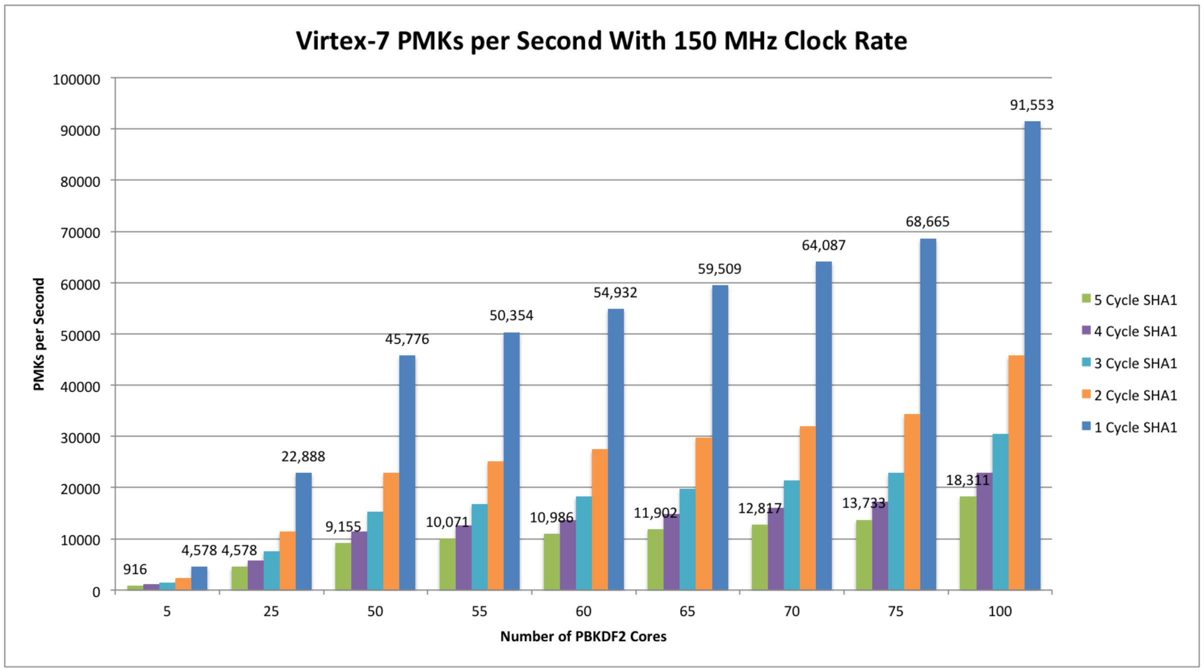

Figure 5.3 Graph of SHA1 Implementation vs PMKs per Second at 150 MHz on a Virtex-7

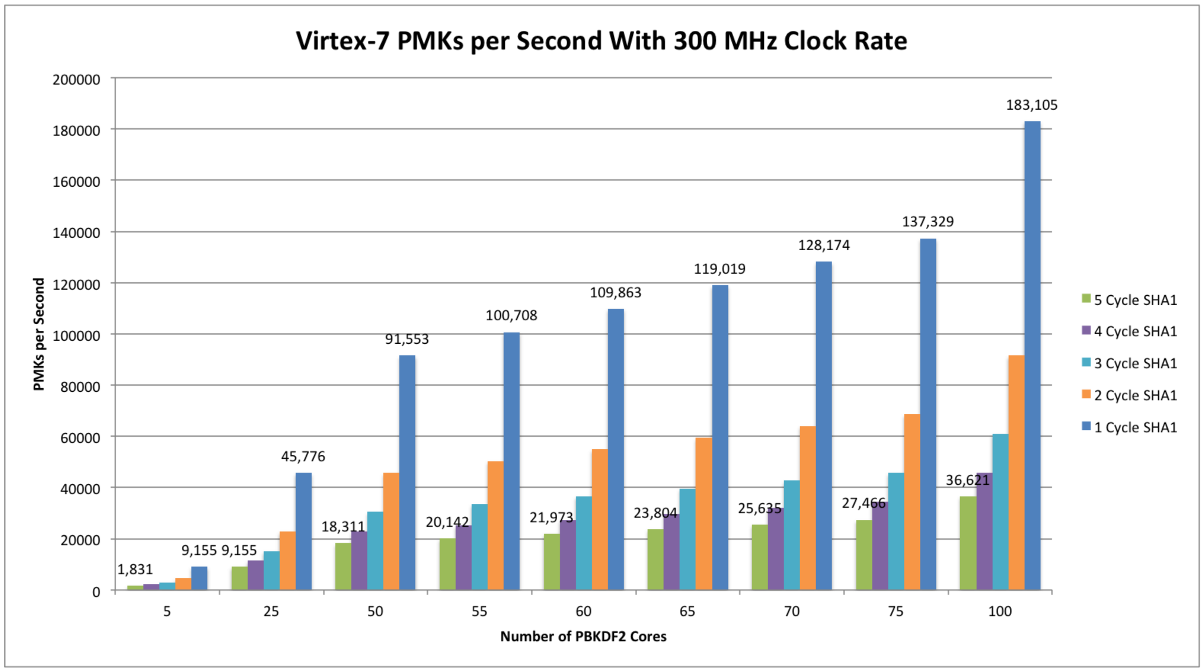

Figure 5.4 Graph of SHA1 Implementation vs PMKs per Second at 300 MHz on a Virtex-7

CHAPTER 6. Conclusion

6.1 Summary

This work has introduced an expandable architecture targeted for an FPGA based platform, with the purpose of recovering WPA/WPA2 passphrases. We provided a brief history of wireless LAN, reviewed the dictionary based attack to recover the passphrase, showed an analysis on the attack space, introduced our architecture, and finally analyzed the performance of the system. The results show that currently the design may not perform as well as GPU implementations, but that with further optimizations our architecture can become competitive with these implementations.

6.2 Future Work

We have reviewed how the performance is highly depended on the chosen implementation of the SHA1 computation core. The design in its current state is not competitive, and a new SHA1 implementation should be sought out in future continuations of this work. This selection should target the 1 cycle per round implementation as discussed. Another interesting approach could be to interleave several independent SHA1 message blocks into the SHA1 pipeline. This would require some register duplication, but would allow each stage of the SHA1 pipeline to have work, essentially providing a 5 times improvement if implemented on the current design.

We also showed that we can double the performance by using the 300 MHz clock rate offered by the Convey platform. This should be the next approach to improve the performance of the system. Targeting a higher device utilization was also shown to have a dramatic effect on the production of PMKs, and can be a viable improvement on future iterations.

Finally, we showed that the Virtex-5 LX330 is an older technology, and that the current high end Virtex-7 chip offers an almost 6x larger reconfigurable fabric. While the Convey platform is designed around the Virtex-5 LX330, future Convey platforms may offer higher density options, and would provide a large impact on performance.

BIBLIOGRAPHY

[1] Aircrack. http://www.aircrack-ng.org/. [Online; accessed December-2013].

[2] cowpatty. http://www.willhackforsushi.com/Cowpatty.html. [Online; accessed December-2013].

[3] pyrit. https://code.google.com/p/pyrit/. [Online; accessed December-2013].

[4] Abramson, N. (1970). The aloha system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), pages 281–285, New York, NY, USA. ACM.

[5] Abramson, N. (1985). Development of the alohanet. *Information Theory, IEEE Transactions on*, 31(2):119–123.

[6] Alliance, T. W.-F. The history of wi-fi. https://www.wi-fi.org/sites/default/files/History_of_Wi-Fi_201301.pdf. [Online; accessed December-2013].

[7] Corporation, C. C. Convey hc product brochure. http://www.conveycomputer.com/files/3613/5085/4052/Convey_HC-2_Product_Brochure.pdf. [Online; accessed December-2013].

[8] Corporation, C. C. Convey pdk fpga design practices. http://www.conveysupport.com/alldocs/WhitePapers/PDK/WP_FPGA_Design.pdf. [Online; accessed December-2013].

[9] Corporation, C. C. Convey personality development kit reference manual. http://www.conveysupport.com/alldocs/ConveyPDKReferenceManual.pdf. [Online; accessed December-2013].

[10] Corporation, C. C. Convey programmers guide. http://www.conveysupport.com/alldocs//ConveyProgrammersGuide.pdf. [Online; accessed December-2013].

[11] Corporation, C. C. Convey reference manual. http://www.conveysupport.com/alldocs/ConveyReferenceManual.pdf. [Online; accessed December-2013].

[12] Corporation, C. C. Personality development kit (pdk) for convey hybrid-core computers. http://www.conveycomputer.com/files/1313/5085/4646/PersonalityDevelopmentKit.pdf. [Online; accessed December-2013].

[13] Dictionaries, O. How many words are there in english? http://www.oxforddictionaries.com/us/words/how-many-words-are-there-in-the-english-language. [Online; accessed December-2013].

[14] Edney, J. and Arbaugh, W. A. (2003). *Real 802.11 Security*. Addison Wesley, Reading, Massachusetts.

[15] Hulton, D. (2006). The openciphers project. http://openciphers.sourceforge.net/oc/. [Online; accessed December-2013].

[16] IEEE. Official ieee 802.11 working group project timelines. http://grouper.ieee.org/groups/802/11/Reports/802.11_Timelines.htm. [Online; accessed December-2013].

[17] Kaliski, B. (2000). PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational) http://www.ietf.org/rfc/rfc2898.txt.

[18] Krawczyk, H., Bellare, M., and Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) http://www.ietf.org/rfc/rfc2104.txt. [Online; accessed December-2013].

[19] Lemstra, W., Hayes, V., and Groenewegen, J. (2010). *The innovation journey of Wi-Fi: the road to global success*. Cambridge Univ Pr.

[20] Merriam-Webster. How many words are there in english? http://www.merriam-webster.com/help/faq/total_words.htm. [Online; accessed December-2013].

[21] National Institute of Standards and Technology (2012). *FIPS PUB 180-4: Secure Hash Standard*. Supersedes FIPS PUB 180-3.

[22] Openwall. John the ripper password cracker. http://www.openwall.com/john/. [Online; accessed December-2013].

[23] Renderlab. Church of wifi wpa-psk lookup tables. http://www.renderlab.net/projects/WPA-tables/. [Online; accessed December-2013].

[24] Satoh, A. and Inoue, T. (2007). Asic-hardware-focused comparison for hash functions md5, ripemd-160, and shs. *Integration*, 40(1):3–10.

[25] Xilinx (2009). Virtex 5 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. [Online; accessed December-2013].

[26] Xilinx (2013). Virtex 7 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [Online; accessed December-2013].