Message-Digest 5 (MD5) Hash Reversal System

Gareth Callanan[†],Jean Swart[‡]and Matthew G Smith[§]
EEE4084F Class of 2016
University of Cape Town
South Africa
[†]CLLGAR010 [‡]SWRJEA005 [§]SMTMAT015

***Abstract*—This paper describes the design and testing of a device created to reverse the effects of the Message-Digest 5 (MD5) Hash Function in a massively parallel manner. The device will be used to find the original data used to generate the 128-bit MD5 hash. The system is initially tested on a Nexys 4 FPGA platform with the understanding that the project can later be implemented on an ASIC platform for massively improved performance.**

It was found that the prototype FPGA system resulted in significantly faster hash reversal than the golden measure, a parallel system running on a CPU. This result was obtained using only one solving module and can be expanded in parallel.

I. Introduction

This report deals with the overview and design of an application-specific integrated circuit (ASIC) based Message-Digest 5 (MD5) hash reversal system. The MD5 algorithm is a hashing function which produces a 128-bit hash value for an arbitrary length input through an extensive series of bit-wise combinatorial operations.

As of 2012 [1] the MD5 algorithm is no longer recommended for use in cryptographic applications but remains widely used for other purposes such as data integrity checking in the form of check-sums [2].

This topic requires a parallel hash reversal system to be implemented on a platform of choice. In this context, this means that the system should be able take in a MD5 hash of unknown origin and output the word, phrase or characters that generated that hash.

The MD5 hash-reversal device can be used for password recovery. If the MD5 hash of a password is known, the device will be able to quickly find the corresponding password using a brute-force attack.

Due to the fact that the MD5 hash system is a one-way function and not true encryption or encoding, it cannot be decrypted. A MD5 hash can only be reversed through methods such as brute-force or dictionary attacks where the reversal system tests all possible character combinations or all words in a given list to see if they generate a MD5 hash matching the input hash. This makes for an extremely coarse-grained problem which is ideal for parallelization, as numerous solver modules can be implemented with no intercommunication.

A dictionary attack can be a highly useful tool for reversing hashes which were generated with large words that could be found in a standard word-list. However, a parallel brute-force reversal system with a high enough clock speed will be fast enough for all words with a length close to that of the average password, 8 characters. [3]

It was chosen to design the reversal system using an Artix-7 field-programmable gate array (FPGA) and later transition to a high-speed ASIC platform. These platforms were chosen over other parallel platforms such as a multi-threaded CPU due to their ability to handle many more parallel processing instances. The coarse granularity of the problem lends to the idea that the more parallel instances the system can handle, the faster it can reverse a given hash.

This report is divided into an overview section detailing the plan of implementation at a high level of abstraction, a detail design section explaining the inner workings of the system, a prototype design section explaining the simplifications and compromises made in order to implement the system on an FPGA and a conclusion section.

II. Overview

A. User Interaction

This device is not designed to be a stand alone device and requires an interface with a PC running either Windows or Linux. The PC will send commands and data to the ASIC, including the MD5 hash to reverse and reversal parameters. The ASIC will then perform the brute-force attack to find the characters which generated the input hash and send the data back to the PC. All communication will be performed over an RS232 serial link. A faster link was not necessary as data transfer is expected to be a very small portion of the overall execution time. A block diagram showing the high-level system structure is shown in Figure 1. A detailed block diagram of the system is shown in Figure 2. This diagram shows the internal structure of the system and how the various modules interface with one another. The solver worker modules are each given an index, `i`, where $0 < \texttt{i} < \texttt{N}$, with `N` being the number of parallel solver workers in the system.

The PC will use custom software with either a command line or graphical user interface, allowing the user to issue commands to the ASIC and providing feedback based on status information send back by the ASIC.

Due to the fact that MD5 uses an arbitrary length input, the ASIC could potentially take a long time to find the original text data which generated the hash, causing the user to become impatient. In order to alleviate this, the user will be provided with periodic feedback about the number of combinations that have been attempted.

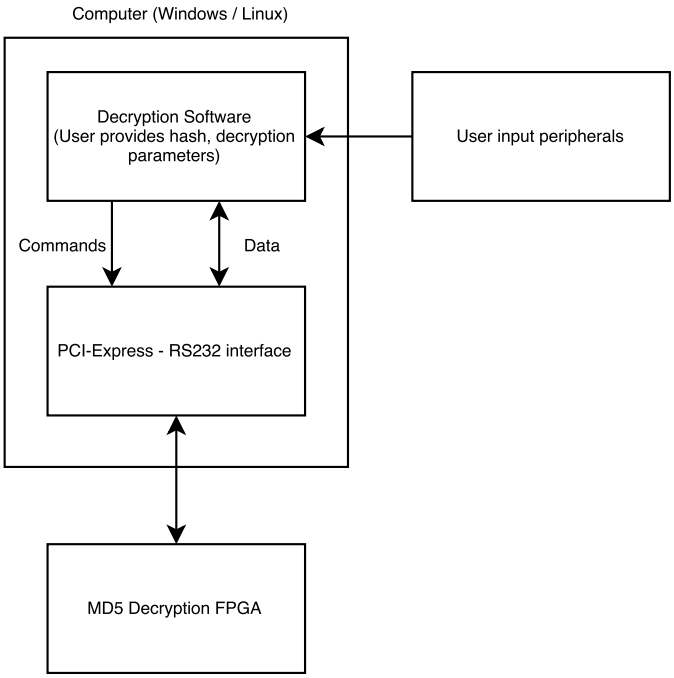

Fig. 1. High level block diagram for the system

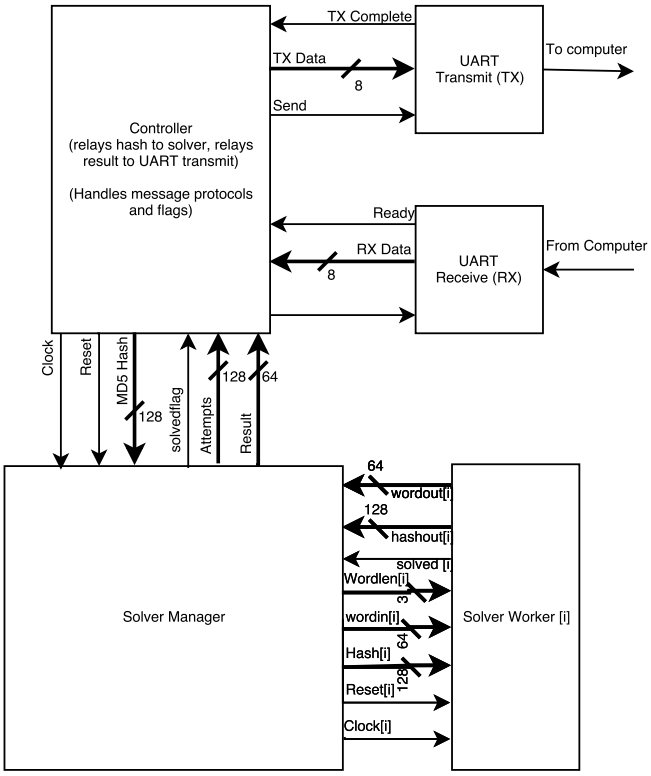

Fig. 2. Detailed block diagram of the system

B. High-Level System Structure and Operation

As stated earlier, the high-level block diagram for this system is shown in Figure 1.

After receiving the MD5 hash to be reversed, a brute-force attack will begin. After every attempt in the brute-force process the resulting hash will be compared to the input hash. If they do not match, the user will be shown the number of attempts and the next attempt will begin. When a match is found the user will be shown the original message that produced the hash and the total number of attempts that were made at finding this data.

C. MD5 Algorithm

This section serves to explain the basics of how the MD5 algorithm operates and how it could be implemented in sequential and parallel forms.

1) Message Padding:

The MD5 algorithm begins by padding the input message so that its length is divisible by 512. A binary 1 is appended to the end of the message, followed by zeroes until the length of the message is 64 bits less than a multiple of 512. The remaining 64 bits are given as the length of the original message, modulo 2^{64}.

2) Message Partitioning:

The padded input message is split up into blocks of 512 bits each, with each block being made up of sixteen 32-bit words.

3) State Modification:

The output of the MD5 algorithm is a 128-bit state which can be divided into 4 words of 32 bits each. Let these words be $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$. Initially, these words are set to fixed constants, and these constants are modified using the sixteen 32-bit words of each 512-bit message block.

The modification for each message block occurs in 64 "rounds", with a different combinatorial function, $\mathbf{F}$, being used every 16 rounds. The different combinatorial functions in order of their use are:

$$F_1 = (\mathbf{B} \wedge \mathbf{C}) \vee (\overline{\mathbf{B}} \wedge \mathbf{D}) \tag{1}$$

$$F_2 = (\mathbf{B} \wedge \mathbf{D}) \vee (\mathbf{C} \wedge \overline{\mathbf{D}}) \tag{2}$$

$$F_3 = (\mathbf{B} \oplus \mathbf{C} \oplus \mathbf{D}) \tag{3}$$

$$F_4 = (\mathbf{C} \oplus (\mathbf{B} \vee \overline{\mathbf{D}})) \tag{4}$$

These functions are used to generate a value which is added to a certain 32 bit word of the message block and rotated to form a new value of $\mathbf{B}$. The other words of the 128-bit state are rotated such that $\mathbf{A} = \mathbf{D}$, $\mathbf{C} = \mathbf{B}$ and $\mathbf{D} = \mathbf{C}$. These new values are added onto the original values of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ to

form the new 128-bit state. This process is repeated for each 512-bit chunk and the output is reconstructed in little-endian from $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$.

D. Experimental Procedure

A simple parallel C++ program was created as a golden measure. This was done for two reasons, the first of which being to see which components of the of the solving algorithm could be made parallel and the second being to obtain an estimate of the time it should take to complete the solving algorithm. The algorithm used to achieve this can be seen in the form of a flow chart in Figure 3. The same algorithm was implemented on all threads of the CPU. The found flag referred to in the diagram is a global flag shared between all threads that will be set high when any of the threads generates a hash matching input hash. Each thread will generate a hash for the current word in the brute force attempt and then check the found flag to see if any of the other threads generated a successful result before continuing.

The program was run on 16 threads on a an AMD FX 8350 and used an existing MD5 encryption library [4]. It was tested on strings from " " to "$\sim\sim\sim$" and only looked at characters with ASCII values from 32 to 126 (i.e. " " and "$\sim$") as this range represents all characters on the standard QWERTY keyboard. The decryption time is expected to be in the bounds: $k\Sigma_{n=1}^{M-1}94^n < t < k\Sigma_{n=1}^{M}94^n$ where n is the position of the character in the string starting at 0, k is a constant of proportionality and M is the string length in characters.

The test of the prototype was conducted in a similar manner. A hash would be fed to the prototype and the time taken for the prototype to produce the corresponding password was measured and plotted. It was tested on the same range of strings as the golden measure.

Due to design of the prototype, it was very simple to produce a mathematical equation for the time taken to find the corresponding password from a hash. This equation is:

$$Runtime(s) = \frac{T}{N}n + c \tag{5}$$

Where T is the clock period, N is the number of solver modules and n is the number of hashes that need to be generated to find the correct data. The transfer overhead should remain constant and is equal to c.

E. Hardware and implementation

The FPGA used for initial implementation was an Artix-7 FPGA on a Nexys 4DDR development board.

The encryption process explained in subsection II-C was implemented on a 64 stage pipeline on the FPGA for each solver worker module. At each pipeline stage one of the 64 rounds are run. However, this causes the first 64 hashes of each worker to be incorrect as the pipeline is still being filled. As such, a `valid`/$\overline{\texttt{invalid}}$ flag will be included in the pipeline. It is necessary to use a pipeline as the propagation delay of the gates could cause errors due to the large size of the combinatorial circuit required to calculate the hash in a single clock cycle. Using a large combinatorial circuit in lieu of a pipelined approach will also allow for fewer solver modules to be implemented on a device with limited space such as the Artix-7. Using a pipeline allows for the maximum amount of hashes to be tested on the maximum number of parallel solver workers without the propagation delay of gates affecting the result.

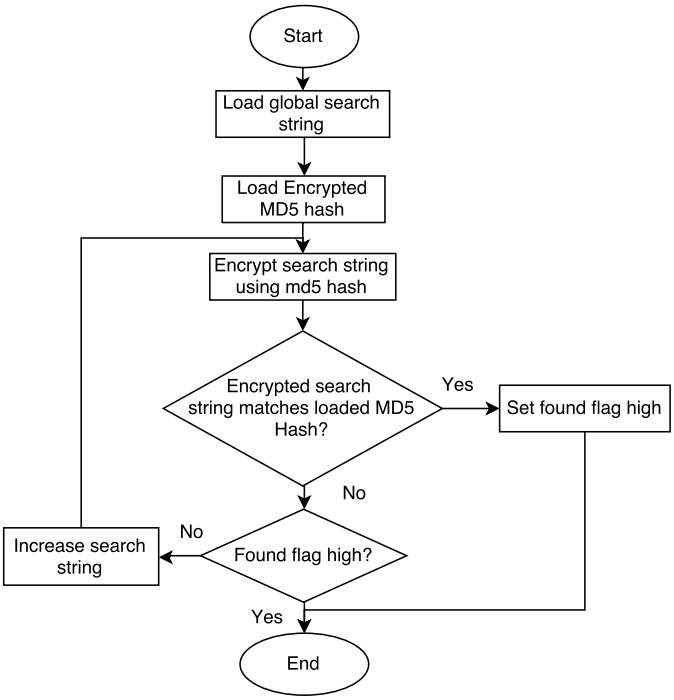

Fig. 3. Flow chart of solving algorithm

III. Detailed Design

A. Detailed System Operation and System Structure

The activity diagram showing the operation of the system is shown in Figure 4. When the MD5 hash is retrieved the brute-force computation will be run and the number of brute-force attempts will be sent to the controller at the same time. When the reversal is complete the original data will be sent to the user.

The controller module is used as the communications and data processing hub of the device. It receives, translates and transmits data between the solver manager module and the UART receive and transmit modules. If any messages have special flags or commands to be handled, this module will convert that information into appropriate commands to be send to the solver manager.

The solver manager module is used to keep track of the current brute-force string, the number of attempts and to perform distribution of the strings to be hashed between solver workers.

The actual solving/reversal process is performed on a set of fixed-function solver workers. A block diagram of a single solver worker module is shown in Figure 5.

Each worker obtains the next string to be hashed from the solver manager, computes the MD5 hash of that string and

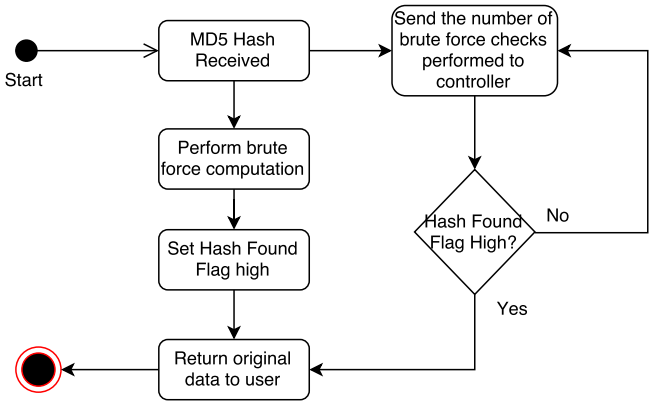

Fig. 4. Activity Diagram of the reversal system

compares the result to the input hash. The solver worker will return the string that it was hashing and the hash of that string to the solver manager. If the produced hash matches the input hash, the solver worker will assert a high on the `solved` line to indicate that it has successfully reversed the hash.

As mentioned earlier, each solver worker uses a 64-stage pipeline. This pipeline is subdivided into four parts of 16 pipeline stages, with each of the four parts performing one of the four combinatorial functions discussed in II-C

The number of workers depends on the capabilities of the platform used.

B. Communication Interfaces and Message Formatting

As mentioned above, the ASIC interfaces with the PC using RS232 serial communications [5]. The baud rate (i.e. the number of bits sent per second) has to be synchronized between the receiver and the sender. The chosen baud rate is 100 kBd. The clock speed on the ASIC is in the megahertz range. The ASIC will need to wait 10 µs per bit and 80 µs per byte.

In order to communicate with the ASIC, a python library, Pyserial, is used [6]. The commands used are shown below:

```
import serial
fpga = serial.Serial('COM5',100000,bytesize=serial.EIGHTBITS); #Establish Connection
fpga.write(b"\x01"); #Send a byte of data
bytein = fpga.read() #receive a byte of data
```

Data is sent to the ASIC at a relatively slow rate. The UART Receive module shown in Figure 2 acts as a buffer for this incoming data. This module waits for the receive line to go low, then checks the incoming signal every 10 µs and stores this value in a buffer. Once all bits have been collected a flag is set high to indicate to the ASIC controller that a byte is ready for collection. The UART Transmit module sends data from the ASIC controller to the PC in a similar manner.

A number of codes are established. These codes are 8-bit long values that PC and ASIC transmit to each other. They are used to issue commands to the ASIC and to inform the PC of the progress of the ASIC. Table I indicates all the codes that have been established.The ASIC controller module will interpret these codes and the perform the corresponding action.

TABLE I
TABLE SHOWING THE CODES ESTABLISHED TO FACILITATE COMMUNICATE BETWEEN THE ASIC AND THE PC

8-bit Code	Function
0b00000001	Incoming Hash - this instructs the ASIC that the next 16 incoming bytes are going to be an MD5 hash that needs to be stored.
0b00000010	Request Number of Combinations - this is a code that will instruct the ASIC to transmit the number of combinations currently attempted.
0b00000011	Transmitting Number of Combinations - this code sent by the ASIC indicates that the next 16 transferred bytes will represent the number of combinations attempted.
0b00000110	Confirm Solved - This indicates the hash has been found and that the next 16 bytes to be transferred will be the found hash.
0b00000111	Not Solved - This indicates that the hash has not yet been solved
0b00001000	Request solved status - This is a command sent by the controller to check where the ASIC is in terms of its transfer process.
0b00001010	Request Hash - ASIC sends the hash to the controller for error checking purposes.

C. Input and Output Hash Comparison

Due to the nature of the MD5 algorithm it is possible for multiple words/messages to map to the same hash. In the current implementation the brute-force attack starts from the lowest ASCII character combination and iterates to higher combinations. Thus, the word that is returned to the user will be the shortest possible word or the first ASCII character combination which results in that hash.

For the purpose of password retrieval it is unlikely that the word used to generate the hash would have been anything other than the first possibility. The other messages which would generate the same hash will likely be exceptionally long.

In the final system users will be able to reject the reversed hash and the system can continue to brute-force the hash until the next possibility is found.

IV. Prototype Design

This section relates to the prototype FPGA implementation of the MD5 hash reversal system. It details the simplifications that had to be made to the conceptual design in order to implement it on the Artix-7 platform.

A. Clock Speed

It is not possible to run the FPGA at as high a clock speed as an ASIC, and as such the clock speed on the FPGA was limited to 100MHz.

B. Solver Workers

Due to space restrictions on the Artix-7, it was only possible to implement a single solver module. With a clock speed of 100MHz this will still result in 100 million hashes being generated every second.

The Verilog code written allows for the number of solver modules to be changed simply by adjusting a parameter in the solver manager module. This allows for the system to be easily scaled using parallel solver workers on a higher capacity FPGA.

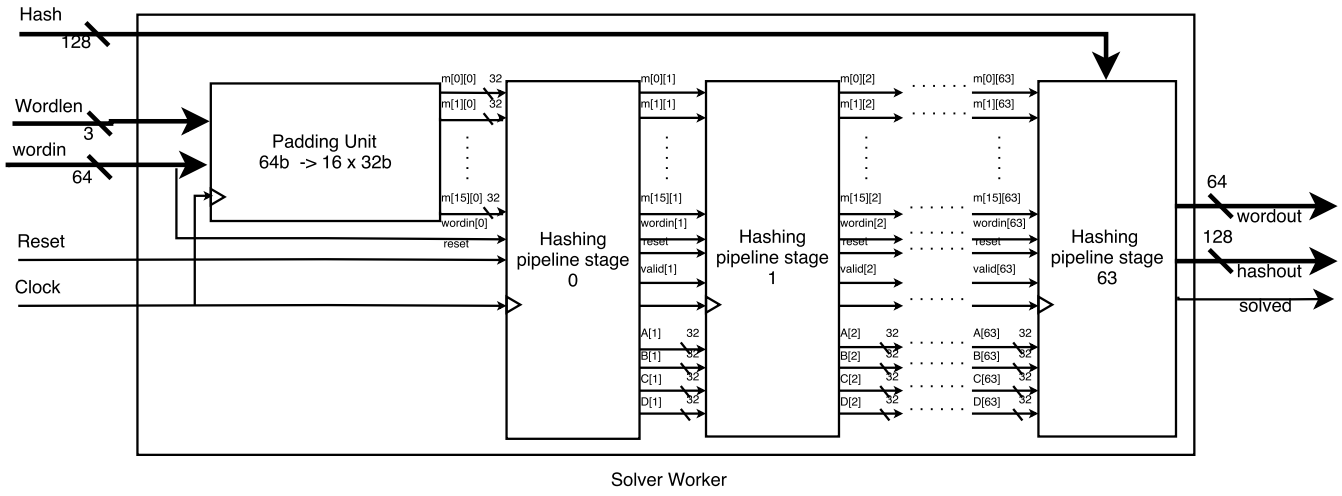

Fig. 5. Block diagram of a single solver worker module

C. Word Length

In order to simplify implementation and save space on the FPGA, the maximum word length of the brute-force attempt was limited to 8 characters. The solver manager module thus incremented the string to hash between " " (space) and "~~~~~~~~".

D. Communication Interfaces and Message Formatting

The prototype contained the full UART functionality as described in the Detailed Design Section. The FPGA has a clock speed of 100 MHz, as such the UART modules will hold the output line constant for 1000 clock cycles to meet the timing requirements of 10 µs.

V. Results

A. Measured Results

The system was tested by choosing a number of different sample words, calculating their corresponding MD5 hash and then feeding this hash into the golden measure and the prototype systems for reversal. The time taken to produce the original sample word was then timed for each system. Table II shows a comparison of some of these sample cases. The reason that only some hashes were tested was that because of the fact that the number of hashes required to be computed changes significantly with the length of string to be found, the time taken for the golden measure to reverse that hash became too long to compute.

The results from the golden measure test for reversal of the first 180 000 ASCII character combinations are plotted in Figure 6. The same results for the prototype test are shown in 7.

The prototype results for this range of character combinations do not show the same linear trend as the golden measure results due to the fact that the UART transfer overhead dominates the actual time taken to reverse the hash.

Tests on a much larger sample size of character combinations were performed on the prototype and the results

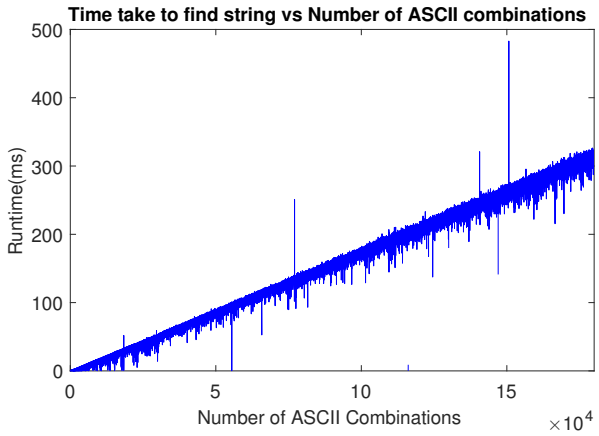

Fig. 6. Time taken to find strings with different characters using the golden measure.

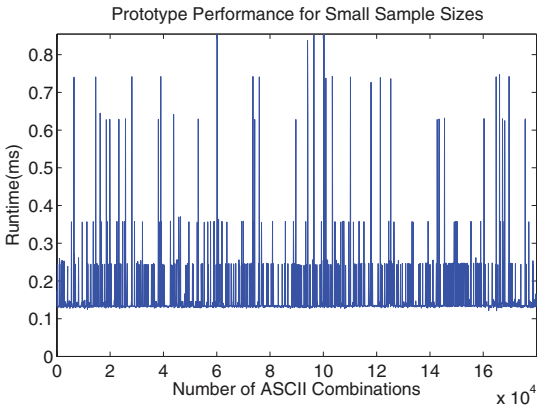

Fig. 7. Time taken to find strings with different characters using the prototype.

TABLE II
TABLE OF DIFFERENT TEST CASES AND RUN-TIMES

Phrase	Number of hashes to be computed.	Golden Measure Run-time(s)	Prototype Run-time(s)
so	7 975	0.0001	0.163
axe	591 611	0.946	0.156
wax	783 860	1.332	0.164
test	70 386 626	113.465	1.267
vader	6 847 950 842	11 734,2340	60.934
Newton	350 872 752 449	Too long to compute	3340.630

are shown in Figure 8. The transfer overhead is much smaller portion of the run-time for longer character combinations and thus a similar linear trend to the golden measure emerges.

B. Mathematical Analysis and Speedup

The equation for the time to reverse a hash for an ASCII character combination for the golden measure system was found from the graph in Figure 6. It is shown below, with **n** representing the number of ASCII character combinations before the string to be tested:

$$\mathbf{Runtime(s)} = \mathbf{1.7 \times 10^{-6} n} \tag{6}$$

The equation for the time taken to reverse a hash on the FPGA was shown in the overview section. The clock period is 10 ns and the number of solver modules in the prototype was 1. **n** again represents the position of the string in all character combinations. The expected run-time for a single hash is:

$$\mathbf{Runtime(s)} = \mathbf{1 \times 10^{-8} \times n + 0.2} \tag{7}$$

The transfer overhead does vary (data sometimes needs to be resent to ensure reliability) so an average value for this overhead was chosen for this equation.

From the previous results and equations the speedup of the prototype system over the golden measure was calculated. Figure 9 shows this calculated plot.

The point where the speedup is greater than one occurs where the number of ASCII combinations is equal to 118344.

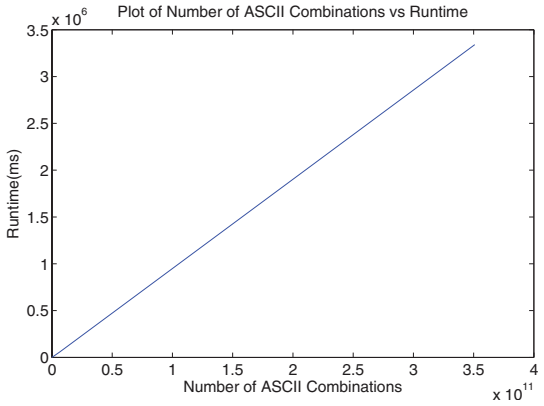

Fig. 8. Time taken to find strings with different characters using the prototype for longer length strings.

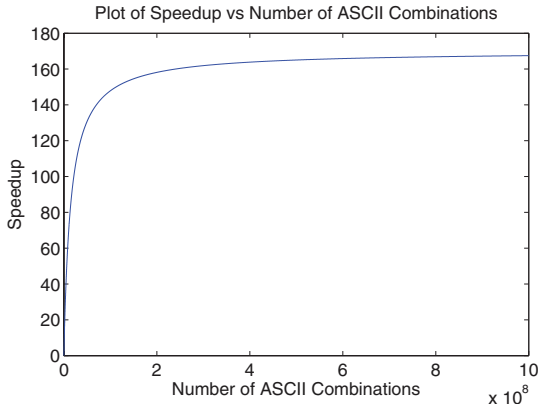

Fig. 9. Speedup Plot of the FPGA implementation over the Golden Measure.

This is equivalent to finding the corresponding hash for the three character string ",)d".

As the sample size increases the speedup tends to a constant value of approximately 170.

VI. CONCLUSION

The results of the experiment show that the prototype system was a success, reversing hashes up to 170 times faster than the golden measure. Because of the fact that the golden measure was a parallel implementation on a CPU, the speed up of the prototype over a purely sequential system will be even greater.

The prototype is able to be easily parallelized through changing a parameter in the Verilog code of the system. This will provide massive improvements to the speed up of the system. Each additional solver module added in parallel would increase the speed up of the system by an additional 170 for larger problem sizes.

The next stage of development would be to expand the system to remove the limitations imposed by the Artix-7 FPGA platform. This includes increasing the clock speed, adding parallel solver worker modules and increasing the number of characters that the system can solve for. Additional features for the user can also be added, such as the ability to set the starting point of the brute-force attack and the ability to reject the result of the reversal and force the brute-force attack to find the next possible string.

REFERENCES

[1] Kaspersky Lab, "What is Flame Malware?" http://www.kaspersky.com/flame.

[2] Fast Sum, "What is the MD5 checksum?" http://www.fastsum.com/support/md5-checksum-utility-faq/md5-checksum.php.

[3] Open Web Application Security Project, "Password Length & Complexity," ,https://www.owasp.org/index.php/Password_length_%26_complexity.

[4] B. Grdelbach, "MD5 library," http://hashlib2plus.sourceforge.net/.

[5] Silicon Labs, "Serial Communications," ,http://www.silabs.com/.

[6] Pyserial, "Python Serial Port Extension," ,https://pypi.python.org/pypi/pyserial.