Cracking the Off the Grid Password Solution

Matthew J. Miller
University of Nebraska at Kearney, millermj@unk.edu

Joshua Stroschein
Dakota State University, joshua.stroschein@dsu.edu

Stephanie Slayden
Kansas State University, sslayde@ksu.edu

CRACKING THE OFF THE GRID PASSWORD SOLUTION

Dr. Matthew Miller
millermj@unk.edu
Dr. Joshua Stroschein
Joshua.Stroschein@dsu.edu
Stephanie Slayden
sslayde@ksu.edu
University of Nebraska at Kearney, Dakota State University, Kansas State University

ABSTRACT

Authentication and authorization to online sites is a difficult problem to solve without the use of cryptography. The standard method of using passwords is clearly an insecure method of authentication. A method of authenticating users utilizing a Latin square was developed by a security enthusiast and touted as secure. This paper demonstrates a novel method of cracking the Latin squares that are used to generate the secure passwords in the Off The Grid (OTG) password management scheme. Our method leverages the cores on Graphics Processing Unit (GPU) using the Compute Unified Device Architecture (CUDA) programming extension to efficiently solve the Latin squares used in the OTG password management solution. We developed a model that represents the possible states and the constraints of the OTG system. We show that the OTG system leaks information about its Latin square and we provide supporting evidence through examples and computation.

Keywords: Parallel Computing, Passwords, Latin Squares, GPU, CUDA, Authentication, Access Control, Cloud Security

1. INTRODUCTION

The use of passwords has become a typical method of authentication for users of digital systems for the past 30 years. This use was spawned out of the original Unix systems, where users shared the resources of a multi-user system. During the rise of the Internet, these password-based systems were thrust upon users of websites. Users were either given an easy to remember password or they were allowed to choose their own password for authentication. Websites are then responsible for storing the secret password that user will use to authenticate to the server. In the early days, websites would store the password in plain text and simply compare the password on the server with the password that the user entered when authenticating.

Attackers have had the ability to gain unauthorized access to Internet connected computer systems as long as computers have been connected to the network. When an attacker gains access to a website that has passwords stored in plain text, then that attacker can then easily authenticate as any user of the system. The loss of a single password from a single website should not give

the attacker access to a different website, except when the user uses the same password for both sites. With the growth in the number of websites that are used for normal day-to-day activities, users do not maintain a unique password for every site.

To thwart password re-use attacks, website maintainers have migrated from storing plaintext passwords to storing hashes of passwords. When a user selects a new password, the website will hash the password using one of the secure hashing algorithms, like MD5, SHA256 or SHA512. When the user authenticates at a later time, the website will compute the hash of the given password with the hash that is stored in the database. If they match, it is highly likely that the user has the correct password. If an attacker can exfiltrate the database of passwords then they can try and crack those passwords offline. Cracking a hashed password means that the attacker will have to guess the password. They can either use brute force, and guess every possible password of a specific length, or they can use rules based on human behavior to guess passwords. Either method increases the difficulty for the attacker to be able to take a password from one website and use it on a different website.

In the creation of public key cryptography has raised the bar for authenticating users (SSH Communications Security, 2018). Users can generate and save a public/private key pair for authentication purposes. They can then store and share a public key on a server. If the security of the server is compromised, then no secret information is lost. This method of authentication is extremely difficult to attack, compared to attacking password based authentication methods used by typical websites.

The World Wide Web Consortium (W3C) is a standards body that creates standards for the Internet and websites. They have created standards for Web commerce, Web Payments and the Web Of Things(W3C, 2019). The W3C has defined the WebAuthn standard(W3C, 2018) for allowing access to hardware devices from Internet websites. These hardware devices provide the ability for a user to authenticate with a website, without sharing private information to the website. The YubiKey(YubiKey, 2018) is an example of a hardware device that can be accessible to websites through the WebAuthN standard. The YubiKey has the ability to use the FIDO2 protocol(Alliance, 2018) to authenticate website users. With FIDO2, a device can generate a random asymmetric public/private key pair. The device can then provide the public key as a method of authentication for the user. The device has the ability to prove that the user possesses the private key, without exposing the private key to the website. Thus, the website does not have any secrets to store. The same public key can be used for multiple websites, without exposing a sensitive key to multiple parties.

During the past two decades, attackers and computer scientists have been leveraging the GPU to do more than processing graphics. The parallel nature of rendering graphics has lead to a vast increase in the number of cores on a graphics card. In 2009 graphics cards had 8 cores and by 2017 a high end graphics card has nearly 4000 cores(Wikipedia, 2018a) (Galloy, 2018). Figure 1 shows the vast increase in GPU cores over the past decade. This vast increase in computing power by graphics cards has lead to their use for other purposes. Such examples include the solving of compute intensive parallelizable tasks, such as the simulation of biochemical molecules(NVIDIA, 2018-05), machine learning (NVIDIA, 2018-04) and the cracking of password hashes. The Hashcat program (Hashcat, 2018-05) is a password cracking program that can utilize ei-

ther a CPU and/or a GPU to crack password hashes. Hashcat has the ability to use either

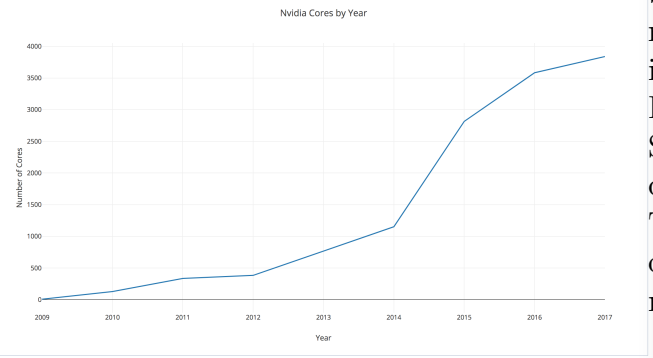

Figure 1. GPU Cores by Year

brute force attacks, dictionary attacks or a combination of the the two. If an attacker has the ability to acquire a password hash stored on a website, then they can attempt to crack that password. There are two standard methods that a website can use to hash a password. The first is to run the password through a secure hashing algorithm. Then when the website attempts to authenticate the user, it will hash the password provided by the user and see if the hashes match. The problem with this method, is that if multiple users use the same password, then their passwords hashes will be exactly the same. Thus the attacker will only need to attack each hash once for all the users. To make attacks harder, websites are encouraged to salt their passwords. With salted hashes, each user of a website has a unique salt stored on the website(Wikipedia, 2108). The website then concatenates the salt with the password and then hashes the concatenated string and stores the salted hash. The stored salted hash will be different for every user, even if those users have the same password (each user has a unique salt). Thus, the attacker will have to attack each salted hash, which will more difficult than attacking unsalted password hashes.

2. BACKGROUND

The OTG password scheme was developed by Steve Gibson of Gibson Research Corporation (GRC). The description of the system is located on its project page(Gibson, 2018). He has designed security systems such as the Shields Up port scanner, a Spectre and Meltdown detector and a DNS benchmark tool. The OTG system is designed to be printed onto paper and then used off line(thus the name).

The OTG password system works by generating a $26x26$ Latin Square. A Latin Square is a NxN matrix where every row and every column have exactly one value from 1 to N. The popular Sudoku puzzles are a more restrictive instance of a Latin square. The OTG password system generates a $26x26$ Latin square where every cell contains a single letter of the alphabet. Every OTG square generated by the GRC website is randomly generated. An OTG user will then print their unique square for use in generating passwords. The OTG website provides directions for the users to use the OTG algorithm, which we will briefly describe below.

2.1 Off The Grid Usage

After the user has created and printed a grid, they are ready to start generating passwords for websites. The first step is to trace through the grid to locate your starting location. Figure 2 shows an illustration of tracing the amazon.com domain through the grid. The user locates the “A” (first letter of Amazon) in the very top row of the grid. They then locate the next letter of the domain “m” and find its location in the same column as the beginning “A” was located. The next letter is Amazon is an “a”. The user switches directions and moves horizontally in the same row as the “m” to locate the next “a” in that row. This process continues

until the user has traced through the entire domain. We will call this location where the user finishes tracing the domain the *starting point*. The *starting point* is then used to generate the password for the website at that domain.

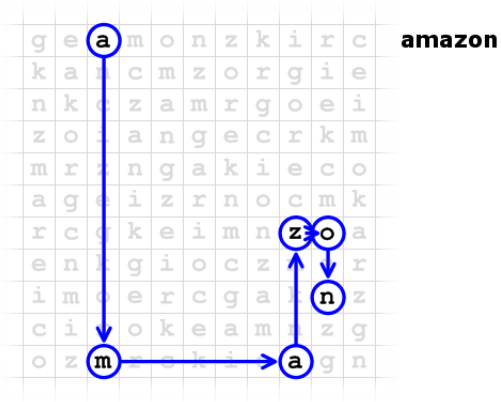

Figure 2. Tracing through the grid

After the user has located the *starting point* they then trace through the grid to generate the password. An interesting note, is that the starting location is the last letter of the domain name. The user then locates the starting letter in the domain name. In our example, the user ended at the "n" in Amazon and now they will locate the starting "A" in the same row. This example is illustrated in Figure 3. After the user has located the "A" in that row the first 2 characters of the password are generated. In this example the "A" is followed by a "gc". The OTG system defines that first to letters of the users password are now "gc". The user then continues tracking the domain, starting at the ending letter of "c" and they switch from moving horizontally to moving vertically. They then locate the next letter in the domain which in this case is the "m" in Amazon. The next two letters of the password are the two letters that follow the "m" in in that column, which is "nz" in this example. This process continues until the user has traced through the entire domain name. In this example the password for Amazon is "gcznegmacmzg".

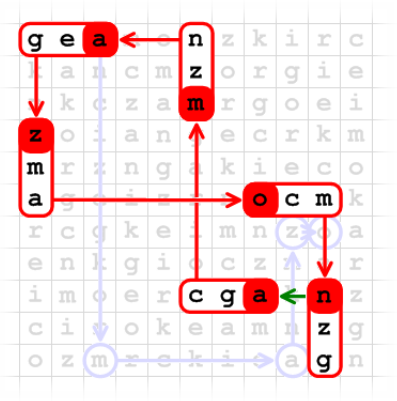

Figure 3. Generating a password

2.2 Special Cases

There are a few special cases that occur in the OTG system. Case 1 is when tracing the grid, the user can reach a letter that is at the edge of the grid. The user then can use the "overshoot" method to wrap around the grid. Figure 4 shows how the generation of a password can wrap around the the grid. Case 2 is when the domain has two consecutive letters in a row (aka Facebook). The OTG website does not specify how to handle this case, but for this case we assume that the user generates two letters to either side of the letters in the same row or column (depending on the current direction of the trace).

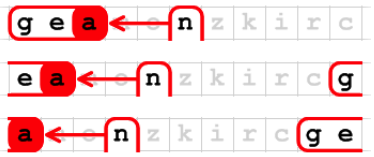

Figure 4. Wrapping Case

3. ATTACKING GRIDS

When attacking the OTG system we assume that there are websites that have OTG passwords leaked. The OTG author assumes that if the passwords to a set of websites are leaked, then the security of other websites will **not** be affected. Our assumption is that there exists a set of passwords that are leaked. In particular, we will have a pair of inputs, the leaked websites domain name and the leaked password, we coin this information a Domain Name Password (DNP) pair.

The simplest method to attack the OTG would be to try and generate all possible Latin squares. But as the OTG website states, there are 9.337 x 10^{426} possible Latin squares. This type of attack would be infeasible to use. Thus a more targeted approach is needed, if this system can be attacked.

Figure 5. Example password Generation

We started with the information that we can glean from the password generation algorithm. We will assume that we are starting with a single users Latin square. In Figure 5 we can see that when a user generates a password using the domain name, the last letter of the domain is followed by the first letter of the domain name. Additionally, we know that the first two letters of the password immediately follow the first letter of the domain name. Figure 6 shows the restrictions on the passwords that are generated. To indicate the restrictions on a Latin square the arrow notation will be used. if $A \rightarrow BCD$ is shown, that will mean that A is followed by zero or more characters in a row and the symbols BCD will be consecutive in the users Latin square. Consecutive characters will adhere to the wrap around effect mentioned in Section 2.2 and thus BCD are logically consecutive. In the example in Figure 5 we know that $N \rightarrow AGC$ is a restriction on all valid Latin squares that could generate the password for this domain name.

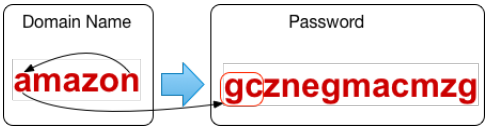

Figure 6. Latin Square Restrictions

A second example that restricts the number of Latin square is in the wrap around effect that can occur (Section 2.2). It is possible both letters from the domain name to be on the edge of the grid. If this is the case, then the wrap around will include the original letter in the domain name. Figure 7 will illustrate this issue. Suppose the we are starting in column 1 on the letter "A". The next letter in the domain is the "M" which is in column N (or the last column of a NxN grid). Due to the wrap around rule the restriction generated will be $A \rightarrow MAB$ and thus we know that A and M are on the border of the grid. This restriction severally limits the number of possible grids that can create this domain name password combination. In this example there are only 2 possible locations for the letter A and M (in that row). There can also exist the combination of $A \rightarrow MBA$, where the starting domain letter is the last letter in the restriction. In this case there are 4 possible locations for the restriction to be located.

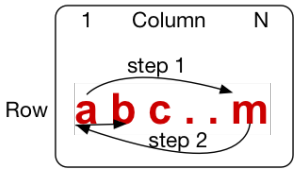

Figure 7. Wrap Around Condition

In addition to the restrictions above we also can glean information from the chain

of restrictions. Each step in the generation of the password will leak information about the set of possible Latin squares. In Figure 3 we have the restriction of $N \rightarrow AGC$. We also know that the column of the letter "C" is the starting point for the restriction $C \rightarrow MZN$. This process repeats for the entire chain of the password. Additionally, we observed that domain names that contain an even number of characters, will initially start generating passwords where the user is moving horizontally through the grid. If the domain name has an odd number of characters, then the user will start by moving vertically through the grid to generate passwords.

3.1 Testing Model

Our approach is to start with Latin squares of a size where there is a known exact number of Latin squares possible. For our tests, we started by using Latin squares of size 10. To represent domains and passwords the letters A-J will be used. For a $10x10$ Latin square there are 9.98 x 10^{36} possible total squares(Wikipedia, 2018b). We chose a random Latin square to use as the base for which to generate passwords from (known as grid0). After the initial Latin square was generated, a set of random domain names was created. For each of the domain names the process described in Section 2.1 was used to generate the password for that domain using grid0. The domain name and password pairs are stored in a plain text file. For this project we generated a random set of 12,288 DNP pairs.

4. LEVERAGING CUDA

The original solutions to solve this problem were written using multi-threading in the Java language. Java is a compiled language, but the speed of operation for a Java program is significantly slower than running the same program in the C language. Additionally , typical computing systems have between 4 and 24 different cores that can run in parallel. Due to these limitations, the prior implementations were not very efficient at solving the Latin squares.

With the prolific rise in the number of cores that can be integrated onto a single GPU the design of the project focused on designing a system that would be able to leverage a high number of cores. The current design requires at least 840 cores for the initial phase, but this number is configurable. This number is based off of the number of possible grids that are generated by a single brute force attack on a 10 x 10 Latin square. Additionally we focused on developing a CUDA based application, which runs on an Nvidia graphics card. CUDA is an extension of the C programming language that allows a developer to write classic C code, with the ability to write code that runs on a GPU. CUDA allows the programmer to define code that is run on the host machine (CPU) and on the device GPU. There is an interface provided by CUDA that allows host code to interact with the device. CUDA also provides a mapping between RAM and the device memory. This allows the host and the device to seamlessly synchronize and transfer data. The CUDA drivers take care of the actual copying of data over the bus that connect the host machine to the graphics device.

4.1 CUDA Implementation Issues

CUDA has the advantage of having a vast number of raw cores that can be leveraged by an application. This does come at the cost of having a much lower amount RAM that can be accessed by the graphics card. Typical cards have the ability to address between 4 and 8 gigabytes of RAM. This is con-

trast to the fact that CPU's can access to an excess of 128 gigabytes of RAM. The initial designs of the CUDA software were designed recursively. The depth of the recursion was only a few hundred calls deep, but this type of design would crash the CUDA software. The design was then switched to maintain the state in RAM. The RAM is maintaining by using the CUDA Unified Memory Model. This model allows the programmer to allocate memory in both host and device levels, and then synchronization is implemented by the compiler and provided by the CUDA driver software.

The implementation of the system also had to take into account that a minimal amount of state could be saved between. The limited amount of RAM available to the GPU forced a depth first approach to solving the Latin squares. The state was stored for all the previous computations as if the system were implemented recursively, but without the overhead of calling functions.

4.2 Implementation Details

The starting point of the software was to read in the domain name and the password associated with that domain name. This data was then converted from text to a linked list data structure shown in Listing 1. This structure holds a single restriction $A \rightarrow BCD$ in a compact form. The integer representation of the characters are stored in the *letters* array. The direction character stores if the restriction is either a horizontal restriction or vertical restriction. The *Path* structure is a linked list, where each restriction points to either the next restriction, or points to *NULL* if it is last last one in the list. This data structure is created one time during the reading of the DNP pairs from the file.

Listing 1. Path Structure

```
typedef struct Path
{
struct Path * next;
char direction;
int letters[4];
char * domain;
char * pass;
} Path;
```

The grid structure stores the information about the state of the grid, given the path restrictions. It is shown in Listing 2. The grid maintains a N x N array of characters that represent the ASCII character that is at the current location. If the character is unknown, then a sentinel dash character (-) is stored in cell. This allows the grid to be printed to the console in a easy to read format. The *ok* variable stores the result of if the grid has any errors, when trying to set a character at a particular location. The error occurs when either 1) there is already a different character in that location or 2) that character is already in the current row or column. This can occur for any of the 3 characters when a path is being traced.

One of the major optimizations that occurred when developing this implementation was the use of bit vectors to store the information about the letters in the rows and columns. For example suppose that the program decides that a "C" should set in Row 1, Column 7. A simple implementation would store the fact that "C" could not be in any cell in Row 1, as well as any Cell in Column 7. For a NxN grid, this would require either a list in every cell N elements for NxN cells or it would require a bit vector for every cell NxN. These solutions either take up a relatively large amount of space (on the space limited GPU) or time to go through every row and column to check the restrictions. The optimized solution was to use a single bit vector for every row and every column. This required $2N$ amount of storage for a NxN grid. If the system tries "C"

should set in Row 1, Column 7, then "C" is removed from the Row 1 bit vector and the Column 7 bit vector. Each letter is represented by using a bit mask where each letter is 2^N where N is the ordinal value of the letter in the alphabet. Thus A is $2^0 = 1$, B is $2^1 = 2$ and C is $2^2 = 4$ and so on and so forth. As the maximum size for a Latin square in the OTG system has 26 cells, a single 32 bit integer can be used to store the bit vector. This representation allows system to set the restrictions in O(2) time and it uses O(2N) space to store these restrictions. The implementation uses bitwise *or* to add a new restriction and bitwise *and* to check for the existence of a restriction. These restrictions are stored in the grid using the *row* and *col* fields of the grid structure.

Listing 2. Grid Structure

```
typedef struct Grid
        {
        int *row;
        int *col;
        char ** Cells;
        char ok;
        } Grid;
```

The location data structure is used to store information about where the system is guessing a restriction will go within a grid. This data structure is shown in Listing 3. The x and y variables are used to store where the first letter in a restriction is supposed to be located. In the example $A \rightarrow BCD$ the letter "A" is located at (x,y) within the grid.

The *type* field in the location structure can either be Full or Partial. A Full search type is used to go through every combination of x and y coordinates. This is used for the first restriction within a path. The Full search type is used to generate the primer for the grids. A full search on a $10x10$ grid, that does not have any special case restrictions, generates 840 possible grids. Each of these grids is then distributed to a different core on the GPU during subsequent calls. The rest of the restrictions of the path will all be be partial locations. A full is also used when the next DNP pair is searched.

For both the Full and Partial, the system uses the direction in the path (Listing 1) to determine how $nextX$ and $nextY$ are incremented. For the Partial path, if the restriction in horizontal, then $nextY$ and y will remain constant through the loop and the $nextX$ value will increment through the loop from index 1 to N.

Listing 3. Location Structure

```
typedef struct Location
{
    uint8_t x;
    uint8_t y;
    uint8_t nextX;
    uint8_t nextY;
    uint8_t lastX;
    uint8_t lastY;
    uint8_t type;
    uint8_t edge;
} Location;
```

The *Location* data structure tracks the current location and the system then increments the current location until it has gone through all possible locations. If the system were written recursively, then the state would be kept track of by the recursive calls, but it had to be done manually. The *State* structure was used to store the state information. It keeps track of the current *Grid*, *Path* and *Location*. Additionally, the state keeps track of the information about the total number of iterations and the number of valid grids that occurred during those iterations. This statistical information helps researchers to understand the complexity and the runtime of the system, when cracking Latin squares.

Listing 4. State Structure


```
typedef struct State{
    Grid grid;
    Location location;
    Path * path;
    long count;
    long iterations;
}State;
```

The *State* structure had to be created to accommodate the problem of recursion in CUDA on the GPU. The developer implemented a *State* structure for every depth of the search, such that backtracking to previous states could be accomplished. Every time a valid grid was encountered, the system saves the current state, and then clones the entire state to the next depth. The algorithm then increments the state and performs a check to see if the current state can be incremented. If it cannot be incremented, the algorithm then pops states until it finds one that can be incremented. If there are none, then the algorithm has finished its search.

5. METHODOLOGY

The hypothesis of this research was that the OTG system leaked information with every password generated. To prove this hypothesis, we first needed to create a Latin square and a set of DNP pairs. If the OTG system leaked information, then we should see a lower number of possible grids that would need to be checked, compared to a brute attack on all possible Latin squares.

The process we used to test the Off The Grid system was to generate a randomly generated Latin square of size 10. A program was developed that would chose a random set of characters to represent a domain name (repeat letters were omitted). For each domain name, the algorithm would locate the starting point, by tracing the path of the domain name in the Latin square as described in Section 2.1. The algorithm would then generate the password for each domain. The output of this program was the 12,288 unique DNP pairs.

After the DNP pairs were generated, the testing of the system could begin. Many shell scripts were created to automate the task of extracting a particular DNP pair and then attacking it with the GPU password cracking system described in Section 4. The scripts would either 1) run that DNP pair through the password cracking system or 2) combine multiple DNP pairs and run them through the password cracking system.

The password cracking system would generate a small amount of log information as it ran. It would time how long the process took, generate statistical information and it could even verify that the original grid was located at each depth of the search. The statistical information collected would count the number of possible grids that were searched as well as the number of valid grids at each depth. The verification that the original grid was optional and it did not positively influence the algorithms performance at any point.

For the tests that were run we used the NVIDIA GeForce GTX 970 graphics cards. A server was built using 4 graphics cards connected via PCI. These cards fit in a 2U server chassis, though further research shows that 8 or more cards can be connected by using USB-to-PCIE convertors. The password cracking system is designed to use one graphics card at a time, though multiple instances of the program can be run at the same time.

5.1 Single domain password

To verify that the password cracking system was working correctly, all of the 12,288 DNP pairs were run through the system. Figure 8 shows the number of possible grids generated by each DNP pair. The smallest

number of possible combinations was 8 and the maximum was just less then 13 million. The testing at this stage was to run all the grids through the cracking system. The data clearly shows a wide amount of variance between the number of possibility generated by a DNP pair. One property of a Latin Square is that you can rotate the square 90°and the properties of the square remain. Thus, only 1/4 of the grids are a result of the original square. Thus, the smallest possible number of grids that can be generated is 4. A result of 8 shows that there are two possible grids (each rotated 4 times).

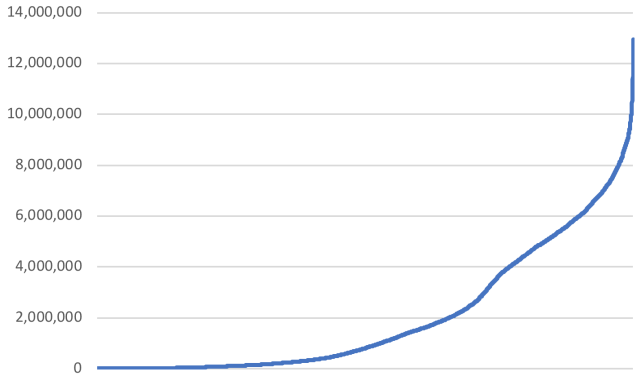

Figure 8. 1 password, Number of Possible Grids

Figure 9 shows another view of the data generated for all 12,288 DNP. The filled portion of the grids ranged from 11 to 19 percent with 1 domain name password combination. For a six character password, more than 90% of the grids (11,104/12,288) yielded 16 character from the final grid while over 51% (6,292/12,288) of DNP generated 18 or more characters. This data shows that on average a 6 character password leaks 18 or more characters of 100 characters in the grid.

5.2 Two domains

The data presented in Section single show encouraging results. There is data leakage for each password generated. Given that

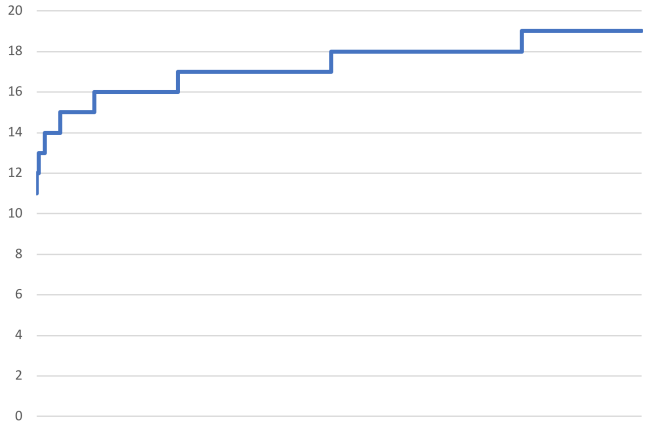

Figure 9. 1 password, Percentage of Grid Filled

there are 12,288 passwords generated for the first stage, it would not be feasible to generate all possible grids pairs in a reasonable time frame. For example an average grid #10052 had 3,637,872 possible valid grids and it took 13.96 seconds to run that single test. When this is multiplied by every possible grid that we could combine it with, then the time to run all of these grids would not be feasible at this time given the current hardware in our lab. As the goal of this paper is to pinpoint the uppermost bound on the amount of data leaked by passwords encoded using this scheme, we opted for a more targeted approach. The grids that produced the smallest number of grids were combined together as a set of 2 of DNP pairs.

By combining the smallest grids, the time required to combine these DNP was minimized. Each grid was combined with all the other grids in the top 40 (excluding itself). The test generated 1560 (40x39) different sets of 2 grids that were combined, and then ran through the password cracking algorithm. The top 40 grids had between 8 and 88 different valid grids and the filled percentage ranged from 11% to 15%.

Figure 10 shows the results from generating the possible grids for the combination

of two of the top 40 smallest DNP poirs. The number of valid grids ranged from 4 to 560. As the grids in this test run were much smaller in size, the number of possible grids had a much smaller variance than in the first test.

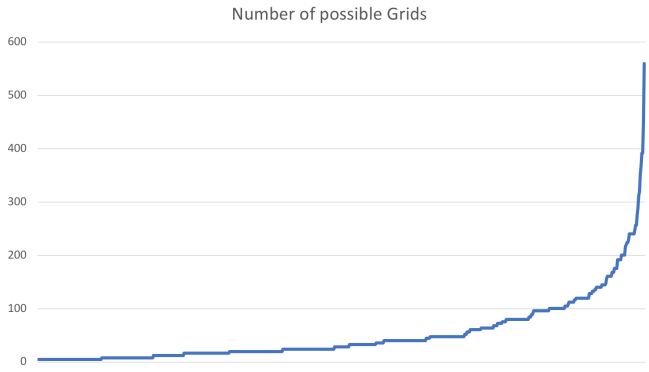

Figure 10. 2 passwords, Number of Possible Grids

Figure 11 shows the percentage of the grid that is filled in the same data set. The percentage ranged from 13% to 28%. Again, we can see that data is still being leaked by the use of the OTG password system. The results of this test were quick and we could see that running pairs of DNP through the algorithm yielded more information than running a single DNP through the system.

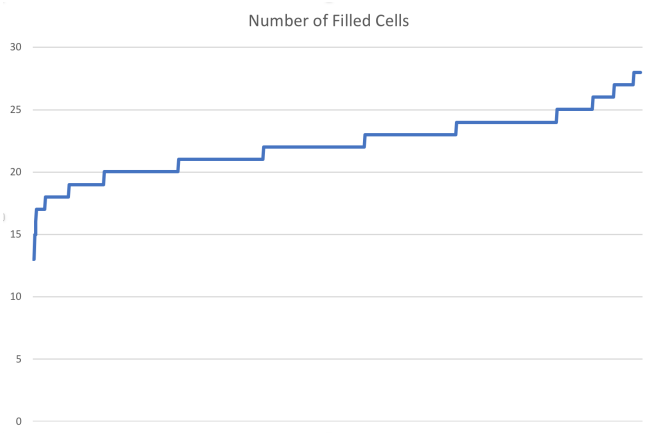

Figure 11. 2 password, Percentage of Grid Filled

Analysis was done on the top 40 smallest grids, and the results were intuitive. The constraints shown in Section 3, especially in the wrapping situations yielded the smallest grids. It is clear that if there is a wrap around that occurs, then the resulting password yields a much smaller set of possible grids than any other password. Thus these passwords were weak, and they would shrink down the state space involved. But, the passwords that were in the top 40 would often combine not just 1 wrap around password, but often 2 or 3 of these in succession. This would force the number of possible valid grids to be very small.

5.3 Optimal Selection

The above approach did not yield a set of optimal results. Section 5.1 showed that for every character in a 6 character domain name we netted 3 characters worth of information (51% of grids yielded 18 characters or more). However, in Section 5.2 we saw that the maximum number of characters of information leaked was 28. This means for the two DNP pairs, we yielded 2.3 characters worth of information per character in the passwords. This result is due to the selection of small grids over optimal grids that would be combined. The optimal set of grids to combine would be grids that do not contain any overlap with one another. Then each grid would yield the maximum amount of information when run through the password cracking algorithm.

To test our theory about selecting the optimal grids, a group of python programs were written to extract the final grids from the log information created by the password cracking system. These scripts would get the final grid's output and then combine it with the final output of other 12,287 grids.

Running the python script on pairs of DNP pairs, the output resulted in over 48,000 combinations that yielded 38 char-

acters when combined. This result shows that if these DNP pairs would be combined, then the for every character in the two domains, we yield 3.166 characters of information leaked. Running one of the optimal pairs through our algorithm yielded the result in 1.5 days, running on a single graphics card.

Another python script was created to attempt to combine the results of combining 3 DNP pairs together. The results of the script were narrowed down to print only the grids that had more than 55 characters in the grid. The data shows that more than 13,000 of the combined grids had either 56 or 57 characters filled in with the grid.

5.4 Estimation of Possible Combined Grids

Following further analysis of the data, it was found that one could estimate the number of total grids that the GPU would have to check for each password combination. To do so, one would start by parsing the output and obtaining the total grids checked and number of valid grids for each individual password in the desired combination. The combination of 2 grids (Grid A and Grid B) is performed by first computing valid grids for Grid A and then computing all possible grids that could occur for Grid B. A pessimistic estimate is to compute the valid grids for Grid A, and then multiplying that by the possible grids for Grid B. The computation is done for Grid A -¿ Grid B and then in reverse Grid B -¿ Grid A. The most accurate estimate is the minimum of the two products.

For example, in Grid #1734, it was computed that there were 80 possible valid grids after computing a total of 6,688 possible grids. In Grid #4773 there were 64 total possible valid after computing a total of 86,808 possible grids. When these two grids were combined, our program computed an estimate as shown above. The calculation is performed both directions to extract the smallest estimate. For this example, our program would take $(80 * 86,808)$ and $(64 * 6,688)$ to obtain our two estimations. The calculation of Grid #4773 followed by Grid#1734 (6,688 x 64) gave us the smaller estimation of 428,032, so we considered it to be the most accurate estimate for the combination of the two grids.

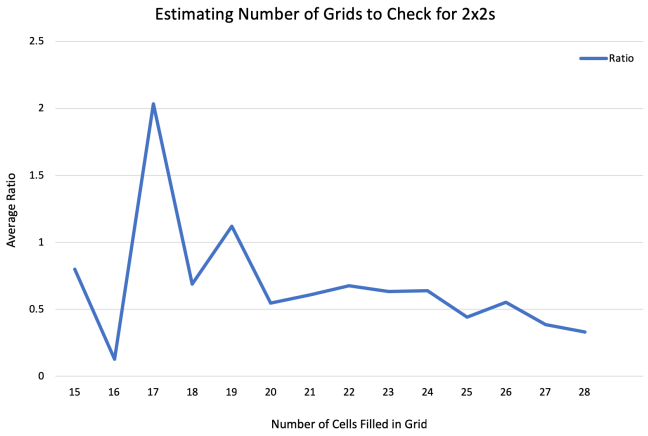

Figure 12

A test of combining Grid #1734 with Grid #4773 found that it checked an actual total of 162,244 grids. These calculations were performed on the top 40 smallest grids. Each top 40 grid was combined with every other top 40 grid. We then calculated the ratio of the actual findings with our estimation, it showed that our approximation was generally an overestimation. In general, the actual number of grids checked was only 65% of our estimate. Figure 12 shows the result of running the algorithm on the top 40 grids. This is ordered by the number of cells filled. The graph includes grids filled from 15 to 28 cells in a 100 cell grid.

5.5 Example Bounds on Filled Grids

A Latin square that is filled more than 1/2 full can become fully solvable. We were able to compute fillable squares given the state of a Latin square. For example when the grids #5601,#5939 and #4323 were combined, we yielded a result of 57 unique cells in the Latin square. By using an algorithm that checks to see if a cell has only one possible value, we were able to complete 96 cells in the grid. This leaves 4 cells empty where there are two different possible solutions.

-	e	a	g	i	f	b	d	-	c
c	g	h	a	j	e	i	b	d	f
f	j	d	e	h	c	g	i	b	a
e	a	g	h	f	j	d	c	i	b
g	d	j	i	c	b	a	f	e	h
b	f	i	c	g	d	j	h	a	e
d	c	e	b	a	g	h	j	f	i
a	i	c	j	b	h	f	e	g	d
-	b	f	d	e	i	c	a	-	g
i	h	b	f	d	a	e	g	c	j

6. RESULTS

The results of our password cracking system showed promise, as we were able to leak information from the OTG Latin square used to generate website passwords. The results of running a single DNP pair through the password system was very quick. The time for a single run ranged from 0-16 seconds with the average taking 4.4 seconds.

The results of combining two different top 40 smallest possible grids resulted in grids that only had 4 possible grids, which is essentially one grid that is rotated 4 times. The average number of grids that had to be checked with 2 small grids was 143,801. The minimum number of grids checked was 25,884 and the maximum number was 559,504. The grid that resulted in the maximum number (Grids #8016 and #8273) took only 11 seconds to run and it resulted in 23 of the cells in the grid being filled. We also had a combination of Grids #11 and #4797 yielded 25 filled cells with the number of grids checked at 66,880.

Comparing the results from the above two tests to the computed number of Latin squares shows a stark difference. For all possible Latin squares of size 10 there are $\approx 9.9\text{x}10^{36}$ possible Latin squares. We were able to compute 25% of the grid using only $6.6 * 10^5$ computations. Thus if there were two more edge case grids that were optimally combined, 50% of the grid could be filled. This could possibly allow for a grid to be solved in about $1.2 * 10^6$ computations. This is 30 orders of magnitude less than the total number of Latin squares of size 10.

7. CONCLUSIONS AND FUTURE WORK

Our hypothesis was that given a set of Domain Name Password pairs, it would be possible to gain information about the original grid. We have created a set of constraints, based on the design documents provided by the creator of the Off The Grid, that can be used to reduce the state space of possible Latin squares. We have implemented several algorithms that represent and attack the different facets of the OTG system. We have given evidence that it is possible to reduce the amount of work required to crack the Latin squares used in the OTG password system. We believe that given the calculations shown in Section 6 we can see that it is possible that the use of Latin squares as a password generation scheme can be compromised given a small set of passwords. We have shown that character in a domain name generates about 3 characters of the original Latin square. If we extrapolate this out to a full size 26x26 (676 cells) Latin square it would take about 19 domain name password

pairs to fully crack an OTG system. (676 / 2 (fill 50% of grid) / 3 (3 characters per domain character) / 6 (6 character password)).

7.1 Security Considerations

Given the above evidence, we would not recommend using the OTG for generating passwords for websites. We have shown that there are possible weaknesses in the system that could allow the original grid to be recovered. In addition, there is not a method that can be used to reset a single password if the password is lost. The loss of passwords occurs on a regular basis as security breaches have increased considerably over the past decade. Finally, OTG does not support multiple logins names for websites. There are a variety of reasons for having multiple accounts and the OTG system has no mechanism for multiple logins. Additionally, standards like WebAuthn provide a much higher level of security using asymmetric encryption.

7.2 Future Work

The first area we would like to pursue more research, we would like to generate more possible passwords to find more weak DNP pairs. These weak DNP pairs can be used to find a closer approximation to the theoretical bounds that can be used to attack the OTG system.

The second area of future research is the integration of the entire process into one seamless tool. Currently there is a CUDA C Program that implements the parallel password cracking attacks. There are Bash scripts that parse the output of the password cracking and produce CSV outputs which can then be parsed and read using Excel. Those results are then manually sorted and used to create the next stage of computation. Additionally, for the optimal computation, we utilize Python scripts to combine the grids to get a quick handle on which combinations will yield grids the largest portion of a grid to be filled, which is then run through a Java program that finds constrained cells and fills them in. All of these programs should be combined into 1 streamlined system to make the process seamless and more efficient.

The third area of research is to expand the size of the grids that we will do processing on. Currently we used a size of grid where we have an exact size on the number of possible grids. We would like to expand the research to attack grids of size 15, 20 and 26, where 26 is the ultimate goal of the research. This may require additional hardware and modification of software. The password cracking software is mostly written to accommodate larger sizes of grids, but the generation of an initial set of grids and the distribution of those grids to blocks of cores depends on the size of the input, the number of cores on a GPU and the available RAM on those GPU's. This process would require some fine tuning, but should be a straightforward task.

REFERENCES

Alliance, F. (2018, 05). *Fido2 project.* https://fidoalliance.org/fido2/. Retrieved 2018, from `https://fidoalliance.org/fido2/`

Galloy, M. (2018, 04). *Cpu vs gpu performance.* Retrieved 2018-04-17, from `http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html`

Gibson, S. (2018, 04). *Off the grid.* Retrieved 2018-04-17, from `https://www.grc.com/offthegrid.htm`

Hashcat. (2018-05). *Hashcat, advanced password recovery.* Retrieved 2018-04, from `https://hashcat.net/hashcat/`

NVIDIA. (2018-04). *Gpu-accelerated tensorflow.* Retrieved 2018-05-01, from https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/tensorflow/

NVIDIA. (2018-05). *Gpu-accelerated gromacs.* Retrieved 2018-05-06, from https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/gromacs/

SSH Communications Security, I. (2018, 05). *Ssh (secure shell).* Retrieved 2018, from https://www.ssh.com/ssh/

W3C. (2018, 05). *Web authentication: An api for accessing public key credentials level 1.* Retrieved 2018-05-06, from https://www.w3.org/TR/webauthn/

W3C. (2019). *Web of things.* Retrieved 2019, from https://www.w3.org/WoT/

Wikipedia. (2018a). *Geforce 10 series.* Retrieved 2018-04-17, from https://en.wikipedia.org/wiki/GeForce_10_series

Wikipedia. (2018b). *Latin squares.* Retrieved 2018-04-17, from https://en.wikipedia.org/wiki/Latin_square

Wikipedia. (2108, 05). *Salt (cryptography).* Retrieved 2018-05, from https://en.wikipedia.org/wiki/Salt_(cryptography)

YubiKey. (2018, 05). *Web authentication: An api for accessing public key credentials level 1.* Retrieved 2018-05-06, from https://www.yubico.com/

A. APPENDIX 1. RESULT OUTPUT

The top 40 smallest Domain Name Password Pairs are shown in Table 1. Tables 2 and 3 show the results of combining pairs of 2 DNP pairs from Table 1 with each other. These tables only show the pairs that have only a single unique result grid (4 possible grids with rotation).

Table 1. Top 40 Smallest Domain Name Password Pairs

Grid Number	Number Filled	Valid
503	12	8
5419	12	8
4797	14	12
8179	12	16
11873	13	16
2954	13	24
5129	12	24
645	14	32
16	13	40
1978	13	40
2606	15	40
9839	14	40
11357	14	40
9159	14	44
758	13	48
1749	14	48
5581	13	48
8997	13	48
11181	13	48
8788	13	52
4648	15	56
4773	15	64
2430	11	72
11	13	80
933	14	80
1373	14	80
1582	13	80
1734	14	80
3143	14	80
5421	14	80
6221	13	80
6674	13	80
7409	12	80
8276	13	80
8016	14	84
1620	13	88
8064	12	88
8273	13	88
9347	13	88
10969	15	88

Table 2. Results combining Two Domain Name Password pairs

Grids	Filled	Total Possible	Valid
503-5419	20	42248	4
503-4797	21	27632	4
503-8179	24	31432	4
503-11873	20	44392	4
503-2954	19	25884	4
503-5129	21	30632	4
503-11181	21	66920	4
503-4648	24	64152	4
503-933	20	44324	4
503-1620	24	62312	4
503-10969	22	27232	4
5419-4797	16	29280	4
5419-2954	18	28260	4
5419-16	18	121920	4
5419-1978	18	121920	4
5419-758	17	46040	4
5419-1620	19	68880	4
4797-8179	22	51880	4
4797-11873	20	74680	4
4797-2954	17	46772	4
4797-5129	21	51560	4
4797-645	21	46820	4
4797-16	20	141440	4
4797-1978	20	141440	4
4797-9839	23	60128	4
4797-4648	21	85880	4
4797-11	25	66880	4
4797-933	21	57180	4
4797-1734	20	48604	4
4797-6674	21	79960	4
4797-8064	20	46820	4
4797-10969	20	47760	4
8179-16	24	270920	4
8179-1978	24	270920	4
8179-9347	20	30756	4
11873-2954	20	30132	4
11873-11181	18	95704	4
11873-1620	24	88680	4
11873-10969	22	34200	4
2954-5129	21	44844	4
8064-10969	22	92512	4

Table 3. Results combining Two Domain Name Password pairs Continued

Grids	**Filled**	**Total Possible**	**Valid**
2954-16	19	213124	4
2954-1978	19	213124	4
2954-9839	21	107892	4
2954-4648	21	103684	4
2954-11	22	68724	4
2954-933	20	50184	4
2954-8064	20	27356	4
2954-10969	19	28524	4
5129-16	21	345440	4
5129-1978	21	345440	4
5129-10969	22	37200	4
645-10969	22	37744	4
16-11181	20	409008	4
16-4648	21	362800	4
16-1734	23	228624	4
16-1620	23	396880	4
16-10969	23	217840	4
1978-11181	20	409008	4
1978-4648	21	362800	4
1978-1734	23	228624	4
1978-1620	23	396880	4
1978-10969	23	217840	4
2606-8276	23	126640	4
9839-9159	20	230756	4
9839-8273	21	302316	4
9839-9347	21	183700	4
9839-10969	24	178636	4
9159-4648	21	182480	4
9159-10969	22	91520	4
758-1582	18	184792	4
1749-1734	24	77912	4
11181-11	24	179320	4
11181-933	19	123260	4
4648-1373	25	450960	4
4648-8273	22	282200	4
4648-9347	21	252144	4
4773-11	23	322288	4
2430-8273	20	160840	4
2430-9347	21	91424	4
3143-10969	22	112344	4
8016-8273	23	559504	4

