EEE4120F Project

Mikhail Russel [†] , Rory Schram[‡] , Natasha Soldin[§] and Reid Stuart [¶]
EEE4120F Class of 2023
University of Cape Town
South Africa
[†]RSSMIK001 [‡]SCHROR002 [§]SLDNAT001 [¶] STRREI003

***Abstract*—The following design report is for the implementation of a Versatile Accelerated Digital Encryption Recovery (VADER) unit on a Field Programmable Gate Array (FPGA). The encryption recovery applies to hashed passwords that require computationally complex and expensive recovery methods. The chosen hashing function is a MD5 method that performs one way encryption of any given input. This implementation investigates the potential speed-up or increased efficiency that an FPGA, and its associated parallel programming, can achieve in contrast to a sequential password recovery program. Theoretically, the ideal system would comprise of both a dectionary and brute force attack method but given the time and resource costraints of the project, only the brute force attack was implemented. The FPGA's parallel implementation achieved a speed-up of 10.7763 over the serial python implementation.**

I. Introduction

A. Project Description

The Versatile Accelerated Digital Encryption Recovery (VADER) project aims to recover passwords using a acquired hashed password and hashing function.

Physically it will be an add on hardware element, more specifically a Field Programmable Gate Array (FPGA), that utilises parallel programming techniques to more efficiently complete computationally expensive tasks - in this case hashed password recovery.

Hashing passwords is a commonly used practise and thus this type of system, one that addresses and minimises the complexity of hashed password recovery is greatly desired and motivated.

B. Proposed Solution

The project solution, as dictated by the project scope, will be a single system that implements two attacks in sequence. The first attack will be dictionary based and if that proves to be unsuccessful then the second or fail safe attack which is brute force will be executed. These two attacks and their ideal combination is discussed further in section III below.

C. Report Structure

The following report structure detailing the project includes the following:

1) **Background**: provides the necessary explanations of the theoretical concepts used within this project and referenced throughout this report. It explains the practise of password hashing, compares and contrasts hashing vs. encryption, describes the specific operation of the project's selected hashing function MD5 and explains the different hashed password recovery attacks, namely dictionary and brute force.
2) **Methodology**: lays out the ideal approach to the project, breaking it down into three sections being the two hashed password recovery attack methods individually and then the ideal combined approach that incorporates both of them. It also describes possible tests that can be run on the three programs and specifies the hardware and software that the project will require.
3) **Design**: proposes the possible design for all programs in both a Python golden standard implementation as well as a Verilog FPGA simulated one.
4) **Proposed Development Strategy**: provides a brief overview or thought experiment about what future development could look like if this VADER system was to be deployed as a commercial product.
5) **Planned Experimentation**: explains how the testing process will be performed on the golden standard and FPGA implementations of the VADER system.
6) **Results**: shows the results produced from the tests performed on the system during the experimentation phase of the project.
7) **Conclusion**: provides a summary of the overall project and recommendations on future work.

II. Background

A. Hashing vs. Encryption

Hashing differs from encryption in that encryption is a two-way function that takes plain text, encrypts it to produce cipher text and allows for decryption which outputs the original plain text.

Hashing however, is described as a one way (or irreversible) function that takes in plain text and produces a hashed function that cannot undergo decryption. The original plain text can be recovered but it is incredibly computationally expensive and complex [1]. These concepts are graphically explained and contrasted in figures 1 and 2 below.

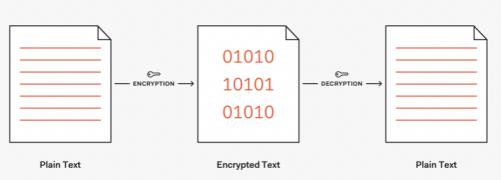

Fig. 1: Encryption and Decryption Diagram

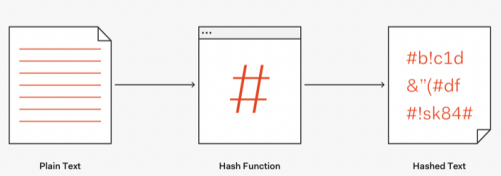

Fig. 2: Hashing Algorithm Diagram

In order to recover a hashed password, a plain text password is needed. The hashing function is applied to it and the result is compared to the original hashed function. This process is repeated until the two hashed functions match, signaling that the chosen plain text password is in fact the original password. The complexity of 'de-hashing' is thus derived from the vast number of plain text password possibilities that need to undergo this process and the repetitive nature of the program.

B. MD5 Hashing Protocol

The project's chosen hashing function is the MD5 hash protocol, also known as the message-digest algorithm. This function, as described above, is a one-way cyrptographic method that accepts an input of any length and produces a fixed length (128-bit) hashed password [2].

The MD5 hash function executes in four steps [3] [4]:

1) **Bit Padding**: the original password is padded such that it is 64 bits shy of a multiple of 512 bits. The first padding bit is 1 and the rest are 0's.
2) **Length Padding**: this bit padded message is then appended with 64 bits at the end. This is used to record the password's initial length and results in a message that is a multiple of 512 bits that is ready to be encrypted.
3) **MD Buffer**: four 32-bit buffers are initialised, the manner of which is shown by table I below.

Word A	01	23	45	67
Word B	89	Ab	Cd	Ef
Word C	Fe	Dc	Ba	98
Word D	76	54	32	10

TABLE I: MD Buffer Initialisation [3]

4) **Processing**: each message is broken down into 16 32-bit sub-blocks that are processed using logical operation functions as shown by table II below. These functions are used to execute the MD5 hashing in which the MD Buffer is 'mixed' with the padded input message.

F(X, Y, Z)	XY v not (X)Z
G(X, Y, Z)	XZ v Y not (Z)
H(X, Y, Z)	X xor Y xor Z
I(X, Y, Z)	Y xor (X v not (Z))

TABLE II: Logic Functions [3]

The MD5 hash protocol is no longer considered to be cryptographically secure according to the Internet Engineering Task Force (IETF) as it does not meet the two criteria required, these being that an external attacker would not be able to (1) produce a message that matches a hashed password or (2) produce two different messages that match the same hashed password [2]. Thus, MD5 is susceptible to varying attacks.

C. Dictionary Attack vs. Brute Force Attack

As described in section II-A above, the manner in which a hashed password is recovered is by taking in a randomly chosen word, applying it with the specified hash function and comparing it to the hashed password whereby a match would signify the identification of the original plaintext password - which is repeated until a match is found. The source of the plain text password stems from the type of attack that is chosen to be implemented - within this project's scope this can either come from a dictionary of a brute force generation.

1) Dictionary Attack: This type of attack would make use of a dictionary, of which every dictionary entry can be tested to see if it matches the correct hashed password [5]. The benefit of initially using a dictionary attack is that the password is most likely to be a combination of comprehensible words as they are easier to remember (i.e. those that will appear in a dictionary).

2) Brute Force Attack: This type of attack would make use of a generated 'list' of every possible password combination given the limitation of what characters or characteristics the password comprised of. This refers to password length, letters being of upper or lower case (or both) and the use of digits or special keyboard characters. The benefit of this approach is the assurance that the password will be retrieved due to the extensive and thorough generated database, however the process of working through that database can be cumbersome and slow which can result in the compute time of the program being far longer than that of the Dictionary Attack due to the length of the generated 'list'. To illustrate this, say for example that the password is limited to four characters in length, the use of both lowercase, uppercase, and digits but no special keyboard characters. There would then be 62 $(26 + 26 + 10)$ possible characters that could make up the password. Therefore, there would then be $62^4 = 14776336$ (just shy of 15 million) combinations.

Therefore, ideally the system should first employ the dictionary attack and then if failure of password retrieval occurs, the brute force attack will be employed. This is in attempt to minimise the complexity of the program (i.e. not to have to resort to using the brute force attack) while still ensuring that the target password in retrieved.

III. Methodology

The methodology is a continuation and explanation of the Proposed Solution introduced above and further contextualised in section II above. It will list the neccesary hardware and software required for implementation as well as describe the method of implementation of the two individual attacks (Dictionary Attack and Brute Force Attack) and then go on to describe the potential testing that can be applied to obtain meaningful results.

A. Hardware and Software

For successful implementation and testing of the proposed VADER system, regardless of the chosen implementation approach (i.e. dictionary, brute force or both), the following hardware and software elements will be required:

1) **Personal Computer (PC)**: a computer is required to code and host the integrated development environments (IDE) required for both the sequential and parallel implementations. In this case Python and Verilog respectively.
2) **Python**: this programming language is used to code the golden measures or sequential programs for both the dictionary and brute force attacks.
3) **Xilinx Verilog**: programming language and compilation environment that will simulate the FPGA and run the parallel Verilog code for both the dictionary and brute force attacks.
4) **Simulated Nexys4 DDR Board**: this is the chosen FPGA that has the following specifications:

Specifications	
Category	**Value**
FPGA Part Number	XC7A100T-1CSG324C
Logic Slices	15,850 (4 6-input LUTs & 8 flip-flops each)
Block RAM	4,860 Kbits
Clock Tiles	6 (each with PLL)
DSP Slices	240
Internal clock	450 MHz+
DDR2	128 MiB
Cellular RAM	16MB
Ethernet	10/100 PHY
SD	microSD card connector
Pmod Connectors	4 Pmod ports
VGA	12-bit VGA port
Audio	PWM audio output
Microphone	PDM mic
On-Board Sensor	Temperature Sensor
Display	2 4-digit seven segment displays
Switches	16
Buttons	4
LEDs	16
Power	USB 5 V (2.5 mm coaxial) supply
Logic Level	3.3 V
Size	4.3 x 4.2 inches

TABLE III: Nexys4 DDR Board Specifications [6]

5) **Github**: git was used to for project collaboration and version control. The project's github repository can be accessed here.

B. Project Method and Testing

This project effectively involves three programs: one that implements only the dictionary attack, one that implements only the brute attack and one that implements the ideal system (which is a sequential combination of the two starting with the dictionary attack and followed by the brute force attack to ensure password recovery).

All three programs will be implemented and tested in a serial/software implementation and in a parallel/hardware one.

- The programs will be coded in Python which will execute sequentially and act as the project's golden standard.
  - A golden standard is a correctly functioning version of a program. It may be un-optimised and slow in execution but it has guaranteed correct operation. It is often used as a model to compare other implementations against.
- The programs will be coded in Verilog on an FPGA which will execute in a parallel manner.

These two implementations will be compared to determine if a significant speed-up or increased program efficiency can be obtained by using an FPGA or digital accelerator in this project's application. It is hypothesised that the FPGA will allow for a faster hashed password recovery than a sequential software approach. Further testing will also be conducted to test project implementation.

The tests that will then be applied to these programs include:

1) Comparing the serial and parallel versions for all three programs to determine their speed-up and by extension their potential system improvements.
2) Modifying the degree of parallelization (i.e. increase the level of parallel programming) to determine the optimal level of parallelization or number of parallel threads/ modules for the project's application

IV. Design

This section will explain the design of the aforementioned three approaches: dictionary attack method, brute force attack method and the two methods combined. An overview of the general operation of these three methods will first be given with the aid of flow charts as seen by figures 3, 4 and 5 below. This will provide a basis of understanding from which a more detailed design can be created. With this basis, the Python and Verilog designs will be described in detail for each of the three methods.

A. Overview of General Attack Methods

1) Dictionary Attack Method: The manner in which the dictionary attack is conducted is graphically explained by the flow chart shown in figure 3 below.

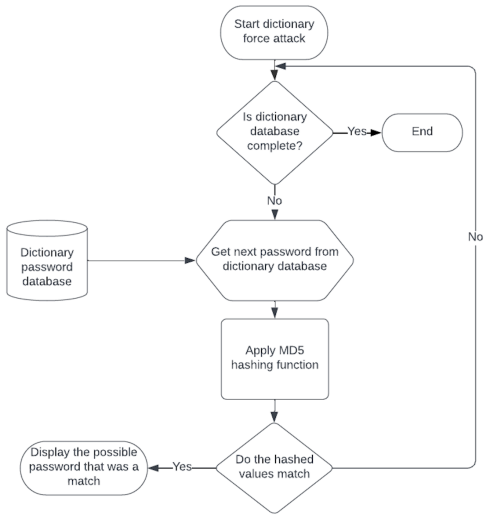

Fig. 3: Dictionary Attack Flow Chart Representation

This shows that passwords are retrieved from a dictionary database before having the hashing function applied to them. These hashed passwords are then compared to the target hashed password until a match occurs.

2) Brute Force Attack Method: The manner in which the brute force attack is conducted is graphically explained by the flow chart shown in figure 4 below.

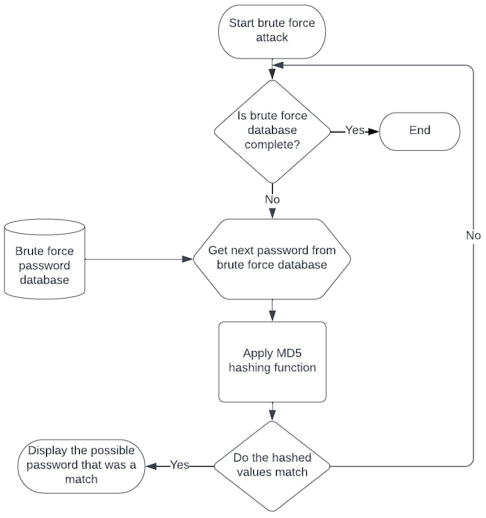

Fig. 4: Brute Force Attack Flow Chart Representation

This shows that passwords are retrieved from a pre-generated brute force database, one that has every possible combination of password, before having the hashing function applied to them. These hashed passwords are then compared to the target hashed function until a match occurs.

This process appears very similar to that of the Dictionary Attack Method, the difference however is that the brute force's database is far larger and 'full-proof' in the sense that it will undoubtedly contain the target password but in and amongst a far bigger dataset - which will increase the program's execution time.

3) Combined Attack Method: The manner in which the Dictionary Attack Method program and the Brute Force Attack Method program will be combined to produce the ideal overall system program is graphically explained by the flow chart shown in figure 5 below.

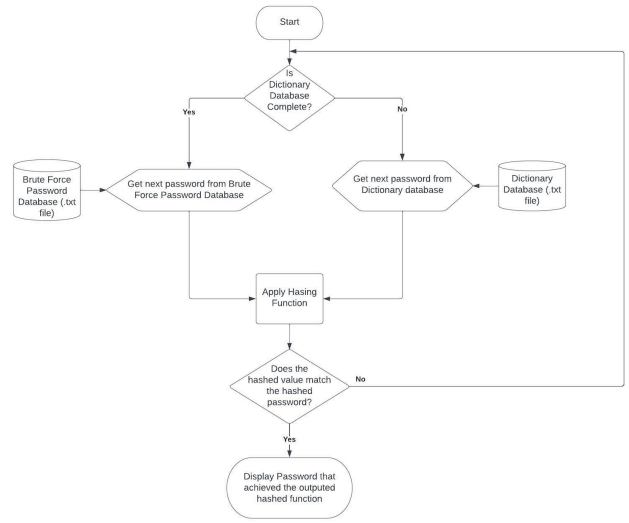

Fig. 5: Dictionary and Brute Force Attacks Combination Flow Chart Representation

This shows that the hashed password is first compared to each of the hashed dictionary values obtained from the dictionary database, this continues until the ends of this database is reached. This signifies that the dictionary attack method has failed and then the password is compared to each of the hashed brute force generated values from the brute force database. This will ensure password recovery.

B. Attack Method Designs

The three attack method designs will overlap substantially due to the fact that the combined attack has both the dictionary attack method and brute force attack method incorporated within. Therefore, the combined attack method will first be designed and then an explanation will follow on how to implement the separate attack methods individually (i.e. the effective design of the dictionary attack method and brute force attack method individually).

1) Python Golden Standard Implementation: The approach for the design of the Python Golden Standard system can be seen in the UML diagram in figure 6 below.

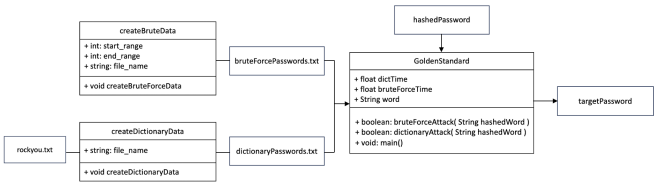

Fig. 6: Proposed Design of Golden Standard Python Implementation

The python `Golden Standard` programs will take in as input the hashed password (`hashedPassword`) that requires recovery. Depending on the attack method being applied, the program will execute in the following different ways:

- **Dictionary Attack Only**: the password is sent in as a parameter to the `dictionaryAttack` method which compares it to each hashed dictionary entry within the `password.txt` text file. This text file was generated using the `createDictionaryData` program which takes in a common dictionary text file `rockyou.txt` file as input and applies the hash to each dictionary entry. When a match occurs the program breaks and outputs the `targetPassword` and saves the execution time to the `dictTime` variable. **Note**: there is a chance that a stand alone dictionary attack method will not recover the password in which case the program will output a message to the user signalling the recovery failure.
- **Brute Force Attack Only**: the password is sent in as a parameter to the `bruteForceAttack` method which compares it to each hashed value entry within the `password.txt` text file. This text file holds the hashed value of all possible password combinations and is generated by the `createBruteForceData` program. When a match occurs the program breaks and outputs the `targetPassword` and saves the execution time to the `bruteForceTime` variable. **Note**: unlike the Dictionary Attack, this attack method can be implemented as a stand alone method which will find the target password.
- **Combined Attack Method**: the password is first sent in as a parameter to the `dictionaryAttack` method and executes as described above, if this attack is successful then the the program breaks and outputs the `targetPassword` and saves the execution time to the `dictTime`. However, if the dictionary attack is unsuccessful (i.e. the end of its `password.txt` text file without finding a match), then the hashed password is passed into the `bruteForceAttack` method and executes as described above until a match occurs the program breaks and outputs the `targetPassword` and saves the execution time of the dictionary portion to the `dictTime` variable and the execution time of the brute force portion to the `bruteForceTime` variable.

2) Xilinx Verilog FPGA Implementation: The approach for the design of the Verilog FPGA system can be seen in figure 7 below.

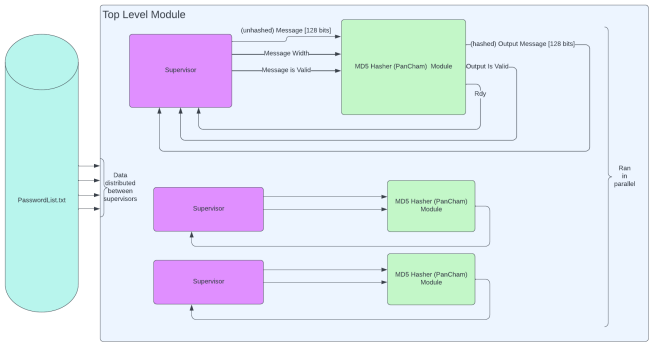

Fig. 7: Proposed Design of Verilog Modules

Figure 7 shows that the design approach would make use of a *Top Level* module. The top-level initializes *n* amount of supervisor modules connected to *n* amount of MD5 Hasher modules, where *n* is a variable dependent on the total number of passwords. Each supervisor has access to a set amount of passwords, as well as the target hashed password. An example of how the Top level module calls a supervisor is shown below:

Each MD5 module is based on the Fig *Pancham* MD5 Hasher module [7]. These modules take in a 128-bit value message, the message bit width, and whether the value is valid or not. It outputs a 128-bit hashed message, whether the message outputted is valid, and if the module is ready for its next input.

Each supervisor manages their own MD5 hasher. It ensures that there is a new message to be hashed when the MD5 module is ready. It also completes the check to see whether the outputted hashed function from the hasher module matches the target hashed function.

Both the Brute Force attack and Dictionary attack implementation are described in more detail below, and are shown in 8. In both implementations, the time taken to hash each password is dependent on the time it takes for the MD5 module to complete its hashing algorithm. This is directly proportional to the clock speed of the board, which has its limitations. Thus the only way to speed up each implementation would to be increase the number of MD5 modules.

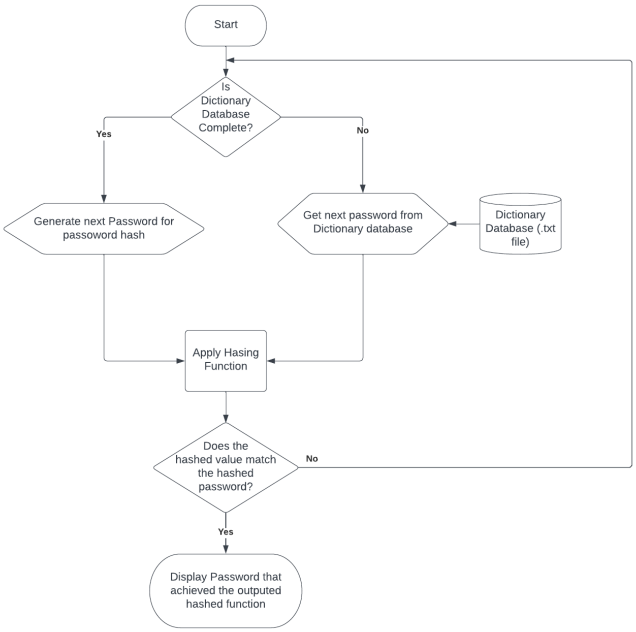

Fig. 8: Proposed Design of Verilog Modules

The individual attack methods can be extracted from the above explanation as follows:

- **Dictionary Attack Only**: the top level module would simulate the FPGA reading passwords from a SD card. The top level module would then distribute the passwords between the supervisors to execute in parallel. Thus each supervisor would be starting at a different point in the text document. The supervisor also monitors the output, checking whether or not it produces the targeted hashed password.
- **Brute Force Attack Only**: each supervisor would be given an initial set of characters for which to start its brute force attack. It would begin incriminating each character whenever the MD5 module is ready for a new input. This would be run in parallel by initializing each supervisor to start at a different point of character. The supervisor also monitors the output, checking whether or not it produces the targeted hashed password.

V. Proposed Development Strategy

As it stands, the system's golden standard implementation is in it's final form, meaning that both the dictionary attack and brute force attack are fully functional and execute in the correct manner. It is therefore possible to extract consistent golden standard timings to provide insight into the serial performance of the programs.

The FPGA implementation of the brute force application is currently in a working prototype phase in a simulated environment. As it stands, the FPGA implementation will generate the brute force password list "on the fly", and will hash and compare a hard coded hashed password of our choice, if it is contained within the brute force password limitations that have been set out in section II-C2.

There is currently no FPGA implementation of the proposed dictionary attack method, simulated or physical, due to project limitations detailed in section VI below. However, for future implementation of this method, ideally an SD card that stores the dictionary database will be used. This database would be loaded into the builtin micro-SD card slot on the Nexys board such that the program can access it, iterate though it, hashing the passwords and comparing them to the original hashed password to recover the target password. The process of iterating through this list sequentially could be optimised by implementing parallel programming techniques - in this case each supervisor initiated will be responsible for a section of the dictionary database.

In the future, the proposed development strategy would be to physically implement both the dictionary and brute force attacks on an FPGA board. Ideally, a user of the system could input one password value into the FPGA system using hardware based wires or even a USB connection. Once the value of the target hash is loaded into the system, the system would begin an FPGA implementation of the dictionary attack, and if this fails, then the system would begin the FPGA implementation of the brute force attack. This final proposed solution would implement the ideal system and could be manufactured as a commercial product capable of hashed password recovery.

For initial testing purposes implemented by this project, implementation of separate dictionary and brute force golden standard methods and implementation of only the brute force FPGA simulated method is adequate. It shows progress towards the ideal system described above.

VI. Planned Experimentation

It was decided that given the scope and limitations (i.e. timeline, resource availability, student's experience) of the project, the project would only practically implement the Brute Force Attack Method. This method was chosen because as highlighted in section III above, it can act as a 'stand alone' hashed password recovery method as in it will undoubtedly retrieve the target password whereas with the Dictionary Attack Method, if the password is not within the dictionary database which could be due to password complexity (i.e. inclusion of digits and characters) - which is common practice in ensuring password security, then there is a chance that the target password will not recovered. Regardless of the cumbersome nature of the Brute Force Attack Method and its related, extensive database, it should still be able to illustrate the aim of this project and achieve a speed-up when implemented on a FPGA.

Therefore, the only Design that will be considered and implemented for testing are the Python Golden Standard Implementation and Xilinx Verilog FPGA Implementation Brute Force Attack Only for comparison as well as the Python

Golden Standard Implementation Dictionary attack only to show proper operation.

A. Time Measurement

In order to obtain meaningful conclusions of the programs' performance, their execution time will be required to be obtained. This is done differently for the python and Verilog implementations as described below.

1) Golden Standard Serial Time Measurement: The Golden Standard method run in python uses the inbuilt python function time() to calculate the time taken to run the dictionary attack and the brute force attack respectively. The program is then looped to increment the number of passwords it needs to crack and the new timing values get recorded. After each implementation of the program the timing values are written to csv files for ease of access.

2) FPGA Parallel Time Measurement: The Pancham module pulls a *ready* wire high when it completes a hash. A counter would be added to determine how many hashes are completed within a certain amount of time which can be translated to hashes per second. This would be confirmed by measuring the period for which the *ready* goes high. As more supervisors and MD5 modules are added, the rate at which the passwords are hashed will increase (e.g. five supervisors would result in hashing speeds around 5 times faster than an individual module). Thus, It is expected that there will be an increase in speedup as more parallel implementations (Supervisors) are added. This will continue until the overheads of each Supervisor begin to become noticeably larger compared to the processing task it is allocated. Thus creating a tapering off of the speedup graph. These measurements would be independent of whether the supervisor is running the brute force attack or a dictionary attack. Additionally the overhead costs of loading in the dictionary and hashed passwords from their respective text files are not included in our timings and are treating as prepossessing.

B. Calculating Speedup

The golden standard brute force attack program will run and be timed cracking a single password (9999) which represents the 'hardest' password the brute force attack program will be required to recover. The Vivado simulation likewise will crack the same password (9999) for various numbers of supervisors. Due to the simulation speed in Vivado being so slow, a reliable value of hashes per second should be obtained at low supervisor numbers which are achievable - this will then be mathematically extrapolated to obtain timing values at higher numbers of supervisors.

The speedup achieved by the VADER system will be calculated by taking the golden standard execution time divided by the FPGA execution times. The number of supervisors on the FPGA will serve as a representation of parallel implementations and be treated as the independent variable plotted against speedup to show the relationship speedup has against the amount of parallel processing.

If the system was implemented purely in software we would expect to see the amount of speedup achieved tapering off as the ratio between overheads to create each parallel implementation (Supervisor) and the time they actually save begins to decrease. The system we have created using the FPGA however should not suffer from this limitation as once the board is configured each parallel branch exists in the hardware allowing it to be used with zero overhead. Plotting the speedup values calculated against the amount of parallel implementations we would thus expect to see a linear increase of speedup. This increase would continue until the system is capped by the hardware limitations of the FPGA. This limitation can be calculated using the physical number of LUTs available of the FPGA itself and the numbers of LUTs that each module of our system requires.

VII. Results

A. Python Golden Standard Implementation

The brute force program was tested on a worst case scenario password that is defined within the scope of our investigation. This worst case scenario password is "9999", as it is the last possible password iteration that can have its hash checked with the target hash. The execution time of the brute force algorithm to determine the password that matches the hash of "9999" was determined to be 9.4292 seconds. This value will be used to calculate speedup in conjunction with the theoretically calculated brute force timing values collected by increasing the number of supervisors that are running in parallel.

B. Xilinx Verilog FPGA Implementation

1) Implementing Supervisors in Parallel: The example below in Figure 9 shows the *Top Level* module generating multiple supervisor modules

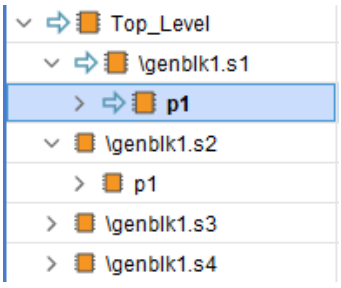

Fig. 9: VADER Structure

This module is then given control of its own *Pancham* module. The *Supervisor* starts generating passwords, as shown in 10.

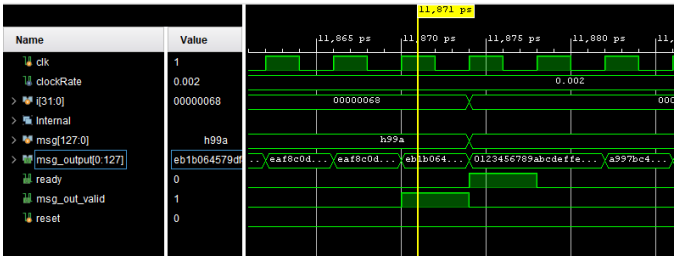

Fig. 10: Example of Brute force attack incriminating Passwords

Figure 10 above shows the operation of the brute force attack: the *msg_out_valid* signal goes high when the target hash is a valid hash, and then *rdy* signal goes high indicating when the module is ready for the next password. The *Supervisor* checks to see if the output hash matches the target hash when *msg_out_valid* goes high.

In the example below, the target password is **a9BE**. This corresponds to a target hash of **13aad5860139a22c6dd6d1304a2a0ec9**. Once the target hash has been found, the reset is pulled high and the console outputs: ”Password found to be a9BE” as seen by figure 11 below.

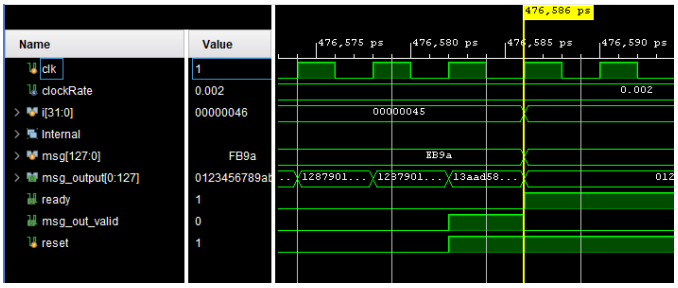

Fig. 11: Example of a Supervisor Finding the Targeted Hashed Password

Note: the message is required to be reversed to make use of the *Pancham* module, this was just discovered to be the manner in which the module operates.

2) Calculating Speed of Pancham Module: First, the clock rate that the system would operate was chosen. Because the brute force attack was the only method implemented and it generates passwords during program runtime (i.e. the FPGA does not read from a SD card), the clock rate can be increased until it matches the board's maximum clock rate. Thus, 400MHz was selected. The 400MHz is achievable by using the *Clocking Wizard IP*, which takes the 100MHz internal clock and converts it to a 400MHz clock for this project's purpose.

The number of clock cycles taken for an individual Supervisor Module to complete checking one password was recorded to be 67 cycles. Using the clock rate and number of cycles to complete a hash, the seconds per hash and hashes per second were determined.

The seconds per hash was found to be 167.5 $ns/hash$, while the hash per nano-second was 0.06 $hashes/ns$. This was confirmed by taking a measurement of the time taken for the MD5 module to set *msg_out_valid* high, indicating that the module has completed hashing. The Vivado simulation could not simulate at such a speed - due to the software limitations, resulting in the clock speed needing to be scaled up.

In this experiment, the chosen password was the worst case value, hence the password being ”9999”. From the hashes per second calculated and confirmed in simulation, Figure 12 below shows the time taken to reach the chosen password as the number of supervisors increases.

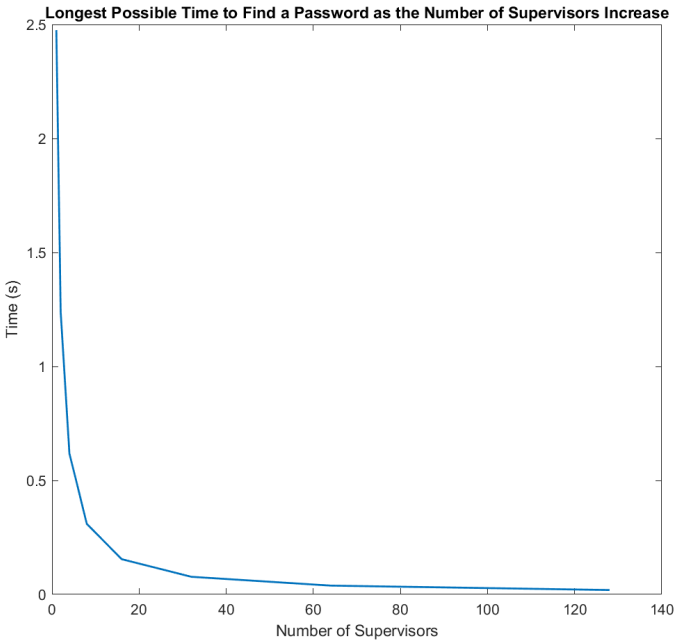

Fig. 12: Time Taken to Recover the Target Password with a Varying Number of Supervisor Modules

This indicates that the time taken for one supervisor to find the password is 2.475 seconds. As more supervisors are added, the time taken to reach the worst case is reduced. At 128 supervisors, the time taken was 0.0193 seconds. This is 128 times faster than using one supervisor. From this data, it can be concluded that the speed-up, when comparing supervisors is directly proportional to the number of supervisors. This is shown in Figure 13.

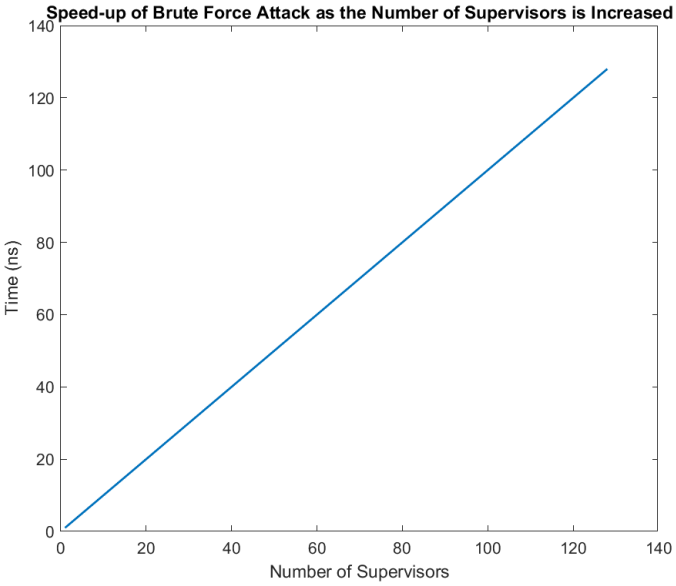

Fig. 13: Speed-Up of the Program with a Varying Number of Supervisor Modules

This is however unrealistic in a physical implementation, as each Supervisor would use a certain amount of look-up-tables (LUTs) and there is a finite amount of LUTs available. Therefore increasing the number of supervisors to a certain degree is an impractical method of program improvement. This is further discussed in the comparison section below.

C. Comparing Serial and Parallel

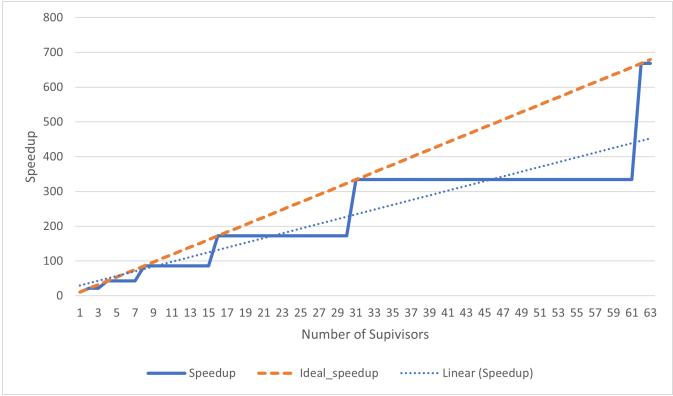

Fig. 14: Serial and Parallel Speed-Up Comparison

Figure 14 above shows the ideal linear increase (orange line) in speedup with respect to number of supervisors, this line was calculated and does not take into account the fact that a single iteration of the brute force data cannot be divided any further into sub-iterations.

The actual speedup achieved by the system in this case is shown by the blue line. The steps in speedup are located at the values of supervisors where the total number of brute force iterations needed to be completed is perfectly divisible by the number of supervisors. This means that the system is wasting no time waiting for any parallel branch which may have run longer than the others otherwise. When the trend line (dotted blue line) is plotted we can see that the increase is still linear in nature just with a slightly smaller gradient that the ideal situation.

Note: the specific values calculated are relative to the password we wish to crack (9999) if this was a different password such as ”Yoda” the step function would appear different.

VIII. Conclusion

The speedup results obtained show a linear increase in speedup as the amount of supervisors running in parallel increased. For a single supervisor, the FPGA implementation still achieves a speed-up of 10,7763 over the python implementation. In theory, a maximum speedup value would be achieved when the maximum amount of LUTs are used up by the supervisors. A speedup value greater than this maximum could not be achieved because the FPGA does not have enough LUTs available to implement in the program being executed.

A. Recommendations for Future Work

As it stands now, the working implementation is only via a simulated FPGA and it only implements the Brute Force Attack Method. For future work on this project, focus will be placed on implementing all attack methods, ultimately the ideal combined attack method on a physical FPGA. This will allow for more accurate results to be obtained as the physical limitations of the board reveal the practical application that can be achieved by the system

References

[1] Puneet, “What is difference between encryption and hashing? is hashing more secure than encryption?” Aug 2022. [Online]. Available: https://www.encryptionconsulting.com/education-center/encryption-vs-hashing/

[2] M. E. Shacklett and P. Loshin, “What is md5 (md5 message-digest algorithm)?” Aug 2021. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/MD5#:~:text=The%20MD5%20(message%2Ddigest%20algorithm,for%20authenticating%20the%20original%20message.

[3] S. M, “Md5 hash algorithm in cryptography: Here’s everything you should know,” Feb 2023. [Online]. Available: https://www.simplilearn.com/tutorials/cyber-security-tutorial/md5-algorithm

[4] “Md5 algorithm: Know working and uses of md5 algorithm,” Mar 2023. [Online]. Available: https://www.educba.com/md5-alogrithm/

[5] T. Porter and M. Gough, “Chapter 7 - active security monitoring,” in *How to Cheat at VoIP Security*, ser. How to Cheat, T. Porter and M. Gough, Eds. Burlington: Syngress, 2007, pp. 185–206. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597491693500086

[6] Martha, “Nexys 4 ddr.” [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4-ddr/start

[7] S. Mitra, “Pancham, an md5 compliant ip core.” [Online]. Available: https://pancham.sourceforge.net/