Ditto: Identity Duplication using Keystroke Dynamics

Dylan Bettermann
bett0159@umn.edu

Andrew Helgeson
helge206@umn.edu

Brian Maurer
maure113@umn.edu

Kevin Mehlhaff
mehlh014@umn.edu

Ethan Waytas
wayt0012@umn.edu

ABSTRACT

Keystroke dynamics is a biometric measurement which uses typing patterns to identify users. In this research we explore the efficacy of keystroke dynamics countermeasures through mimicry. We confirm keystroke dynamics ability to uniquely identify users in normal scenarios. The use of keystroke dynamics is evidenced by the available software solutions and the products utilizing them. Keystroke dynamics is currently used in some systems as passive identification mechanism to enforce per-user licensing restrictions. We explore a counter-measure to keystroke dynamics through the scenario of a business who wants to reuse licenses for potentially expense software. We do so by profiling user's typing patterns and partitioning them into groups where everyone in a group is mimicked under one profile. The goal of our countermeasure is to fool the keystroke dynamics software into believing all users in the partition have the same keystroke profile.

General Terms

Measurement, Performance, Experimentation, Security, Human Factors, Standardization, Languages, Verification

Keywords

Ditto, Masada, keystroke, licensing, identification

1. INTRODUCTION

Keystroke dynamics is a very active research field with Google Scholar reporting over 900 publications involving keystroke dynamics since 2011. Research has focused on a broad range of uses for keystroke dynamics including primary user authentication [2][16], passive intrusion detection [13] and even detection of emotional states[1]. Recognizing the human difficulty in memorizing *secure* passwords, DARPA is seeking to utilize keystroke dynamics to potentially remove passwords [14]. While not ready for use by DARPA, there are plenty of software solutions available such as KeyTrac which lowers the entry barrier for developers. With the expansion of this niche biometric market the security of these solutions should be analyzed more closely.

1.1 Threat Model

Much of the current research focuses on measuring the efficacy of keystroke dynamics for authentication by looking at false positive and negative rates. These tests are important but they don't look at the effectiveness of keystroke dynamics in the context of an attacker intentionally trying to type as someone else. The goal of our research is to explore the strength of keystroke dynamics in the face of a mimicking attacker using a software aid. Next we discuss two motivating scenarios for this research.

1.2 Monetary Motivation

Monetarily, there exists the scenario where a software product uses keystroke dynamics to enforce a per-user licensing restriction. For this research consider the following scenario. A company of 100 employees has purchased 10 licenses for access to an online database which all employees will use. Assuming this online database uses keystroke dynamics it should raise some red flags when multiple users are using a single licenses. The developer can then request more licenses be purchased and obtain the proper compensation. If all 100 employees could be mimicked under 10 keystroke profiles then the software company would see a large loss in revenue.

It is important to note that an IT staff member with administrative access on all of the 100 user's computers could theoretically implement the above scheme without any of the users knowing.

1.3 Security Motivation

Another concern is security. Consider a special *"secure"* OS that implements a primary password authentication with a secondary passive keystroke identification. The ability to mimic a user could cause unwanted information leak through a breach of security. There are some additional considerations from the attacker's point of view. First the attacker must have some way of passing the primary authentication for a given user. This is outside the scope of this research but we assume the attacker can either find the password or exploit a host vulnerability. If the user has access to the OS as a privileged user then it may be considered unnecessary to also mimic the keystrokes since they could do their damage before the passive identification flagged them. While it is true that the user could wipe memory or steal a few key files before keystroke identification flagged them, there are scenarios where it would be more beneficial to the user to maintain access to the system under the guise of a privileged user to obtain information later on.

There is a technical difference between the monetary and security examples. In the monetary example, the software is running on trusted hardware from the developer's point of view. Our Ditto program operates at the kernel level to capture all keystrokes before they reach any user applications. In the context of a secure OS it would be less likely, except through some exploit, that Ditto could be added at the kernel level of the OS. If the secure OS was running virtualized then Ditto or a similar program could operate at the VMM level.

For the remainder of this research we will focus on the first scenario (online database). We assume that if it is shown that Ditto can successfully mimic users then it will work in the second *secure* OS scenario pending the solution of the technical problems.

1.4 Terminology

This section defines the terminology related to keystroke dynamics and the acronyms that may be used for the brevity of exposition.

- *Keystroke dynamics (KD)*. The analysis of a person's typing pattern as an identification measure.

- *Key Press*. An event where a key is pressed down. This can be delivered either from hardware in such as a keyboard or injected in software.
- *Key Up*. An event where a previously pressed down key is released. Once again this can come from either hardware or software. For the purposes of implementing Ditto a strict 1-1 correspondence has been assumed between down and up key events. In hardware this makes sense because a key cannot be pressed again until it is released but software could operate differently leading to strange results.
- *Fly Time*. A fly time is the time it takes to move from one key to another. Fly times are defined as the time from a key down event of one key to the key down of the next. Notice that this does not imply a key up comes before the second key down. Since multiple keys can be pressed at the same time no assumption can be made about the corresponding order of key up events.
- *Press Time*. A press time is the time from the key down event to the key up event for the same key. This is where the assumption of a 1-1 correspondence between down and up event is necessary to accurately measure times. This requirement has a special case when a key is pressed down which generates multiple key down events and one key up event.
- *Stroke*. A stroke is any key event such as a key up or down event. They represent some change in the state of a key.
- *Keystroke Profile (profile)*. A keystroke profile is a set of fly and press times that are supposed to represent the typing pattern for a given user. In the results, the terms reference profile and base profile are used. This means the profile that we are trying to match against.
- *Raw Profile*. A raw profile is a set of profile data that has just been collected from the user but not reduced. This will have a list of times for fly pairs and individual key press times.

1.5 Layout of Paper

Section 2 will present related research in the area of keystroke dynamics and the statistical models. Section 3 will introduce and provide more detail on each piece of the project from raw data collection to mimicking using Ditto. Section 4 goes over experimental results from KeyTrac and Masada. Section 5 explores future research and section 6 finally concludes.

2. Related Research

While the use of keystroke dynamics has been around for a relatively short time, the use of behavioral biometrics has been employed throughout history. One of the first documented cases was when the telegraph was invented in 1844. Very quickly, it was realized that individual telegraph operators could be distinguished by their unique tapping rhythm. This typing rhythm was used during World War II to tell if enemy soldiers were trying to create false Morse code messages. Ever since then, the realization that a user’s patterns are unique led to a desire to implement a system to accurately differentiate people (similar to a fingerprint).

Unfortunately, during the 1980s, it was found that continuous detection by keystrokes was not enough. Users were inconsistent, typing differently at various times throughout the day. These deviations led to programs that would lock out valid users quite frequently, rendering the defense mechanism useless. It was not until 1999, when Monrose and Rubin [11] suggested that keystroke biometrics could be used as a secondary verification method. In short, a primary method to authenticate would be used (e.g. password), and keystroke dynamics would help verify the primary method. An example of this would be to examine how a user typed in their password, as it was found that short, known phrases were typed relatively consistently over time. This spawned a renewed interest in keystroke dynamics.

As mentioned, Monrose and Rubin first introduced the idea of using keystroke dynamics as a secondary authentication scheme. They stated that its use of a primary authentication scheme was inherently flawed because matching a user with a particular signature depends on his or her typing rhythm and this rhythm does not stay consistent over long periods of typing. The environment can change or the attention span or alertness of the user can fluctuate and affect the typing rhythm. The authors argued that keystroke dynamics combined with a more traditional authentication scheme can lead to a more robust and secure system.

There are two main classes of keystroke dynamics systems: static and dynamic. A static keystroke system takes a length of user input and tries to authenticate against it. This could be used at a login prompt, where the user would type not only their username and password but also some text to sample their keystrokes. Our Masada program used this approach. The other class, dynamic keystroke systems, can be useful when the user is accessing restricted files or is operating in a high-risk environment. These dynamic systems are continuously checking keystroke patterns.

In a 2009 paper Killourhy and Maxion compared a variety of anomaly-detection algorithms to compare collected keystroke data against stored used profiles [10]. They determined that some detectors can have very low error rates such as the Mahalanobis with an equal error rate of only 0.100. We used this algorithm in Masada because of our capability to implement it and its detection quality.

3. Approach

From data collection to profile mimicking we have created a pipeline of tools which will be described in this section.

3.1 Data Collection

We had a few main goals when we initially started collecting data. First, we wanted to have a set of participants type the same passage of text. This would allow us to compare profiles of different participants in a situation where the profiles would be as similar as possible. Second, we wanted to be able to collect fly time and press time data for each participant as they typed the given text. This guided us to use a web based collection interface. Using a website with JavaScript, as each participant typed we would be able to capture all the keyboard key-up and key-down events, calculate the press and fly times, and store the users final data in a JSON object for future analysis.

Initially we were hesitant to use a web based interface as we were uncertain of the timing accuracy of JavaScript in a browser compared to a program that would be compiled and run directly on an operating system. John Resig, creator of JQuery, has done JavaScript timer accuracy analysis and determined that most modern browsers display an accuracy within 5ms [12]. We didn't have much choice, since it would have been nearly impossible to meet with all of our participants physically, and so we decided our proposed web based interface would serve its purpose without ruining our research.

After the data was collected we had 45 participants worth of data, which gave us a large enough pool of profiles to create meaningful partitions. With this we could then begin to test with our profile mimicking implementation, Ditto, against the online KeyTrac software and Masada, our implementation of the Mahalanobis distance. It should be noted that we only gathered one typing sample for each user. To create a more accurate profile, users should be profiled multiple times over several days to cancel out anomalies.

3.2 Profile Reduction

Raw collected profiles are not useful to Ditto since it needs a single time to emulate for flies and presses. The raw profiles contain lists of observed times which much be reduced to single times. The reducer is a set of Python scripts for loading profiles from storage (database in our case), gathering statistics and choosing times based on those statistics. Next we will describe the simple statistical analysis used in the reduction phase. Future work could look at better statistical inference in reducing profiles.

The reducer first does a basic outlier filtering that is 1.5 times the IQR for the set of times in a given fly or press set and throws these times out. This isn't enough because sometimes we collected a single data point for a certain key (or key pair) when the user took a break from typing causing their time to be significantly high. Because of this, after filtering outliers, all times greater than two seconds were thrown out. We make the assumption that even the slowest typists or at least those that would be using Ditto in one of our scenarios would not have a fly or press time greater than this.

The next step of the reducer is to do some basic statistics on the remaining values such as mean, median and standard deviation. For a given key or key pair we reduce the remaining times to the average of those times. There is still a problem with the profile which is missing key times. During collection we do not collect every possible fly time, but we do get most press times for the alphabet. If Ditto does not find a time in the profile it treats it as 0 ms which would look wrong to the user and to most keystroke identification software. To fill in the gaps for fly and press times we select random times around the mean of all fly or all press times respectively. The value we used was ±50% of one standard deviation on either side of the mean where the percentage of the standard deviation was chosen at random.

This approach to filling in times is fairly naïve but will work well depending on your application. If the keystroke dynamics software you are fooling had its reference profile created under Ditto then these filled in times are what it will believe for that stroke. If the reference profile was created while not under Ditto there can be problems as the filled in times will likely be inaccurate to the actual times. Future work could look to improve this keystroke filling by looking at the relative position on the keyboard of the keys that are missing and infer a time from other strokes that are in a similar orientation. This approach would assume that a user's fly or press time is affected by the position of the keys relative to their fingers.

3.3 Profile Partitioning

After raw profiles have been collected, reduced and filled they must still be partitioned. Remember that in our scenario the purpose of partitioning was to emulate a group of users under one profile to only use one license. The motivating idea behind partitioning is the idea of a separation between the sensitivity of a computer's timing measurement and a human's visual perception. If a person typing at 50 wpm is not able to notice when they are being delayed to 45 wpm then you know they can both exist in the same partition.

One might ask why we don't just partition everyone into the slowest group and save more money. Through our testing of Ditto it has become apparent that the imposed delay can have a large effect on usability. When typing under a much slower profile than your normal speed it decreases your accuracy as you cannot see what you are typing and increases your irritability towards the system. This means that partitioning all users into one group would in fact lead to decreased efficiency which likely would cost more than added licenses. We have considered two simple partitioning schemes for our research, fixed and scaled partitioning. Further research could use usability analysis of perceived user delays while typing to get a better estimate where the true partition ranges lie.

Fixed partitioning defines a set of wpm boundaries that create a set of buckets (6 in our case) which users are dropped into. This has the advantage of allowing us to define how many partitions we have which may be useful in reducing the number of licenses needed. The downside is that fixed partitioning will likely lead to low usability as you stray from the middle of the speed distribution. Very fast typists will notice smaller changes in typing speed and therefore need tighter partitions. On the other end, a user who types at 29.99 wpm may be placed into a partition that has an average of 10 wpm which is a large visual difference. To help fix this problem we implemented scaled partitioning.

Scaled partitioning makes the simple assumption that as typing speeds get faster, smaller partition sizes will be necessary to maintain usability. This means that in an example partitioning we may place all users from 0 to 20 wpm in one group but at the upper end only users from 100 to 105 wpm. This approach attempts to maximize usability but reduce the number of partitions. In practice we found that many of our partitions were empty after running through this partitioner. A future approach could be more dynamic and analyze the distribution of coupled with some usability data to make smart partitioning choices.

3.4 Ditto

Ditto, like its namesake Pokémon character, is tasked with mimicking users but more specifically their keystroke profile. This section explores the technical design of Ditto and the challenges faced in matching input key strokes to profile times.

3.4.1 Technical Design

Ditto is written in C++ using the interception library [15] which is a kernel mode driver for Windows which allows us to hook keyboard input at the kernel level. This is ideal from other userspace solutions since it allows us to capture and modify keystroke timing before nearly any other application sees it. With this we can run Ditto and have the desired keystroke profile applied to anything that is typed on the computer. Next is a brief technical overview of how Ditto works.

Ditto reads in profiles that are in a specific binary format which contains sets (structs) of times. These profiles include both fly and press times and are assumed to be in milliseconds. At startup Ditto will load these times into 2D matrix for fly times and a vector for press times which allows for quick, real time access later.

While running, Ditto uses either two or three threads. The first thread listens to the interception driver waiting for key strokes.

When strokes are received, some limited processing is done and the stroke is stored on a dispatch queue. The dispatch queue is read by the dispatcher who then finds the correct timing for this stroke which would be a fly time for a key down or a press time for a key up. In normal operation, Ditto will send out all strokes in the same order they were received. This can cause unavoidable deviations from a profile's timing but there is a potential workaround using a third thread and interleaving keys which Ditto implements as a configurable option. See section 3.4.3 on Ditto's accuracy for more information on this.

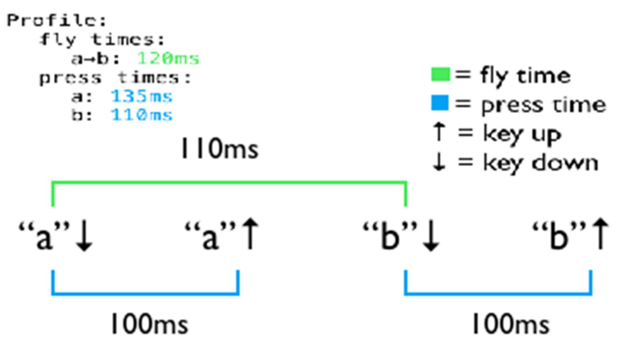
3.4.2 *Timing*

This section discusses more about how Ditto handles timing strokes (just the basic serial ordering). At any point in time Ditto is maintaining two main chunks of information necessary for timing. Those are the last key down and its dispatch time as well as a map of previous key downs and their dispatch time. The first is necessary for determining fly times when we get a new key down stroke. We look at the current key and the previous key to look up in the profile. After a key down is dispatched, an entry is made in a map with the code for the key as the key in the map and the value being the dispatch time for the key down. To look up the press time all we need is the key's code and we can compare that with the time since the key down was delivered.

In the basic serial mode of operation, the Ditto dispatch thread only processes one stroke at a time. If Ditto determines it is too soon to deliver the key then it goes into a thread sleep. The boost thread sleep being used has a claimed precision of 5ms which is unacceptable for our purposes but we are able to observe precision sub 1ms by increasing the scheduling priority of the Ditto application. The accuracy of the thread sleeps is backed up by the timing code which uses the C++11 chrono library showing that we are nearly always within 1ms of the target profile time except when we are limited by the serial ordering (which isn't an artifact or the timing code).

3.4.3 *Core Accuracy Challenge*

There is an inherent limit to Ditto's accuracy with respect to a given profile if we enforce a strict serial order of incoming to outgoing key strokes. Even though keys are always received in a serial order, their timing will be augmented by Ditto. If the user were to actually type at this augmented speed, the ordering of the incoming keys would have likely been different. Consider an example where a user types the letter 'a' followed by the letter 'b'. Figure 1 shows the order and timing the user typed the keys in.



```
Profile:
   fly times:
      a→b: 120ms
   press times:
      a: 135ms
      b: 110ms
```

Figure 1 Example input timing of strokes

The profile times in Figure 1 show a 120ms fly time from *a* to *b* and a press time of 135ms and 110ms for *a* and *b* respectively. Figure 2 shows the timing that Ditto will apply to these key strokes to attempt to match the profile.

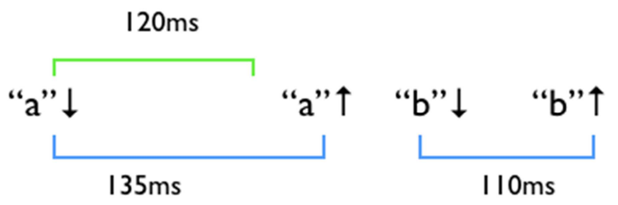


Figure 2 Serial output timing after profile

We made it so that the user finished the upstroke on 'a' before pressing 'b' but this isn't always the case. Ditto will try and delay this press time for 35ms extra to meet the 135ms time for press *a*. If it does this then the fly time for a→b will also be 135ms which is 15ms over the profile time. As mentioned before, if the user were to actually type at the profile's press time for 'a' there would likely be a different ordering of the incoming strokes. Imagine the user's finger on 'a' was artificially held down while typing to meet the 135ms press time (rather than Ditto doing it in software). Despite being potentially distraught about why their finger is stuck to the key they would likely press 'b' before letting up 'a'. Figure 3 shows the new ordering you would have.

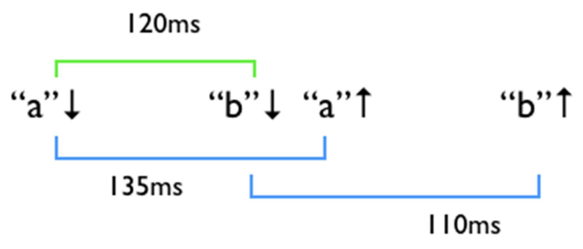


Figure 3 Interleaved output timing

As this displays, when we augment times we should really interleave keys that were previously serial if their times overlap post augmentation. The problem is that Ditto wants to maintain the strict serial order of the strokes to ensure the user gets the same result, even if it sacrifices profile accuracy. One potential way to alleviate this would be to allow for the interleaving of keys, but then we start altering the ordering of keystrokes. In the example above, consider a program that had some special action that occurred when you pressed 'a' and 'b' at the same time. Originally the user didn't do that but after interleaving the keys that action would be triggered. We can still make this a viable option if we consider which keys we allow to be interleaved. Most alpha-numeric keys are not sensitive to strange behavior with this interleaving but there are modifier keys such as shift/ctrl/alt that are important to have in a serial order. If the user types "And" we don't want to hold the shift too long and end up with "AND", this would in fact reduce the profile accuracy since the user's error rate would increase and psychologically they would slow down to avoid these problems.

This serial to interleaving step is necessary if you want to obtain higher profile accuracy but it presents a few issues which have been displayed already. The first is that there can be unwanted side effects from applications interpreting the interleaved state of key events. The other is that you have introduced a new way for a detection tool to rat out Ditto. If an application knows the timings for a profile it can do some simple tests to see if serial keys magically get interleaved. This is difficult since Ditto runs at the kernel level but imagine some hardware chip that is tasked with detecting Ditto which would carry out this keystroke test occasionally.

3.5 Masada

As with the goal of Ditto to mimic a user's typing profile, the goal of Masada is to detect if a person is who they say they are. To do this, a user's typing data is collected in an application and compared to a saved profile.

3.5.1 Design

The motivation for Masada came from the need to test Ditto in two different ways. The first was to make a detector that was roughly comparable to the detector used in the CMU paper [10]. By replicating an existing detection method, Ditto could be tested and considered robust, at least in the sense of reliably fooling a detection method. Second, we needed to consider what a real world developer would do in order to implement keystroke dynamics in an application. This approach required thinking about some of the different constraints that can be placed on an application when being developed for-profit.

See section 3.5.3 for more information regarding our approach to reproduce the CMU paper's method of detection (of course the following also gives insight into building the application).

Regarding the second point, we needed to construct our own program for detection. To do this, we focused on the multiple users under one license example. A software developer, in order to increase revenues, might require exactly one user per license. The software developer would likely monitor a user's typing pattern and determine if multiple people were using one license. There are some caveats, however, that we took into consideration.

First, it is likely that the developer would only be able to track keystroke information inside the program, no outside key tracking would be allowed (of course this would not be an issue for Ditto, as the entire operating system runs under one keystroke profile). This constraint is valid as a company would not want some arbitrary program collecting keystroke information.

Second, it is likely that the developer would have users create a typing profile to compare against (rather than building one on the fly). The program would load a profile based on the login of a user and then determine by some method if that user was valid.

Third, the method used to determine a valid user would likely be implemented by the developer, as opposed to buying off the shelf software due to cost and/or integration issues. Therefore, we determined that Masada should be a self-contained program that tracked typing only while using the application, and that the application would determine if a user was valid.

Some articles from IBM provided the basis for Masada [4][5][6]. There were three main articles, and each was related to a different approach of detecting a user by their typing. Specifically, we focused on the article about continuous detection, as that is what we wanted to implement for Masada. In the continuous detection article, we focused on how the author collected the key strokes and fly times. We also looked at how the distance between a base profile and collected data was analyzed, but we found that to be insufficient. The author would collect data, create a range, and then try to find a match by hashing the collected data and comparing to the base profile hash. Based upon research [10], this method appeared to be relatively weak, hence the use of a more sophisticated calculation was needed (see 3.5.3).

3.5.2 Layout

The layout of Masada is roughly as follows:

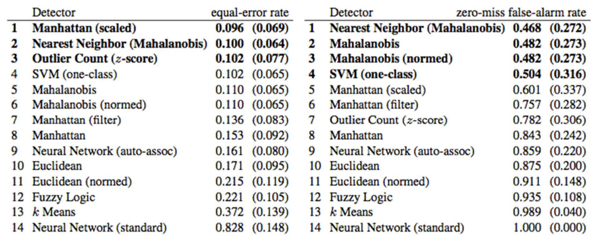
- Login page to choose a user profile. This profile is loaded into Masada.
- Methods to track both the press and fly times of all keys a user types while using Masada.
- A method to determine the distance between a user's base profile and the data collected during runtime.
- Masada was written in C# using Visual Studio. The target operating system was Windows 7.

The first two functionalities of Masada are straightforward. A user's previously collected profile is loaded into the program. While a user types, all press and fly times are collected and saved for later analysis. The third functionality is where it is determined if a user matches the loaded profile.

3.5.3 Distance

As mentioned previously, the third functionality is to determine the distance between a user's base profile and the data collected in Masada. In order to do this, we need to find how different (or similar) a user's base profile is as compared to the data collected in Masada while using it. There are a number of different ways to calculate this difference, as shown in the below table:

Table 1 Comparison of keystroke distance algorithms



	Detector	equal-error rate			Detector	zero-miss false-alarm rate	
1	**Manhattan (scaled)**	**0.096**	**(0.069)**	**1**	**Nearest Neighbor (Mahalanobis)**	**0.468**	**(0.272)**
2	**Nearest Neighbor (Mahalanobis)**	**0.100**	**(0.064)**	**2**	**Mahalanobis**	**0.482**	**(0.273)**
3	**Outlier Count (*z*-score)**	**0.102**	**(0.077)**	**3**	**Mahalanobis (normed)**	**0.482**	**(0.273)**
4	SVM (one-class)	0.102	(0.065)	**4**	**SVM (one-class)**	**0.504**	**(0.316)**
5	Mahalanobis	0.110	(0.065)	5	Manhattan (scaled)	0.601	(0.337)
6	Mahalanobis (normed)	0.110	(0.065)	6	Manhattan (filter)	0.757	(0.282)
7	Manhattan (filter)	0.136	(0.083)	7	Outlier Count (*z*-score)	0.782	(0.306)
8	Manhattan	0.153	(0.092)	8	Manhattan	0.843	(0.242)
9	Neural Network (auto-assoc)	0.161	(0.080)	9	Neural Network (auto-assoc)	0.859	(0.220)
10	Euclidean	0.171	(0.095)	10	Euclidean	0.875	(0.200)
11	Euclidean (normed)	0.215	(0.119)	11	Euclidean (normed)	0.911	(0.148)
12	Fuzzy Logic	0.221	(0.105)	12	Fuzzy Logic	0.935	(0.108)
13	*k* Means	0.372	(0.139)	13	*k* Means	0.989	(0.040)
14	Neural Network (standard)	0.828	(0.148)	14	Neural Network (standard)	1.000	(0.000)

For Masada, it was determined that the Mahalanobis distance would be most appropriate. It is relatively simple to calculate (referring back to the criteria that a developer would be implementing this without previous experience in this field), but is used as a reliable measurement for anomaly detection [10]. The equal-error rate differs only by 0.014 from the best detector. Additionally, its zero-miss false-alarm rate is ranked second among all the detectors. The zero-miss false-alarm rate is where the false positive rate is zero (use of this threshold will result in a higher false negative rate). It should be noted that the Nearest Neighbor (Mahalanobis) distance is a variation of the Mahalanobis, really only differing in the number of sample typing profiles compared to. Given the scope of this project, specifically the limited base profiles that could be created, the Mahalanobis distance was chosen. This distance is calculated as:

$$D = \sqrt{(x-y)^T S^{-1} (x-y)}$$

Equation 1 Mahalanobis distance formula

In equation 1, 'x' is the base profile vector, 'y' the collected data vector, and S the covariance matrix (the inverse is taken). The final value returned by the calculation is essentially the standard deviation of the two profiles as compared to each other. As the value becomes greater than 0, the less likely two profiles are the same.

A distance for both press and fly times is calculated, and then a final, weighted distance, is calculated from the two distances. From analysis of the base profiles, it was found that press times appeared to be more consistent and had smaller standard deviations than fly times. Therefore, it felt appropriate to weight the distance slightly, so the final distance is determined as:

Final Distance = 0.6 * PressTimes + 0.4 * FlyTimes

Equation 2 Keystroke weighting

Once the final distance (see Equation 2) is calculated, some upper threshold must be determined. A range of 0 - 0.5 was determined to be a valid range for a user to not be rejected [3]. A rate over 0.5 and a user would be rejected.

Masada was designed to reliably detect a user based on their typing profile, and to do so in such a way that would be rigorous, but also feasible (from a developer's point of view). We feel that both these requirements were met. Please see section *4.2.2 Masada Results*.

3.6 KeyTrac

In addition to Masada, which implements the Mahalanobis distance, we decided that using Ditto against commercial software would be a realistic test of our implementation. Many of the commercial offerings do not allow for free trial periods, so it was a challenge to find. Wikipedia provided a list of companies, and one such company, KeyTrac, had a free online demo of their AnyText software.

KeyTrac is an IT security solution from TM3 Software GmbH, based in Regensburg, Germany. The company was founded in 2008 as a spin-off of the University of Regensburg. KeyTrac is unique in that it offers analysis on keystroke data that is not pre-known. This means that a user can create a profile using a certain text, and then be verified typing a completely different text. Many of the other commercial solutions required that the profile only compare against the same text, limiting the flexibility of the program.

KeyTrac's demo is browser based application. A user creates a reference profile, and that profile is saved and accessed via an email address. To test against the profile, the user logins in with an email and types in the collection box. After typing 290 characters, the user presses a button and the software determines if the user typing created the reference profile. A percent confidence is given, 100% being a total match; we found that KeyTrac rejects users under 90%.

KeyTrac advertises that its software can do the following:

- Preventing online fraud
- Recognizing double registration
- Intrusion detection
- Resetting forgotten passwords
- **Preventing subscription - /account-sharing**
- **Protect software licensing**
- E-learning: Online exams
- Author verification
- Email and document signature
- Credit agencies

The two bolded bullet points highlight our attack against keystroke dynamics software; our research specifically concerns these areas.

4. Results

To check Ditto's mimicking ability we tested against commercial software, KeyTrac and our own implementation of the Mahalanobis distance in a Windows application called Masada. KeyTrac is closed source proprietary software and we do not have knowledge of its keystroke detector implementation.

4.1 KeyTrac

KeyTrac [8] is a German software company providing software solutions for keystroke dynamics. They provide products for various specific use cases such as integration with outlook. A more general product called AnyText has an online demo which allows users to test firsthand the effectiveness of their product. Since this demo is free of charge, easily accessible via the web and closer to the use case of Masada we chose it to test Ditto.

4.1.1 Experimental Approach

KeyTrac allows users to submit a typing sample and create a reference profile on their website. The first reference profile we made was of a user typing as himself without Ditto and we used this to verify KeyTrac's ability to identify the correct user and reject others.

The second set of experiments involved creating reference profiles on KeyTrac using Ditto profiles for the various partitions 0-5. Using these reference profiles we tested KeyTrac's identification with users not typing under Ditto and then typing under the Ditto profile corresponding to the reference profile in KeyTrac.

The last test is intended to explore KeyTrac's sensitivity to variations in the typed text in correctly identifying users. When creating the reference profile, KeyTrac has you type a specified sample text. While you are not required to type that exact text in, that is how we generated the reference profiles. The amount of text typed is only 290 characters, meaning KeyTrac collects on average around that many fly times when creating a reference profile. This means that KeyTrac must infer more than 50% of your actual keystrokes from the ones they have collected. To test how well they do this, we tried to match the reference profile while typing progressively dissimilar pieces of text. The other pieces of text were created by modifying the base KeyTrac text and using the Levenshtein distance to give a rough similarity measurement.

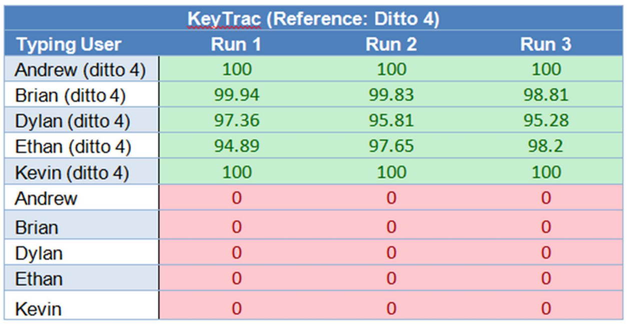
4.1.2 Results

When discussing KeyTrac results it should be noted that the AnyText online demo identifies or *accepts* a user under a reference profile when it has a confidence above 90%. Below this threshold KeyTrac *rejects* the user and no percentage is displayed so it will be marked as 0%.

The first test of KeyTrac was users typing as themselves without Ditto trying to match a profile made by another user. The results of this where as expected with KeyTrac identifying the original user of the reference profile with over 99% confidence and every other user was rejected over all 3 trials.

The next set of tests involved users trying to match a reference profile created under a Ditto profile for one of the partitions. These tests were conducted using the members of this group, all of whom were placed under partition 4. Table 2 shows the results from KeyTrac with users typing under the Ditto profile for the partition and without Ditto.

Table 2 KeyTrac verification with Ditto 4 reference

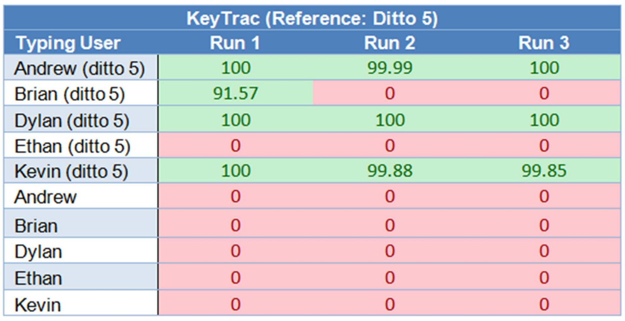


KeyTrac (Reference: Ditto 4)			
Typing User	Run 1	Run 2	Run 3
Andrew (ditto 4)	100	100	100
Brian (ditto 4)	99.94	99.83	98.81
Dylan (ditto 4)	97.36	95.81	95.28
Ethan (ditto 4)	94.89	97.65	98.2
Kevin (ditto 4)	100	100	100
Andrew	0	0	0
Brian	0	0	0
Dylan	0	0	0
Ethan	0	0	0
Kevin	0	0	0

As expected, Ditto was able to mimic all users under the partition profile they were originally assigned. Some people saw lower confidence scores and this is likely a result of typing slower than Ditto in certain circumstances. If the user types too slow for a profile Ditto cannot make up keystrokes to keep up the timing so you inevitably fall out of the profile. As mentioned earlier, we choose the slowest profile in a partition to represent the whole partition and as you can see here even the lower scores have a reasonable margin before the 90% rejection threshold is reached.

The next step after testing users in their own partition under Ditto is to test them in another partition than the one they were assigned. For this we tested the same users in all other partitions, 0-3 and 5. The results for 0-3 where similar to the ones for Ditto 4 with the exception of one hiccup which we attributed to poor timing in Ditto as a result of scheduling in a virtual machine. Table 3 presents the result from KeyTrac with users trying to match the Ditto 5 reference profile.

Table 3 KeyTrac verification with Ditto 5 reference



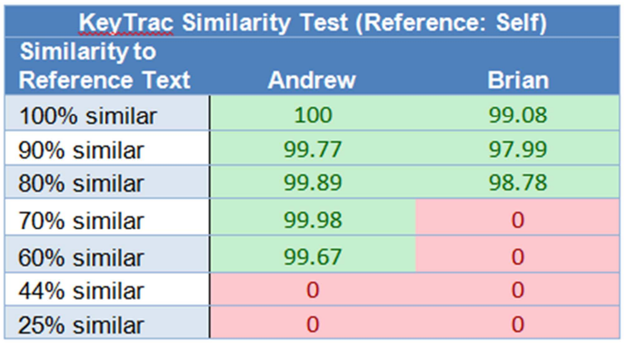
KeyTrac (Reference: Ditto 5)			
Typing User	Run 1	Run 2	Run 3
Andrew (ditto 5)	100	99.99	100
Brian (ditto 5)	91.57	0	0
Dylan (ditto 5)	100	100	100
Ethan (ditto 5)	0	0	0
Kevin (ditto 5)	100	99.88	99.85
Andrew	0	0	0
Brian	0	0	0
Dylan	0	0	0
Ethan	0	0	0
Kevin	0	0	0

Results here were better than expected since none of the users typing were partitioned into the faster Ditto 5 group. We don't believe this means our partitioning was wrong and these results are because of a difference in how the profiles where collected. Our profiles were collected using an excerpt of Don Quixote that is relatively terse and tougher to type than the sample text on KeyTrac. This means that our collected profiles are on average slower across the board than if they had been collected using easier text.

Despite the difference in text used to collect profiles it is apparent that the partitioning wasn't wrong, as we see one third of the trials were rejected. This is what we would have expected when users move to a partition above their own. While these users were not able to type fast enough for the profile they could have potentially matched by typing very fast but in gibberish. Ditto will match anything you type to the profile so you can achieve a higher wpm count by throwing away accuracy. KeyTrac does not know what you are trying to type (their text is only a suggestion) so accuracy is not accounted for in their confidence. Even so, the gibberish you are typing will likely result in fly time combinations that were unlikely encountered during profile creation so you will be comparing KeyTrac's guess for that time to our guess which leads to the next test on KeyTrac's sensitivity to the text being typed.

Without resorting to large set of text or a cryptic passage, KeyTrac or any other keystroke profile collector cannot gather all keystroke combinations in a reasonable amount of text. KeyTrac requires 290 characters for their sample which gives them below 45% of the potential 676 fly pairs for alphabet character and below 25% of the nearly 1300 fly pairs for alpha numeric. These percentages are assuming that every fly pair in KeyTrac's text is unique which they are not. This means that in matching profiles they must infer other keystroke times. Their implementation is proprietary but we can do some simple tests to determine their accuracy as you stray from the collected sample.

Table 4 KeyTrac similarity test. Reference profile self

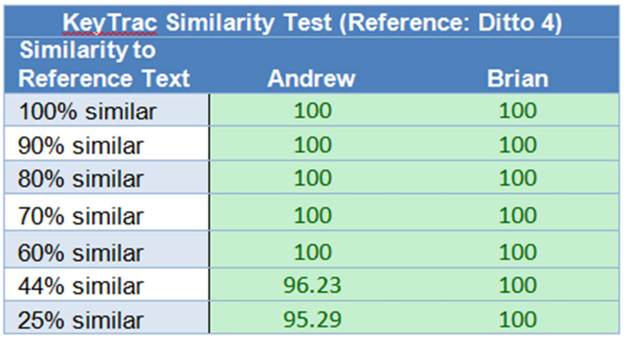


KeyTrac Similarity Test (Reference: Self)		
Similarity to Reference Text	Andrew	Brian
100% similar	100	99.08
90% similar	99.77	97.99
80% similar	99.89	98.78
70% similar	99.98	0
60% similar	99.67	0
44% similar	0	0
25% similar	0	0

Table 4 shows the KeyTrac confidence when matching a user typing against their reference profile. For both users we see that below 50%, KeyTrac's confidence drops below the threshold for acceptance while staying fairly consistent up to the tested 60%.

The text used in this similarity test was maintained for readability to avoid slower or different times resulting from cryptic passages. Even so we see a decrease in the effectiveness of KeyTrac as we explore key strokes that weren't originally captured. We took this further and tested the sensitivity to text similarity while typing under the Ditto 4 partition profile which is the partition we were placed under.

Table 5 KeyTrac similarity test. Reference profile Ditto 4



KeyTrac Similarity Test (Reference: Ditto 4)		
Similarity to Reference Text	Andrew	Brian
100% similar	100	100
90% similar	100	100
80% similar	100	100
70% similar	100	100
60% similar	100	100
44% similar	96.23	100
25% similar	95.29	100

Table 5 shows that using the same text from before Ditto allowed us to be accepted even when the similarity of the text was very low. This may result from our filled in fly and press times being closer to KeyTrac's estimated times than our own actual typing speeds were. KeyTrac could improve this by collecting larger data samples to avoid guessing. While this would help improve the acceptance rate in the case of Table 4 it would not change the results of Table 5 since the times in the reference profile would be gathered directly from the guess times in the profile and taken as the actual time for the user.

Overall we were satisfied with the results we got from testing Ditto against the KeyTrac software. Users typing under the partition they were placed under were accepted every time while typing under Ditto and not at all while not under Ditto. We also saw that Ditto can help improve acceptance rates as text varies because it has filled in times for key presses that otherwise are not collected during profiling.

4.2 Masada

In order to check that Masada was doing its job of accurately verifying users, it was tested both with and without Ditto.

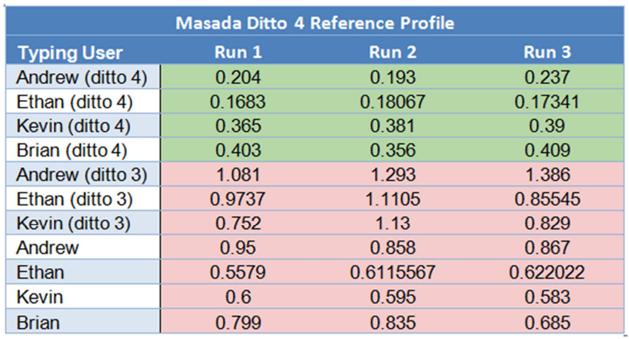
4.2.1 Experimental Approach

Masada was tested three different ways. The first was to see if Masada would accept a user typing without Ditto. The second was to run Ditto with a profile slightly slower than the testing profile, so as to verify that Masada had relatively strong checking. The third test was to run Ditto with the same profile as Masada, to see if Masada would accept the user.

It should be noted that the four people testing Masada all ended up in the same partition, partition 4.

4.2.2 Results

Table 6 Masada reference profile: Ditto 4



Masada Ditto 4 Reference Profile			
Typing User	Run 1	Run 2	Run 3
Andrew (ditto 4)	0.204	0.193	0.237
Ethan (ditto 4)	0.1683	0.18067	0.17341
Kevin (ditto 4)	0.365	0.381	0.39
Brian (ditto 4)	0.403	0.356	0.409
Andrew (ditto 3)	1.081	1.293	1.386
Ethan (ditto 3)	0.9737	1.1105	0.85545
Kevin (ditto 3)	0.752	1.13	0.829
Andrew	0.95	0.858	0.867
Ethan	0.5579	0.6115567	0.622022
Kevin	0.6	0.595	0.583
Brian	0.799	0.835	0.685

Users appended with *(ditto n)* are typing with Ditto under the given partition profile number specified. Users without anything after their name are typing as themselves.

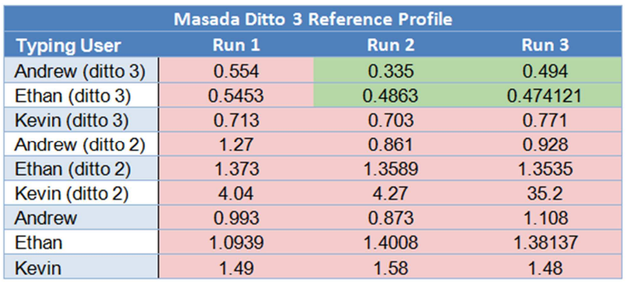
Table 6 shows that Masada was able to correctly reject users typing without Ditto, even when the users where in the same partition as the profile Masada was using. The distances for each trial are, for the most part, well above 0.5, the upper threshold for acceptance. Masada is therefore able to detect that a user does not match a profile, and, more importantly, have the distance far enough away that an accidental match is not likely (the closest rejection was 0.083 away from the upper threshold). In fact, we expected these numbers to be closer to 0.5, and potentially below 0.5, because all the testers were in the same partition.

During the second test, Masada was run with a reference partition profile. Ditto was also run, but it was one profile under what Masada was checking against. For example, if Masada was checking against profile 4, Ditto would be running profile 3.

From Table 6, it can be seen that Masada emphatically rejects the slower profile (here, profile 3). Once again, we expected there to be some cases where Masada accepted a user typing one profile below, however this was not the case. This test showed that Masada could determine that a profile that was not close to the testing profile would not be matched.

Finally, the third test was running Ditto and Masada at the same time with the same profiles. Table 6 shows the test using profile 4. Every single time, the user was verified under Masada, meaning that Ditto was able to successfully fool Masada into thinking the same person was typing. Interestingly, the distances were well below the upper threshold, meaning Masada was fairly convinced that the user typing was the original user who created the reference partition profile.

Table 7 Masada reference profile: Ditto 3



Masada Ditto 3 Reference Profile			
Typing User	Run 1	Run 2	Run 3
Andrew (ditto 3)	0.554	0.335	0.494
Ethan (ditto 3)	0.5453	0.4863	0.474121
Kevin (ditto 3)	0.713	0.703	0.771
Andrew (ditto 2)	1.27	0.861	0.928
Ethan (ditto 2)	1.373	1.3589	1.3535
Kevin (ditto 2)	4.04	4.27	35.2
Andrew	0.993	0.873	1.108
Ethan	1.0939	1.4008	1.38137
Kevin	1.49	1.58	1.48

Table 7 shows the results from running profile 3. Once again, the same three tests were run. For the first two tests, very similar results were observed; the users were regularly rejected (100% of the time). The third test, however, showed some discrepancies from what was predicted.

For the nine tests run where a user typed under Ditto profile 3 while being checked in Masada against profile 3, there were only four that were accepted. Originally, we had assumed that it would be an almost 100% acceptance rate, but this was not the case. In the cases that Masada accepted the profile, three of the four distances were very close to 0.5, indicating that Masada had less confidence about the data. We are unsure why this happened, but there are a few explanations. First, the third tester (Kevin) was using an unplugged laptop. If the laptop had gone into a power save mode, Ditto could have experienced timing fluctuations due to cpu scheduling, reducing the accuracy of the necessary thread sleeps. This would throw off the times, likely increasing the distance. Ditto may also need some fine tuning when a user types faster than the profile it is mimicking. As key presses and fly times are being stored before being released, random fluctuations in thread sleep times could throw off timings. Given the success seen in Table 6, it is likely a technical timing issue that needs to be resolved. Additionally, the tests using Ditto profile 3 on the KeyTrac website were successful, so this situation might have been anomalous.

5. Future Work

Our results are promising, but to be used in a scenario such as our motivating example there would be significantly more work in the security and usability of Ditto to the statistical models used in analyzing profiles.

5.1 Improving Ditto

Ditto is implemented using the C++ interception library which installs itself as a kernel level module and allows anyone to access it keystroke capturing context. This is a serious security vulnerability for anyone such as a company trying to use Ditto. A better approach would be to have a kernel mode driver like interception with an authenticated connection mechanism or implement Ditto at the same level as interception.

The success of Ditto's mimicking is highly dependent on the accuracy and precision of its timers. We have found that this timing can be thrown off to the point of being user visible because of power saving cpu scheduling and cpu scheduling noise as a result of a virtual machine. We have attempted to alleviate this issue by only using AC connected laptops/computers, setting Ditto to high scheduling priority and avoiding the use of virtual machines.

5.2 Better reduction and partitioning

From a usability aspect, if a pipeline from profile collection to Ditto mimicking was to be commercialized a GUI would be immensely useful. Outside of commercial use, a user interface would also be beneficial for viewing statistics of individual profiles and groups of profiles/partitions as a distribution.

The reducer could use a more sophisticated statistical analysis of profile times. The current approach of means and standard deviations works well assuming the distribution of times is normal. As mentioned earlier, a more sophisticated approach to filling in times would help in creating a more accurate profile times while not requiring excessive amounts of data collection.

While taking a more sophisticated approach to filling in times, a larger data collection sample would help improve profile accuracy. Collecting data over several days would help reduce the effect of daily variations in timings.

Partitioning right now is a fairly static approach. The goal in partitioning should be to minimize the number of partitions while maintaining usability by not having users too far away from their partition's profile. Scaled partitioning does allow for more fine grained grouping but you still end up with many empty groups. A more dynamic approach that uses the distribution of collected profiles along with research in typing usability could help improve this phase of the pipeline.

6. Conclusion

Keystroke dynamics has progressed quickly in its short history. While using keystroke dynamics as a secondary security method has breathed new life into the field, advances in hardware also helped. Timer accuracy greatly increased, meaning that times down to the millisecond level could be reliably observed. Massive calculations comparing profiles could also be done faster, meaning that a greater sample of keystroke data could be compared, yielding better acceptance and rejection rates.

Today, keystroke dynamics are used in many websites that require one user per license subscriptions. Developers and companies have found this to be an effective way to enforce their policies. Many different solutions have been found; either in-house keystroke dynamic software is custom written, or off the shelf products are purchased.

This leads us to our research, namely, could we fool a keystroke dynamics program into thinking that only one person was typing under a license, when in fact a whole group of people were? In order to answer this question, we set out to build a program to mimic a user, and we called it Ditto. Ditto was built to capture a user's keystrokes, and then release them based on a stored profile. Once Ditto was framed up, we need to create the profiles Ditto would use. This led us to building a website to collect keystroke profiles. From this, we created a profile of that volunteer's data that Ditto would use.

Once Ditto had profiles to mimic, we needed a way to test the program. To do this, we took two approaches. The first was to implement a keystroke dynamics detector (commonly called finding the distance between two profiles). We used a paper that aggregated [10] various detectors and analyzed their effectiveness. We chose a method and implemented it. The testing program that we wrote was called Masada. It would collect a user's typing data, and then compare that to a supplied profile. A decision would be made, using the Mahalanobis distance, on whether or not the user matched the comparison profile. The second test was to find a commercial keystroke dynamics product and see if we could fool that. Both tests would be used to determine how well a keystroke detector could be fooled, from both a purely research side, as well as commercial.

Our results were rather surprising. When we set out on this project, we figured we would be able to mimic a user successfully under the first test (Masada). This was because a basic detection method was being used (while still strong, it did not factor in things like time of day, how users spell certain words, etc.). For the KeyTrac test, we did not think we would be able to mimic a user. This was interesting because KeyTrac is a commercial product that has had years of development (Masada would be a "child" compared next to the more mature KeyTrac program). However, we found that KeyTrac would accept us almost 100% of the time. There were four people testing, and Ditto was able to convince KeyTrac that all four testers were one person. This was very fun to see, as this result was unexpected, but pleasing.

Masada was also a success, but in a different way. We were able to fool the detector into thinking that only one person was typing 100% of the time, but that was limited to the partition the testers were in. When we tested Masada with a partition the testers were not in, the acceptance rate dropped (Masada could tell we were trying to fool it). We came up with two different reasons for this. The first was that Ditto was having a technical problem with sometimes letting out keys when a faster user typed under a slower profile. Perhaps the sleep times on the threads were not as accurate as hoped and so key presses and fly times were released incorrectly. If this is the case, it is just a matter of making Ditto more robust. The second reason why Masada was stronger at detecting a mimicking user was that the more simple distance calculation was better. Masada only looked at fly and press times. There was no meta-analysis done on what was being typed, so Masada could be fooled in this way.

In the end, the goal of our research was to determine if we could partition users into groups and then mimic a group of users with one profile. From our results, we can conclude that it is indeed feasible to do this.

7. REFERENCES

[1] Epp, C., Lippold, M., Mandryk, R. L."Identifying emotional states using keystroke dynamics." In *Proceedings of the 2011 annual conference on Human factors in computing systems*. ACM, 2011. DOI= http://doi.acm.org/10.1145/1978942.1979046 .

[2] Bergadano, F.. Gunetti, D. and Picardi, C. (2002) User authentication through keystroke dynamics. *ACM Transactions on Information and System Security*, vol. 5, no. 4, 367-397

[3] Garcia, J. D. Personal identification apparatus, November 4 1986. US Patent 4,621,334.

[4] Harrington, N. 2007. Expand your text entry options with keystroke dynamics. (Dec 2007). Retreived Dec 10, 2012 from http://www.ibm.com/developerworks/opensource/library/os-keystroke/

[5] Harrington, N. 2008. Identify and verify users based on how they type. (Mar 2008). Retreived Dec 10, 2012 from http://www.ibm.com/developerworks/opensource/library/os-identify/

[6] Harrington, N. 2008. Create a continuous keystroke-dynamics monitor with Perl and xev. (Oct 2008). Retreived Dec 10, 2012 from http://www.ibm.com/developerworks/opensource/library/os-keystroke-perl-xev/index.html

[7] Jiang, C. H., Shieh, S., Liu, J. C. 2007. Keystroke statistical learning model for web authentication. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 359-361. ACM.

[8] KeyTrac. AnyText. Retreived Oct 5, 2012 from http://www.keytrac.de/

[9] Killourhy, K. S. and Maxion, R. A. 2008. The effect of clock resolution on keystroke dynamics. In *Recent Advances in Intrusion Detection* 331-350. Springer Berlin/Heidelberg.

[10] Killourhy, K. S. and Maxion, R. A 2009. Comparing Anomaly-Detection Algorithms for Keystroke Dynamics. *Dependable Systems & Networks*, 2009. DSN'09. IEEE/IFIP International Conference on. IEEE, 2009.

[11] Monrose, F. and Rubin, A. D. 2000. Keystroke dynamics as a biometric for authentication. *Future Generation Computer Systems*, 16(4), 351-359.

[12] Resig, J. 2008 Accuracy of JavaScript Time. (Nov 2008). Retreived Dec 10, 2012 from http://ejohn.org/blog/accuracy-of-javascript-time/

[13] Sridhar, M., Abraham, T., Rebello, J., D'souza, W., D'Souza, A. 2012. Intrusion Detection Using Keystroke Dynamics. In *Proceedings of the Third International Conference on Trends in Information, Telecommunication and Computing* (2012, July). 137-144. Springer,NY.

[14] Stross, R. 2012. Bypassing the Password. (March 2012). Retrieved Dec 12, 2012 from http://www.nytimes.com/2012/03/18/business/seeking-ways-to-make-computer-passwords-unnecessary.html

[15] Yorick. 2010. Interception. Retreived Oct 5, 2012 from http://oblita.com/Interception.html

[16] Zhong, Yu, Y. Deng, and A. K. Jain. 2012. Keystroke dynamics for user authentication. *Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2012 IEEE Computer Society Conference on. IEEE, 2012.