Winter 2013

Snoop-forge-replay attack on continuous verification with keystrokes

Khandaker Abir Rahman

SNOOP-FORGE-REPLAY ATTACK ON CONTINUOUS VERIFICATION WITH KEYSTROKES

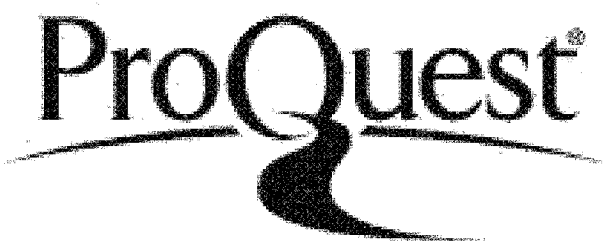
by

Khandaker Abir Rahman, B.Sc., M.S., M.S., M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

March 2013



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

November 15, 2012

Date

We hereby recommend that the dissertation prepared under our supervision by Khandaker Abir Rahman

entitled

Snoop-forge-replay Attack on Continuous Verification with Keystrokes

be accepted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Supervisor of Dissertation Research

Head of Department

Computational Analysis and Modeling

Department

Recommendation concurred in:

Advisory Committee

Approved:

Director of Graduate Studies

Approved:

Dean of the Graduate School

Dean of the College

GS Form 13a
(6/07)

ABSTRACT

We present a new attack called the snoop-forge-replay attack on the keystroke-based continuous verification systems. We performed the attacks on two levels – 1) feature-level and 2) sample-level.

(1) Feature-level attack targets specific keystroke-based continuous verification method or system. In feature-level attacks, we performed a series of experiments using keystroke data from 50 users who typed approximately 1200 to 2300 keystrokes of free text during three different periods. The experiments consisted of two parts. In the first part, we conducted zero-effort verification experiments with two verifiers ("R" and "S") and obtained Equal Error Rates (EERs) between 10% and 15% under various verifier configurations. In the second part, we replayed 10,000 forged impostor attempts per user and demonstrated how the zero-effort impostor pass rates became meaningless when impostor attempts were created using stolen keystroke timing information.

(2) Sample-level attack is not specific to any particular keystroke-based continuous verification method or system. It can be launched with easily available keyloggers and application programming interfaces (APIs) for keystroke synthesis. Our results from 2640 experiments show that (i) the snoop-forge-replay attacks achieve *alarmingly high* error rates compared to zero-effort impostor attacks, which have been the *de facto* standard for evaluating keystroke-based continuous verification systems; (ii) four state-of-the-art verification methods, three types of keystroke latencies, and eleven

matching-pair settings (–a key parameter in continuous verification with keystrokes) that we examined in this dissertation were *susceptible* to the attack; (iii) the attack is effective even when as low as 20 to 100 keystrokes were snooped to create forgeries.

In light of our results, we question the security offered by the current keystroke-based continuous verification systems. Additionally, in our experiments, we harnessed virtualization technology to generate thousands of keystroke forgeries within a short time span. We point out that virtualization setup such as the one used in our experiments can also be exploited by an attacker to scale and speed up the attack.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author Wendolen Abrahman

Date 02/14/13

DEDICATION

To my parents and wife for their never-ending support and for giving me the ability to see this through to the end.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In login time verification, the identity of the user is verified "once" before granting access to the computer. A drawback with login time verification is that an unauthorized user can gain access to the computer by replacing a legitimate user who is logged in, either through coercion (*i.e.*, forcefully replacing the user) or when the logged-in user leaves the computer without logging out. This vulnerability of login time verification is a serious security risk because, after gaining access, the unauthorized user can perform a broad range of malicious activities including installing malware, spreading viruses, and (or) exfiltrating/destroying sensitive data. To deter this kind of unauthorized access, several studies (*e.g.*, [1 – 5] proposed biometric based methods to *continuously verify* the identity of a logged-in user. A subset of these studies used cyber-behavioral traits (*e.g.*, keystroke dynamics [1 – 4], [6 – 11]; mouse dynamics [12 – 14]; and web usage patterns [15]) to continuously verify users. For continuous verification, cyber-behavioral traits are appealing because they, 1) are *non-intrusive*–they emerge naturally from a user's interaction with the computer and user intervention is not required when collecting them; 2) provide *broader coverage*–they can be collected on almost all desktops, laptops, and mobile devices without requiring any special hardware (*e.g.*, fingerprint readers, cameras, or biometric scanners); and 3)

are *available* even when the user is physically away from the computer and is accessing it remotely.

Among the cyber-behavioral traits, majority of the studies used keystroke patterns for continuous verification. Two factors motivate the use of keystroke patterns for continuous verification: 1) typing is one of the most common activities a user performs on the computer and therefore, one could expect a reasonable supply of keystrokes for performing continuous verification and 2) studies (*e.g.*, [16, 17]) demonstrated that an individual's typing behavior can be used as a unique "signature" to identify the individual.

Almost all the studies in continuous user verification with keystrokes have focused on developing methods to improve verification performance. The focus of this research is different. We present a new attack called the "snoop-forge-replay" attack on continuous user verification with keystrokes. The attack is executed in three steps: 1) *snoop* (steal) a victim user's keystroke timing information using a keylogger, 2) *forge* a typing sample using the keystroke timing information stolen from the victim user, and 3) *replay* the forged typing sample in such a way that the continuous verification system thinks that it is the victim user who is typing. The *goal* of the attack is to submit forged typing samples to the verifier so that an attacker can access the computer without being detected. Salient features of the attack follow.

Effective: through a series of experiments conducted using keystroke data from 350 users (150 genuine and 200 impostors), four state-of-the-art continuous verification methods, and templates built with three types of keystroke latencies, we show that the snoop-forge-replay attacks have *alarmingly high* error rates compared

to the error rates of zero-effort[1] impostor attacks typically used to evaluate keystroke-based continuous verification systems.

Few words become deadly: the attack is surprisingly effective even when a small amount of snooped latencies are used to build forgeries. With 20 characters (few words of text) to 100 characters (less than two lines of texts typed in a typical email textbox) of snooped information, we achieved high error rates against state-of-the-art verification systems. (See Figure 4.7, Page 53 for the error rates of snoop-forge-replay attacks launched with short snooped text).

Legacy keystroke samples remain a threat: because the snoop-forge-replay attack uses forgeries built with stolen latencies of a user, the high attack success rates can seem to be obvious and expected. However, we snoop the legacy keystrokes, which are keystrokes of a user captured approximately *six months* before collecting his/her training (enrollment) samples. Given that behavioral traits such as keystroke latencies have high intra-user variabilities and can change over time, it is interesting to note that our attack achieves high success rates when forgeries are created using legacy keystrokes.

Speed and scalability: by using short stolen samples, the attack can be launched quickly as the attacker does not have to wait long to collect victims' keystrokes. By exploiting virtualization, we show that thousands of attacks can be launched to *simultaneously* attack hundreds of users in a short time span. Using a virtualization

[1] In a zero-effort impostor attack, the "natural" typing patterns of one user are used as impostor attempts against another and the impostor does not deliberately try to mimic a victim user.

setup, we created on average 5594.98 to 299.38 attacks per user in 24 hours. In Table 4.7 on Page 48, we give the average number of attacks *per user*.

Three factors make the attack feasible: 1) Many hardware and software keyloggers that can steal an individual's keystroke timings are openly available on the Internet for different platforms (*e.g.*, MS Windows, GNU/Linux). 2) It is possible to develop a "keystroke emulator" to replay forgeries. A keystroke emulator is a software program that generates synthetic key press and release events using APIs like `SendInput` [18] for MS Windows or programs like `xsendkeycode` [19] for X Windows system. 3) In sample-level attack, the attacker deceives a verification system by presenting fake key press and release events to the keystroke sensor. To launch the attack, the attacker does not have to know the internal specifications of the continuous verification system, such as what verification algorithm is being used, verifier's parameter settings, how the templates are constructed, and latencies being used, all of which can be proprietary information.

Contributions of the dissertation are as follows:

In feature-level attack, we performed a series of experiments on keystroke data collected from 50 users. We show that forgeries created from snooped keystroke information have alarmingly high impostor pass rates. Our results show that at verifier configurations yielding less than 0.1 zero-effort impostor pass rates (at ≤ 0.15 false reject rate), the success rates of forgery attempts is between 75% and 88%

In sample-level attack, we conducted 2640 attack experiments with 24 attack configurations, 10 individual and fusion verifier configurations, 11 matching-pair

settings ($24 \times 10 \times 11 = 2640$) and achieved as high as 125.5 to 2915.62 percentage increase in error rates compared to the error rates with baseline zero-effort impostor attacks (see Table 4.9. Page 55 and the discussion in Section 4.2.3, Page 48). Our results reveal that there is a *wide* disparity in the error rates achieved with the zero-effort impostor attacks and the error rates achieved with snoop-forge-replay attacks (see plots in Figure 4.4, Page 50).

Implication: The high error rates with snoop-forge-replay attacks raise two fundamental questions: 1) *is it secure to use keystrokes to continuously authenticate computer users?* and 2) *how can we redesign keystroke-based continuous authentication systems that are resilient to forgery attacks?*

We analyzed the effect of four attack parameters, i) number of snooped keystrokes–we experimented with 20, 50, 100, 200, 600, and 1200 snooped keystrokes; ii) filtering outliers in the snooped keystrokes–we experimented with and without filtering outliers; iii) Gaussian perturbation of snooped latencies–we experimented with and without perturbing latencies, and iv) frequency of occurrence of digraphs in snooped text–we experimented with 1, 2, and 3 occurrences of digraphs.

Findings: Snooping more keystrokes from a victim user does not necessarily result in better attacks. In fact, our results with two verifiers ("S" and "A") showed that snooping more keystrokes decreased the pass rates of the attacks. We analyzed (in Section 4.2.6, Page 56) why snooping more keystrokes may have adversely effected the attack performance. Our results also showed that filtering outliers in the snooped keystrokes and considering digraphs that have occurred at least twice improved the

pass rates of the attack. Gaussian perturbation made the attack weak against "S" and "A" verifiers and had the least effect on "R" and "F" verifiers.

To generate a sufficient number of snoop-forge-replay attacks for evaluation, we emulated the typing activity of a victim user for 24 hours (*i.e.*, we executed a keystroke emulation program for 24 hours to generate a sufficient number of forgeries for each victim). Because we experimented with 150 victim users and 24 attack configurations, we would have to run the emulator for 150 (victims) × 24 (attack configurations) × 24 hours = 3600 days (or approximately 10 years). To perform emulation at this scale, we set up a virtualization environment with 150 virtual machines. We dedicated one virtual machine for emulating a victim. For each attack configuration, we ran 150 emulators *parallely* on 150 virtual machines and reduced the emulation time to just 24 days. See Section 3.4, Page 29 for details on the virtualization environment.

An attacker can exploit virtualization: By parallely running 150 virtual machines, in 24 hours, we forged thousands of attacks against 150 users (see Table 4.7, Page 48). The attacker, by exploiting virtualization, can further reduce the time to forge the same number of attacks, say from 6 to 24 hours, by quadrupling the number of virtual machines. By increasing the number of virtual machines, the attacker can also generate a huge number of forgeries (*e.g.*, in the order of millions) or scale the attack to victimize thousands of users.

We collected keystroke data from 150 users, who gave their typing samples in three phases, over a period of one year. To our knowledge, this is the longest time span data used in continuous keystroke verification research. Using this data, we demonstrate that it is possible to achieve high attack success rates with keystrokes

samples stolen *six months* before the training/enrollment samples. Thus, our work indicates that old stolen keystroke samples remain a threat and an attacker can potentially exploit stolen keystrokes to launch forgery attacks over a prolonged period of time.

CHAPTER 2

BACKGROUND

2.1 Continuous Verification with Keystrokes

In Figure 2.1, we illustrate continuous user verification with keystrokes. In the training phase, keystroke latencies are extracted from the enrollment text and processed, users' keystroke templates (profiles) are created, and a verifier (matching algorithm) is configured. In the verification phase, keystroke latencies are extracted from the verification text. A verifier matches the latencies against the user's template to generate a match score. In continuous verification, extracting latencies from verification text and matching them against the user's template is a continuous process. Details follow.

Keystroke Latencies: Widely used latencies in the literature are: 1) *key hold* latency–is the time between press and release of the same key, 2) *key press* latency–is the time between press of a key and press of the next key, and 3) *key interval* latency–is the time between the release of a key and press of the next key. We experimented with key hold, key interval, and key press latencies.

Template: A template stores the keystroke signatures of a user. We used a 26-by-26 matrix as the template. There are 676 cells in the template. Each cell corresponds to an English alphabet pair: `aa`, `ab`, `ac`, $\cdots$, `zy`, `zz`. In our experiments,

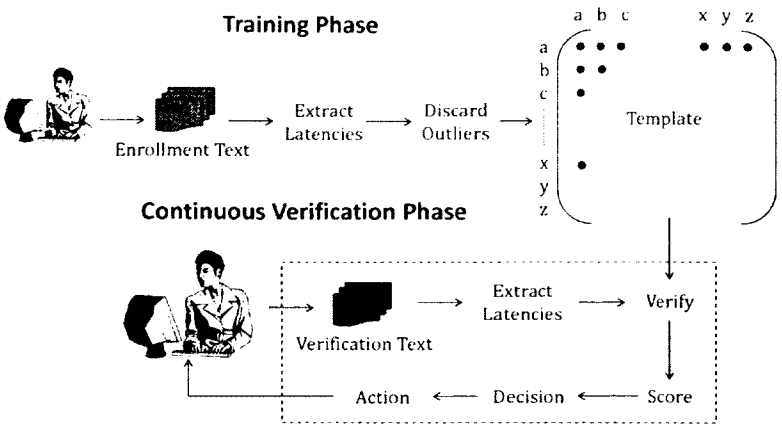


Figure 2.1: Overview of continuous verification with keystrokes. In the training phase, keystroke latencies are extracted and outliers are removed. Each "•" in the template represents a cell entry containing the latency values (and their mean and standard deviation) of an English digraph. In the verification phase, latencies are extracted as the user types the verification text. After obtaining M matching pairs, the verifier matches the latencies with the template and outputs a score. A verification decision is computed by thresholding the score. Based on the decision, the user is either allowed to continue or an action is taken. The perforated box shows the continuous verification loop.

when we used key press (or interval) latencies, each cell in the template stored key press (or interval) latencies of a letter pair. For example, with key press latencies, if cell "ab" has $\{(110, 90, 100), \mu_{\mathtt{ab}} = 100, \sigma_{\mathtt{ab}} = 10\}$, it means that the user (during enrolment) typed ab thrice with 110ms, 90ms, and 100ms delay between the press of a and the press of b and the mean delay is 100ms with 10ms standard deviation. Similarly, if it were key interval latencies, then 110ms, 90ms, and 100ms would be the delays between the release of a and the press of b. Unlike key press and interval latencies, a key hold latency by definition is associated with a letter (and not letter pair). Because our template holds only letter pairs, when we used key hold latencies, each cell stored the key hold latencies of the first letter of its letter pair (*e.g.*, cell "ab" stored key hold latencies of a only when the next letter typed is b). Our template

is homogeneous, meaning it stores only one type of latencies (*i.e.*, either key hold, interval, or press), because the continuous keystroke verifiers used in the dissertation are not designed to operate with a template containing multiple types of latencies (*e.g.*, a mixture of both key hold and interval). Because our template does not store information on "CAPS LOCK" key, it does not distinguish between capital and small letters (*i.e.*, latencies of `ab` and `AB` are stored in the same cell "`ab`").

Outlier Detection: Latency values that markedly deviate from the majority of the latency values of a user can distort the typing profile of a user, especially if the profile contains statistics sensitive to the outliers (*e.g.*, mean). Outliers can occur, for example, when users pause while typing to compose, recollect, or find information. Several studies (*e.g.*, [21, 22]) performed outlier detection and reported performance gains. Therefore, we also included outlier detection in our experiments. We used a distance based outlier detection method that worked well in an earlier work [22]. The method performs two steps on each non-empty cell in our 26-by-26 template matrix–1) for each latency value in a cell, count the neighbors, *i.e.*, the number of latencies in the cell that occur within a predefined neighborhood threshold (r) and 2) a latency value is considered an outlier if the number of neighbors is less than α% of the total number of latencies in the cell. The distance between a latency value and its neighbor is calculated as the absolute difference in latencies. After performing trial experiments, we set r as 100 and α as 68%. Additionally, we discarded all latency values greater than 300ms.

Verifier: The majority of verifiers proposed in the KD literature are password or fixed-text based *login time* verifiers, *i.e.*, verification is performed *once* when a

user enters a password or a predefined text of fixed length. However, in continuous verification with KD, the text of the user is unconstrained (i.e., user is free to type anything) and the verifier should be able to base its decisions on any keystroke that the user types.

Matching Pairs: Because there are no constraints on what a user types during continuous verification, some keystrokes typed during the verification phase may not have reference signatures in the template. For example, if the user types "zen," the cell in the template corresponding to the letter pair "ze" may be empty (*i.e.*, may not contain a latency value). This situation can arise because the letter pair did not occur in the enrolment text used for building the template. This problem can be resolved by performing verification using letter pairs that are common to the template and the verification text. Following [1], we refer to these common letter pairs as *matching pairs*. In our experiments, the number of matching pairs "M" needed by the verifier to output a match score is a configurable parameter.

Keystroke Verification Loop: A verification loop (dotted box in Figure 2.1) repeats four steps: 1) when the user types, record keystroke events using a keylogger, 2) extract keystroke latencies from the event timestamps and process (*e.g.*, filter outliers), 3) perform verification after collecting M matching pairs from the text typed by the user and obtain a match score, and 4) output the match score or make a verification decision by comparing the score against a threshold (T). In a snoop-forge-replay attack, instead of typing text (in Step 1), the attacker *synthesizes* keystrokes with an emulator.

Performance Measures: Impostor Pass Rate (IPR) is the ratio of the number of impostor attempts wrongly accepted as genuine over the total number of impostor attempts. False Reject Rate (FRR) is the ratio of the number of genuine attempts wrongly rejected over the total number of genuine attempts. Equal Error Rate (EER) is the error rate at which IPR and FRR are equal. Detection Error Trade-off (DET) curves show how IPRs and FARs vary with verification threshold. We evaluated the snoop-forge-replay attacks using EERs and DET curves.

2.2 Related Research

2.2.1 Keystroke-based User Authentication Systems

Here, we briefly discuss related research in continuous authentication with KD. Monrose and Rubin in [2] proposed a continuous *identification* method (*i.e.*, a test sample was matched against all the users' templates to identify the closest user). Data was collected from 41 users over a period of seven weeks. However, because of errors, data from 31 users was used in the experiments. Each user typed from a few given phrases and/or sentences of their choice. Participants took part in the typing sessions at their convenience using their own computers. Details on sample sizes (*i.e.*, the number of characters typed by the users for enrolment and testing) is not mentioned. A user's template is comprised of means and standard deviations of key hold and key interval latencies. Latencies with values greater than T standard deviations from their means were treated as outliers and discarded from the template. After discarding outlier latencies, the means and standard deviations were recomputed. Identification was done by matching a test sample (with outliers removed) to the templates of

all users using: 1) the Euclidean distance measure, 2) the non-weighted probability measure, and 3) the weighted probability. The test sample was identified as belonging to the user with the minimum distance or maximum probability value. When free (*i.e.*, unstructured) text was used for enrolment and identification, the authors reported very low (between 17.1% and 23.0%) identification accuracies. However, when text from the given phrases was used, the authors reported 90% identification accuracy with weighted probability measure.

Dowland *et al.* in [3] also proposed a continuous identification method. In their method, the users typed free (unconstrained) text for enrolment and verification. A user's template consisted of the means and standard deviations for key press latencies. Only those latencies that occurred a minimum number of times were used to build the template. Latencies outside 40ms and 750ms range were excluded. During identification, a key press latency from the test sample was considered *valid* if its value was within $T.\omega$ standard deviations from mean value in the template (ω is a weight factor). The user with the highest number of valid key press latencies was considered as the owner of a given test sample. A total of 10 users participated in the data collection, but only four users who gave a large number of samples were used in the experiments. Dowland *et al.* in [3] reported that the best identification accuracy of their method was 50%. In [3], the authors improved this method and reported close to 60% accuracy.

Nisenson *et al.* in [23] proposed a continuous verification method. In their method, each user was treated as an emitter of discrete symbols, i.e., the user emits a sequence of key press events, key release events, and time differentials, which are

mapped to a finite symbol set. The template takes the form of a weighted phase tree built using the Lempel-Ziv universal prediction algorithm on a user's symbol sequences. Verification score was determined by calculating the conditional probability that a given sequence of symbols originated from a user's phase tree. A user was verified as genuine if the probability estimate of the his/her symbol sequence was greater than a threshold. Although this method achieved 96.77% accuracy, the accuracy was estimated with only five users (and with data from a few sessions), which leads us to question the reliability of the accuracy.

Gunetti and Picardi in [1] proposed a continuous authentication method. A user's template consisted of mean n-graph latencies. Authentication was performed using two measures: 1) the relative measure and 2) the absolute measure. The relative measure is the normalized rank disorder between n-graph latency pairs that are common to the template and the test sample. In this dissertation, we implement the relative measure as one of our baseline verifiers. In the absolute measure, a similarity score is computed between an n-graph in the template and the test sample as $\max(D_{temp}, D_{test})/\min(D_{temp}, D_{test})$, where D_{temp} and D_{test} are the n-graph durations in the template and the test sample, respectively. If the ratio is below a threshold, the n-graph is considered valid. Then the absolute measure is computed as the proportion of matching n-graphs that are also valid n-graphs. Gunetti and Picardi performed experiments with 40 users (treated as genuine), who provided 15 typing samples and 165 users (treated as impostors) provided one typing sample each. Best result (IPR:0.044%, FRR:6.833%) was achieved when relative and absolute methods were combined. However, in order to achieve these results, the method required

multiple sessions consisting of 700-900 keystrokes. When the number of keystrokes was reduced to $1/4^{th}$ original length (approximately 200 keystrokes), the accuracy dropped dramatically (IPR:0.3951%, FRR:29.1667%).

2.2.2 Non-zero Effort Attacks on Keystroke-based User Authentication Systems

To the best of our knowledge, this is the first work to propose *non-zero effort impostor attacks* against keystroke-based *continuous* user verification. The key difference between the feature-level attacks in [24] and the sample-level attacks is that, sample-level attack is an automated attack (*i.e.*, a computer program continuously generates key press and release events as if they were being produced by a legitimate user). On the other hand, a feature level attack requires the attacker to know what features the verification system is using. Additionally, the attacker has to know how to input the synthesized features directly into the verification algorithm, bypassing keystroke data acquisition, feature extraction, and preprocessing modules. Because the sample-level attack directly submits (fake) samples to the verification system, the attacker does not have to know the internal details of the system. So the sample-level attack is more practical and easier to launch compared to the feature-level.

Some previous studies in fixed-text (*i.e., password*) based keystroke verification systems used non-zero effort impostor attacks generated by *trained human* subjects. For example, [25] reported higher impostor pass rates when the impostor subjects were allowed to observe how a genuine user typed his/her password. [26] conducted experiments to examine how the amount of practice by the impostor, among other factors, affected the performance of password based keystroke verification systems.

[26] concluded that impostor's practice can be a "minor" threat to password based keystroke verification systems.

In the papers cited above, trained human impostors had to type short strings, containing at most 8 to 12 characters. Therefore, it was possible for a human impostor to practice and type like a genuine user. However, in continuous verification, the impostor would have to type much more. For instance, in our experiments, to generate one verification attempt with 20 matching pairs, a user typed on average 54 characters, which took 14.83 seconds (see Table 4.5, Page 45). At this rate, if the impostor has to continuously type for five minutes, he/she would end up typing more than 2000 characters. Typing so many characters with an intent to mimic a legitimate user is not an easy task for a human impostor. Furthermore, this type of attack is not easily scalable because of the human effort and resources involved. So, for keystroke-based continuous verification systems, impostor attacks by trained humans do not pose as much threat as *automated forgery attacks*, like the "snoop-forge-replay" attack presented in this dissertation.

Therefore, there are at least two practical bottlenecks in using human impostors against continuous keystroke verification systems: 1) how to train a human impostor to consistently type like a legitimate user for long durations and 2) if at all such training is possible, the resources (in time, effort, and monitory costs) needed to train the impostors to launch attacks against larger victim populations can be prohibitive.

Recently, [27, 28] reported an automated impostor attack against *short string* based keystroke verification. In [27] bots inject keystroke events on a client machine in a client-server model. The keystroke latencies are statistically-generated and assumed to

follow Gaussian distribution which are computed by the latencies of a small population of 20 users. [27] reported that these attacks were *ineffective*.

In [28] the attack is a guessing attack that incrementally searches the feature space of a large population of users until a feature vector that matches the target user's template is found. The master-key is repeatedly submitted to the verification algorithm, and every time it is submitted, one of its feature value is changed by one standard deviation until the verification algorithm declares that the key has matched the victim's template. To speed-up the attack, instead of changing one feature value at a time in the master-key, [28] identified conditionally dependent feature pairs and changed two feature values at a time. The attack in [28] works against login-time verification with short pass-phrases. However, the attack is not suitable against continuous verification because: 1) the attack assumes that the text is fixed, *i.e.*, the latencies from which the master-key is derived and the templates that are being attacked come from typing the *same text*, an assumption clearly invalid in continuous verification where the users are free to type any text; and 2) the attack performs a "brute force" search in a feature space that grows exponentially with the length of the pass-phrase. Consequently, with longer pass-phrases, the attack tends to make more erroneous attempts (*i.e.*, infertile guesses) before converging to a vector that successfully passes verification. In continuous verification, a verification attempt has more characters. For example, in our experiments, verification attempts had between 54 characters (with 20 matching pairs) and 2000 characters (with 750 matching pairs). With so many characters, the attack in [28] would have made thousands of unsuccessful

attempts before producing a successful attempt. With so many unsuccessful attempts, a continuous verification system could easily be alerted.

2.2.3 Non-zero Effort Attacks on Other Behavioral Biometric Authentication Systems

The work in this dissertation was motivated by the findings of two studies: 1) [29] studied the effect of forgery quality on handwriting biometric security and showed that impostor pass rates of trained and generative (*i.e.*, "algorithmic") forgery attacks outperformed naive forgeries and 2) [30] evaluated spoofing attacks on gait authentication and showed that attackers with knowledge of their closest person in the database can significantly raise impostor pass rates. Below, we briefly discuss [29], which is closer to our work in this dissertation.

[29] reported the effect of six types of forgery attack models on handwritten signature based verification. One of them was the generative forgery model, which involved algorithmically generating forgeries of a target writer by collecting a small set of writing samples from 1) the target writer (these samples were referred as "parallel corpus") and 2) a set of different writers. Results in [29] showed that, compared to trained human forgers, generative attacks had higher impostors pass rates for block and cursive writers but had lower rates for mixed writers. A notable similarity between the generative attacks in [29] and the snoop-forge-replay attack is that both require a surprisingly low number of stolen samples to generate effective attacks.

CHAPTER 3

SNOOP-FORGE-REPLAY ATTACK METHODS

The attack presented in this dissertation falls under the broader class of generative attacks on behavioral biometric systems [29], but is tailored to attack continuous keystroke-based verification systems. Below, we discuss the steps in snoop-forge-replay attack.

3.1 Snooping Keystroke Timing Information

In this step, the attacker secretly steals a victim's keystroke timing information. For example, if the victim typed the text, "`this is snooped text`," the attacker records a series of timestamps–$P_{\texttt{t}}$ (time when `t` was *pressed*), $R_{\texttt{t}}$ (time when `t` was *released*), $P_{\texttt{h}}$, $R_{\texttt{h}}$, $P_{\texttt{i}}$, $R_{\texttt{i}}$, $P_{\texttt{s}}$, $R_{\texttt{s}}$, $P_{\texttt{SPACE}}$, $R_{\texttt{SPACE}}$, and so on.

An attacker can snoop a victim's keystroke timing information using a hardware keylogger or a software keylogger. Software keyloggers have become the most popular forms of keyloggers because they can be easily developed, are easily available,[1] and can be deployed from remote locations onto a victim's machine (*e.g.*, using trojans and spyware).

[1]Attackers can access hundreds of software keyloggers from code-sharing websites like `www.SourceForge.net`. Anti-Phishing Working Group (`www.antiphishing.org`) reported in [31] that 3121 websites hosted keyloggers in February 2007 alone.

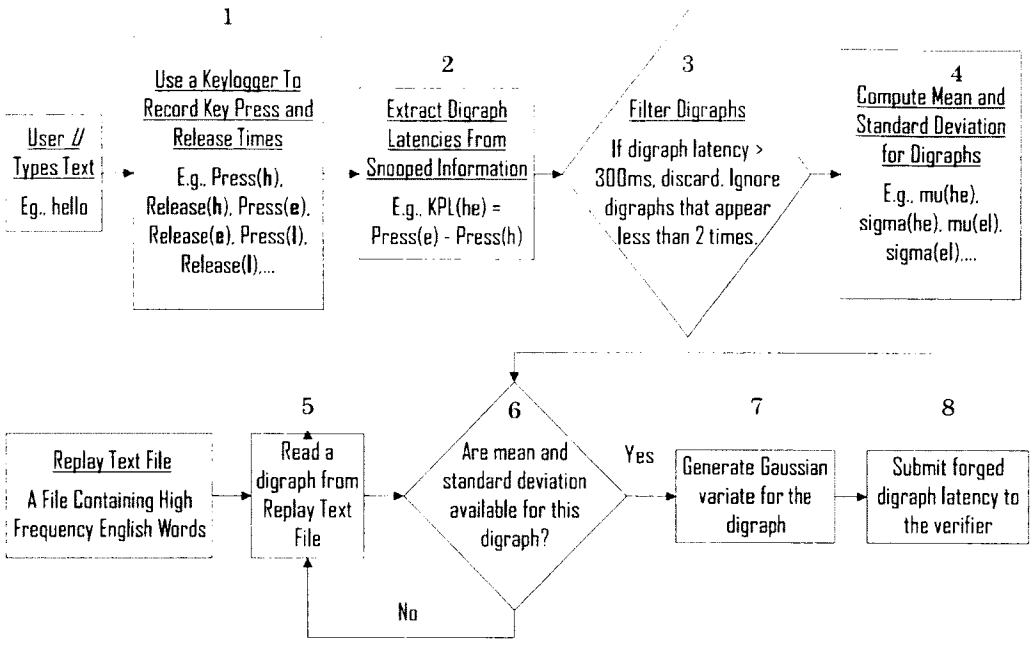


Figure 3.1: Snoop-Forge-Replay attack flowchart. Step (1)–snoop keystroke timings. Steps (2)-(8)–create and replay a forged attempt.

We used keystroke data collected from 150 participants during the period 13-21 October 2009 as *snooped* keystrokes (see Section 3.5 on Page 33 for details). This data was collected using a software keylogger developed in C#. The snooped keystrokes were used to attack templates that were built from keystrokes collected approximately *six months after* the snooped keystrokes.

3.2 Creating a Keystroke Forgery

3.2.1 Creating Forgeries in Feature-level Attack

For creating a forged attempt at the feature-level, we assume that the attacker has the following knowledge.

Knows how to compute keystroke features (*i.e.*, hold, interval, and digraph latencies): An attacker can acquire this information by reading keystroke

dynamics (KD) literature or can use keylogger codes widely published on the Internet. For instance, attackers can access hundreds of keylogger codes from code-sharing websites like `SourceForge.net`.

Knows how to synthesize a keystroke feature from a probability distribution: Though we are not aware of any study that conclusively establishes that keystroke features follow Gaussian distribution, several KD authentication studies have assumed Gaussianality of keystroke features. An attacker can either follow these studies or explore distributions other than Gaussian. However, from an attacker's viewpoint, the Gaussian assumption is appealing because its parameters (*i.e.*, mean and standard deviation) are easy to compute and most programming languages provide tools to generate Gaussian variates.

Knows how to handle extreme feature values: It is well known that the mean is sensitive to extreme values. Based on this knowledge, an attacker can choose to discard large feature values before computing the mean. On the other hand, an attacker can choose to ignore this step.

For each user U_i, we create forgery attempts as follows. We extract digraph latencies from U_i's snooped keystroke timings (see Step 2, Figure 3.1). We assume that digraph latencies follow Gaussian distribution. We implement a simple filter that discards digraph latencies greater than 300ms (Step 3, Figure 3.1). After filtering, we compute the means and standard deviations of the digraphs (Step 4, Figure 3.1).

3.2.2 Creating Forgeries in Sample-level Attack

In this step, we create a keystroke forgery of a victim user U_i at sample-level. A forgery has two parts: 1) "dummy" text and 2) a series of latencies between the press and release of letters in the dummy text. For example, a forgery of U_i can have the dummy text, "`this is dummy text`." The key hold and interval values for this text come from the snooped keystroke latencies of U_i.

Computing key hold and interval latencies from U_i's snooped keystrokes: Let P_t, R_t, P_h, R_h, P_i, R_i, P_s, R_s, P_{SPACE}, R_{SPACE}, P_i, R_i, and so on be the snooped keystroke timestamps collected when U_i typed the text, "`this is snooped text`." Using these snooped timestamps, the attacker computes $kh_{t:h} = R_t - P_t$ (key hold latency of `t`, when the next character pressed is `h`), $ki_{t:h} = P_h - R_t$ (key interval latency between `t` and `h`), $kh_{h:i}$, $ki_{h:i}$, $kh_{i:s}$, $ki_{i:s}$, and so on. If any latency occurs more than once, we take the average. Next, we use the snooped "kh" and "ki" values as the key hold and interval latencies of the letters in the dummy text. For example, in the forgery containing the dummy text, "`this is dummy text`," we use $kh_{t:h}$ for forging the key hold latency of the first "`t`" in the dummy text, and in the same fashion, use $ki_{t:h}$ for forging the key interval latency of "`th`" in the dummy text.

What if there are letters in the dummy text for which snooped latencies are not available?: Because our primary goal is to demonstrate how forgeries based on snooped keystrokes can be used to evade detection, when preparing a forgery, we ignored those letters in the dummy text for which corresponding snooped latencies were not available. This sometimes could render the text generated by the forgery linguistically meaningless, especially when forgery is created from limited

amounts of snooped text. However, note that current keystroke-based continuous verification systems, to the best of our knowledge, do not check the language generated by the typist, and therefore, our attack in its present form straightforwardly exploits this vulnerability.

If the attacker wants to forge specific words to execute a series of commands, then the attacker can choose to fill the missing latencies with very large values, so that they are filtered by the outlier detection method and thus are disregarded by the continuous verifier. An alternate way is to fill the missing latency values using latencies computed from a population of users (as done in [29] for spoofing handwritten signatures).

Table 3.1: Characteristics of the "dummy text" file used in our experiments.

Text Source	497,184 words from COCA corpus and 20 Wikipedia documents
Total # of unique letters	26
Total # of unique digraphs (letter pairs)	676
Total # of letters	5,021,665
Total # of digraphs	4,222,420
10 most frequent letters	e, a, i, n, r, o, t, s, l, d
10 least frequent letters	q, x, j, z, v, w, k, y, f, b
10 most frequent digraphs	in, er, an, on, re, ed, te, ar, en, es
10 least frequent digraphs	qk, jq, qj, xk, jk, qy, qz, vq, qz, qh

Preparing a "dummy text" file: The "dummy text" file supplies text to create a forgery. Technically, the file can contain any text, ranging from multiple repetitions of a single letter (*e.g.*, `aaa...`) to a large text corpus representative of English language usage (*e.g.*, Corpus of Contemporary American English (COCA) [`www.americancorpus.org`]). For our experiments, we created a "dummy text" file

with 497,184 words from COCA[2] [32]. In addition, we added text from 20 Wikipedia documents. (In Appendix A, we give the web addresses of the Wikipedia documents). In Table 3.1 we summarize the characteristics of dummy text.

3.3 Replaying a Forgery of Victim U_i:

3.3.1 Replaying Forgeries in Feature-level Attack

Replaying a forged attempt of a user U_i involves generating keystroke timing events as if U_i was typing some text. Replaying involves two key components: 1) a database containing text and 2) a "replayer" software, which reads the text in the database and generates keystroke timing events. Below, we explain the two components in detail.

Text Database: In feature-level attack, to supply text to the replayer we used a file containing Fry's Instant Word List [33], which is a list of 1,000 most common words occurring in the English text, arranged in frequency order. Fry F[33] showed that the first 300 words in this list make up 65% of all written material.

Replayer Software: A replayer can be implemented in two ways: 1) the replayer *emulates* U_i's typing behavior by synthesizing actual key press and release events for the text in the database and 2) the replayer reads the text and generates feature values (e.g., digraph latencies) for user U_i and submits it to the verifier.

Generating keystroke feature values: A replayer can replicate U_i's typing behavior is by reading text from the database and directly supplying keystroke feature values (digraph latencies in our case) to the continuous verification system. Again,

[2]COCA is a large, freely-accessible text corpus on the web. The corpus contains 410 million words (20 million words each year from 1990 through 2010).

the feature values can be generated using a Gaussian random number generator, with means and standard deviations calculated by snooping U_i's keystrokes.

3.3.2 Replaying Forgeries in Sample-level Attack

Keystroke Emulator: We developed a keystroke emulator that *injects* synthetic key press and release events. We programmed the emulator in Visual C++ and used `SendInput` API. The goal of the emulator is to use the snooped latencies to inject key press and release events for the dummy text in a way that the verifier thinks that it is the victim U_i who is typing the dummy text. The emulator algorithm, referred to as "Algorithm 1," gives the steps to forge and replay a victim user U_i's typing pattern.

The input to the algorithm is a dummy text file and a series of key hold and interval latencies computed from U_i's snooped keystrokes. We initialize variables in lines 2–9. The *trap_counter* variable is used when the program encounters a character pair in the dummy text for which a snooped latency is not available. The *trap_counter* variable counts the number of characters to be traversed in the dummy text, to find a character pair for which a snooped latency is available.

In line 10, the **while** loop ensures that Algorithm 1 replays the dummy text for at least 24 hours. In lines 11-26, we create and replay a forgery. The **If** condition in Line 14 is executed when a letter pair from the dummy text (stored in *first* and *second* variables) has corresponding snooped latencies. If snooped latencies are available, Algorithm 1 calls the `replay` function in line 16 to generate key press and release events. The **else** in Line 20 is executed if the letter pair (*first, second*) does not

ALGORITHM 1: Replay the forgery of user U_i.

Input: Dummy text file containing 497,184 words from COCA and text from 20 Wikipedia pages. Key hold (*e.g.*, $kh_{m:y}$, $kh_{y:SPACE}$, *etc.*) and key interval (*e.g.*, $ki_{m:y}$, $ki_{y:SPACE}$, *etc.*) latencies computed from U_i's snooped keystrokes. Here, "$kh_{x:y}$" denotes the *snooped* key hold latency of x when the next character typed is y and "$ki_{x:y}$" denotes the *snooped* key interval latency between characters x and y.

Output: A replay of user U_i's keystroke forgery.

1 **Initialization:**
2 $n \leftarrow$ Number of characters in the dummy text file.
3 *dummyTextArr* [0:$n-1$] $\leftarrow$ Copy each character in the dummy text file into the array: */*Each cell in the dummyTextArr holds a character in the dummy text file*/*
4 *dummyIndex* $\leftarrow$ 0; */*Index to the first character in the dummyTextArr */*
5 *trap_counter* $\leftarrow$ 0; */*Counter to ensure that character pairs in the dummy text that not do have corresponding snooped latencies do not stagnate the replay. If the snooped latencies are not available even after traversing 500 characters in the dummy text, then character pair is reset to a random character in dummyTextArr (Line 25)*/*
6 *first* $\leftarrow \emptyset$; */*A variable to store first character.*/*
7 *second* $\leftarrow \emptyset$; */*A variable to store second character.*/*
8 *startTime* $\leftarrow$ System time at the start of the program;
9 *currentTime* $\leftarrow$ Current system time;
10 **while** *(currentTime − startTime ≤ P hours) /*We set P to 24.*/* **do**
11 *first* $\leftarrow$ *dummyTextArr* [*dummyIndex*];
12 *second* $\leftarrow$ *first*;
13 **while** *dummyIndex* $<$ *n and trap_counter* $\leq$ *500* **do**
14 **if** ($kh_{first:second}$ and $ki_{first:second}$) is snooped /*checks if letter pair from the dummy text has corresponding snooped latencies.*/ **then**
15 $KH_{first:second} \leftarrow kh_{first:second}$; $KI_{first:second} \leftarrow ki_{first:second}$; */*Forge latencies. "KH" and "KI" denote latencies in a forgery.*/*
16 `replay` (*first*, $KH_{first:second}$, $KI_{first:second}$); */*Replay dummy text by generating key press and release events of first when second is the next character*/*
17 *first* $\leftarrow$ *second*; *trap_counter* $\leftarrow$ 0;
18 **end**
19 **else**
20 *trap_counter* $\leftarrow$ *trap_counter* +1;
21 **end**
22 *dummyIndex* $\leftarrow$ *dummyIndex* +1;
23 *second* $\leftarrow$ *dummyTextArr* [*dummyIndex*];
24 **end**
25 *dummyIndex* $\leftarrow$ Reset to a random cell of *dummyTextArr*;
26 *currentTime* $\leftarrow$ Current system time; *trap_counter* $\leftarrow$ 0;
27 **end**

1 **PROCEDURE 1:** `replay(char c, real hold_delay, real interval_delay)`
Input: The character *c* to be replayed, key hold delay of the character, and key interval delay between the character and the next consecutive character.
Output: Generate press and release events for character *c*.
2 **INPUT** **key*; */*Iniatialize key as the pointer to* **INPUT** *structure.*/*
3 *key* = **new INPUT**;
4 *key* →type = INPUT_KEYBOARD; */*Keyboard event. Use the* "ki" *structure of the* **INPUT**.**/*
5 *key* →ki.wVk = *c*; */*Assign character c to the keyboard event.*/*
6 *key* →ki.dwFlags = KEYEVENTF_KEYDOWN; */*The event is a key press event.*/*
7 `SendInput` (1,*key*,`sizeof` (**INPUT**)); */*Press c.*/*
8 `sleep` (*hold_delay*); */*Sleep for hold_delay milliseconds to generate key hold time.*/*
9 *key* →ki.dwFlags = KEYEVENTF_KEYUP; */*The event is a key release event.*/*
10 `SendInput` (1,*key*,`sizeof` (**INPUT**)); */*Release c.*/*
11 `sleep` (*interval_delay*); */*Sleep for interval_delay milliseconds to generate interval time.*/*

have corresponding snooped latencies. The *trap_counter* ensures that if the snooped latencies $\text{kh}_{first:second}$ and $\text{ki}_{first:second}$ are not available even after traversing 500 (an arbitrarily chosen number) characters in the dummy text, then the *first* and *second* are reset to a random character in the *dummyTextArr* (Line 25).

In Procedure 1, we outline the implementation of the `replay` function (Line 16 in Algorithm 1). The function takes three parameters: 1) c is the character key that has to be pressed and released, 2) *hold_delay* is the delay (in milliseconds) between the press and release of c, and 3) *interval_delay* is the delay (in milliseconds) between c and the subsequent character in the dummy text. To generate the key press and release events, the `replay` function uses `SendInput` function (lines 7 and 10), which is a part of Windows Application UI Development API [18]. The latencies between press and release events are generated using the *sleep* function (lines 8 and 11).

Detailed explanation of lines 10-27 in Algorithm 1: We explain lines 10–27 with an example. Let the snooped text be "`this is snooped text`" and the dummy text be "`this is dummy text`." From line 3, *dummyTextArr*[0] = `t`, *dummyTextArr*[1] = `h`, *dummyTextArr*[2] = `i`, *dummyTextArr*[3] = `s`, and so on. From lines 11 and 12, *first* = *second* = `t`. Because *dummyIndex* = 0 and *trap_counter* = 0, the **if** condition in line 14 checks if $\text{kh}_{\texttt{t:t}}$ and $\text{ki}_{\texttt{t:t}}$ are available from the snooped keystrokes. Because they are not available (note "`tt`" is not present in the snooped text) , lines 21 and 22 in **else** are executed, resulting in *dummyIndex* = 1, *second* = `h`, and *trap_counter* = 1. In the next iteration, the **if** condition in line 14 checks to see if $\text{kh}_{\texttt{th}}$ and $\text{ki}_{\texttt{th}}$ are available. Because both $\text{kh}_{\texttt{th}}$ and $\text{ki}_{\texttt{th}}$ are available from "`th`" in the snooped text, (in Line 15) $\text{kh}_{\texttt{th}}$ is assigned to $\text{KH}_{\texttt{th}}$ and $\text{ki}_{\texttt{th}}$ is

assigned to KI_{th}. Then, in Line 16, KH_{th}, KI_{th}, and *first* (= t) are passed to the `replay()` procedure. In lines 17 and 18, *first* = h, *dummyIndex* = 2, *trap_counter* is reset to zero, and *second* = i. In the subsequent iterations, *first* takes the values: "i," "s," "SPACE," "i," "s," "SPACE," and correspondingly, *second* takes the values "s," "SPACE," "i," "s," "SPACE," and "D." In the next iteration, *first* = SPACE and *second* = D, lines 21 and 22 are executed because $kh_{SPACE:D}$ and $ki_{SPACE:D}$ are not available from the snooped text. From lines 21 and 22, *dummyIndex* = 9, *second* = u, and *trap_counter* = 1. Because $kh_{SPACE:u}$ and $ki_{SPACE:u}$ are also not available from the snooped text, lines 21 and 22 are again executed, resulting in *dummyIndex* = 10, *second* = u, and *trap_counter* = 2. Lines 21 and 22 are executed in the next few iterations, each time incrementing the *trap_counter*, until *second* = t. Because $kh_{SPACE:t}$ and $ki_{SPACE:t}$ are available from the snooped latencies, lines 15-18 are executed. In this fashion, Algorithm 1 continues to execute for 24 hours. The final output of Algorithm 1 is the replay text: `this is text.....`

At this point, we emphasize that Algorithm 1 is *one of the many* possible ways to generate snoop-forge-replay attacks. While maintaining the general idea of snooping and replaying the keystrokes, the attacker can evolve Algorithm 1 in several ways. For example, the attacker can devise heuristics to impute missing latency values or snoop only selected latencies from a victim, to generate desired text or system commands.

3.4 Virtualization Set-up for Forging and Replaying Sample-level Attacks on a Large Scale

To launch a large number of attacks on 150 victim users (see Contribution 3. Page 2), we built a virtualization set-up at Louisiana Tech University's Cyber Security Laboratory. The set-up had 150 virtual machines (VMs). Each VM ran a copy of our keystroke emulator program (*i.e.*, Algorithm 1) to generate forgeries for one victim user.

To create the virtualization set-up, we used a cluster of 8 Dell PowerEdge M710 Blade servers, each with 12 core Intel Xeon 3.33GHz processors. Each server was equipped with 96GB main memory. Four Dell EqualLogic iSCSI storage arrays provided 20TB secondary memory. We used VMware 4.1 vSphere Suite [34] to create 150 VMs. Each VM had Windows XP 32-bit operating system, 2GB main memory, and 10GB secondary storage. In each VM, emulator and "notepad.exe" file (to create an active Notepad window for `SendInput` API) were executed using Powershell 2.0 scripts. We used Condor 7.6.0 software [35] to schedule the execution of Powershell scripts in 150 VMs, simultaneously.

The keystroke emulator takes snooped timing information and "dummy text" file as input, injects key press and release events, and outputs a file containing replay text. When the emulator was running, we used a software keylogger to record the synthetic keystroke events generated by the emulator. The keylogger recorded the keystroke events and stored them in "user$N$.txt" files (where $N = 1$ to 150). A total of four files were associated with each user: two *input* files, one containing snooped keystroke timing information and another containing dummy text), and two *output*

files (*i.e.*, the text file generated by the emulator and "user$N$.txt" recorded by the keylogger). After running the emulator for 24 hours, the "user$N$.txt" (N = 1 to 150) files were collected from Condor server's shared memory and passed through the verifiers to generate verification scores.

3.5 Keystroke Data Collection

We used keystroke data collected from 350 participants at Louisiana Tech University. Majority of the participants were students, but university faculty and staff also participated. We used six Pentium IV desktop PCs to collect keystroke data. The PCs were equipped with Windows XP OS, a QWERTY keyboard, and a mouse. On each PC, we installed an interactive keystroke data collection software developed in C#. Additionally, we used two laptops to register participants and collect voluntary information (such as gender, ethnicity, typing experience, native language, is the participant left- or right-handed, is the participant willing to participate in a future data collection effort, and the participant's university email address).

We collected data during three different periods: October 13-October 21, 2009; April 4-April 30, 2010; and 25 October-9 November, 2010. On all days, we started data collection at approximately 8:00 AM and concluded at 5:00 PM. Each participant was required to register by presenting his/her university ID card. We recorded the following information during registration: 1) first name, 2) last name, and 3) voluntary information. We used two popular locations in the university to set up data collection booths. At each location, we used three PCs for collecting keystroke samples. We

continued data collection during lunch hours (11:30 PM to 1:30 PM) as this was the period of heightened student/faculty traffic.

After registration, we instructed each participant on how to use the data collection software and asked the participant to type *three* types of text: 1) *fixed text*–participant typed the phrase "`I am an undergraduate student of Louisiana Tech University`" 15 times (12 times during October 2009); 2) *copy text*–each participant typed several paragraphs of English text from a document provided by us; and 3) *self text*–participant had to compose and type text. Unlike in fixed text, where the participant had to type the predefined text *exactly*, there were no restrictions on typing copy and self texts. For example, the participants were allowed to make spelling mistakes and typographical errors, and if they chose, they could correct them using Backspace or Delete keys. In the experiments, we do not use fixed text data, so we skip further details on it. For entering fixed text, the GUI included three text boxes: one for entering username, one for entering campus-wide ID number, and one for entering a common fixed phrase.

The keystroke data collection software provided GUI (*e.g.*, text boxes, buttons, and character counters) for typing copy and self texts. Each participant was required to type at least 1800 characters (1200 during October 2009) of copy text. For typing copy text, we provided paper copies of sample texts to the participants. We used five sample texts: 1) Declaration of Independence [36], 2) a transcript of Richard Hamming's "You and Your Research" speech [37], 3) the first 2100 words in Chapter 1 of David Copperfield [38], 4) the first 2000 words in Chapter 1 of Samuel Johnson [39], and 5) the first 1900 words in Chapter 1 of Walden [40]. A participant received one of the five

Table 3.2: A summary of keystroke data usage in our sample-level attack experiments.

Dataset Name	**Set I**	**Set II**	**Set III**	**Set IV**
Collection Dates	13–21 Oct., 2009	1–30 April, 2010	25 Oct. – 9 Nov., 2010	25 Oct. – 9 Nov., 2010
Usage	Snooped keystrokes	Building keystroke templates	Generating genuine & zero-effort impstr. scores	Generating zero-effort impstr. scores
# of users	150 common users in sets I, II, and III			200 new users

sample texts randomly. As the participant typed, the software displayed the number of characters typed. Copy text data collection process ended when the participant typed at least 1800 (1200 during October 2009) characters. After entering the copy text, the participant was required to type about 300 characters of self text. After typing the self text, the participant pressed the "Finish" button and ended his/her data collection session. Self text was collected during April 2010 and October-November 2010 periods.

Copy vs. self text: When performing activities like writing emails, messaging, and word-processing, users typically do compositional typing (*i.e.*, text composition and typing occur as an intertwined sequence of events). Thus, typing self text is a closer representation of a user's typing activity. However, we conducted pilot trials in our laboratory before undertaking full-scale data collection and observed that typing 1200-1800 characters of self text took considerably more time than typing copy text of the same length and in most cases fatigued participants. Because the majority of the participants were students who participated between classes, time was a critical factor for their participation. To achieve a trade-off between participation time and obtaining realistic typing samples, we choose to collect a mixture of copy and self texts.

Keystroke Data Usage in Our Experiments

1) For feature-level attack experiments: we used data from 50 users who participated during three data collection periods, *i.e.*, October 2009, April 2010, and October-November 2010. We used keystroke events generated by typing free and self texts during April 2010 for training (*i.e.*, building keystroke templates). For generating verification attempts, we used keystrokes from free and self texts collected during October-November 2010. We used the keystroke events obtained from typing free text during October 2009 as snooped keystroke data.

2) For sample-level attack experiments: we divided the keystroke data into four sets (see Table 3.2). Set I has keystrokes collected from 150 users during October 2009. For the same 150 users, Set II has keystrokes collected during April 2010 and Set III has keystrokes collected during October–November 2010. We used keystrokes in Set I as *snooped* keystrokes. We used keystrokes in Set II to build 150 user templates. We used keystrokes in Set III to generate genuine and zero-effort impostor scores. To generate genuine scores, we matched each user's template with his/her own keystrokes in Set III. To generate zero-effort impostor scores, we matched a user's template with keystrokes of 149 remaining users in Set III.

Purpose of Set IV: Set IV contains keystroke samples from a new pool of 200 users who are not present in sets I, II, and III. We matched keystrokes in Set IV against the user templates to generate additional zero-effort impostor scores. We did this to compare the snoop-forge-replay attack scores with two baselines: 1) zero-effort impostor scores generated with Set III, and 2) zero-effort impostor scores generated with Set IV.

Note from the "Collection Dates" row in Table 3.2 that there is approximately *six months time gap* between snooped keystrokes (Set I) and keystroke used to build templates (Set II). This manner of data usage is akin to a scenario in which the attacker uses old "legacy" keystrokes to attack a victim user's template.

3.6 Baseline (Zero-effort Impostor Attack) Experiments

For feature-level attack experiments: we experimented with *two* verifiers: 1) Relative (R) verifier [1], and 2) Similarity (S) verifier. We used *one* type of template: TKP–template containing key press latencies.

For sample-level attack experiments: we experimented with *four* verifiers: 1) Relative (R) verifier [1], 2) Absolute (A) verifier [1], 3) Similarity (S) verifier [41], and 4) Fusion (F) verifier. We used *three* types of templates: 1) TKH–template containing key hold latencies, 2) TKI–template containing key interval latencies, and 3) TKP–template containing key press latencies. This resulted in nine verifier-template combinations *i.e.*, (R, TKH), (R, TKI), (R, TKP), (A, TKH), (A, TKI), (A, TKP), (S, TKH), (S, TKI), and (S, TKP). The "F" verifier fuses the outputs from (R, TKH), (R, TKI), (S, TKH), (S, TKI), and (A, TKP) using the weighted sum fusion rule [42].

Extracting verification attempts: From a user's typing sample, we extracted verification attempts as follows: 1) read the text in the order it was typed and extract latencies until M matching pairs are obtained; 2) present the M matching pairs to the verifier to obtain a verification score (this constitutes one verification attempt); 3) read the text from the point where it was stopped in Step 2 until M matching pairs are obtained; and 4) repeat Steps 2 and 3 until the text ends. This procedure

partitions the text into contiguous, non-overlapping, variable-length windows, each containing exactly M matching pairs. Each window corresponds to one verification attempt. We experimented with M values: 20, 40, 60, 80, 100, 120, 150, 300, 350, 500, and 750.

Relative (R) and Absolute (A) Verifiers [1]: Given a verification attempt, "R" verifier outputs a score as follows. Two arrays A_{train} and A_{test} are constructed. A_{train} contains the matching pairs ranked in ascending order of their corresponding mean latencies (in the template). A_{test} contains the matching pairs ranked in ascending order of their latencies in the verification attempt. The "R" measure between A_{train} and A_{test} is computed as the normalized array disorder between A_{train} and A_{test}. The "R" measure lies between 0 and 1, 0 (or 1) indicates a perfect match (or mismatch) between the verification attempt and the template.

The "A" measure verifier outputs a score as follows: for each matching pair, two latency values are considered: 1) the average latency value stored in the template, and 2) the average latency value in the verification attempt. The larger of the two is divided by the smaller. A matching pair becomes *valid* if the ratio falls between 1 and a threshold (after some trial and error experiments, we choose 1.45 as threshold). The "A" measure is given as

$$1 - \frac{\text{number of valid matching pairs}}{\text{total number of matching pairs}}. \tag{3.1}$$

The "A" measure of 0 (or 1) indicates a perfect match (or mismatch) between the verification attempt and the template.

Similarity (S) Verifier [41]: The "S" verifier outputs a verification score as follows: each matching pair in the verification attempt is considered a *valid* matching pair if it falls within $T(=1)$ standard deviations from its corresponding mean in the template. The similarity measure between the template and the verification attempt is calculated using (3.1).

Fusion ("F") Verifier: The verifier fuses outputs of five verifier-template combinations, (R, TKH), (R, TKI), (S, TKH), (S, TKI), and (A, TKP), using the weighted fusion rule. If s_1, s_2, s_3, s_4, and s_5 are outputs of the five verifier-template combinations, then by the weighted fusion rule, "F" outputs $\omega_1 s_1 + \omega_2 s_2 + \omega_3 s_3 + \omega_4 s_4 + \omega_5 s_5$, where $0.1 \leq \omega_i \leq 0.6$ and $\sum_{i=1}^{5} \omega_i = 1$. We included weighted fusion in our experiments because studies (*e.g.,* [42, 43]) show that it performed well in biometric authentication tasks. For fusion, we used five out of the nine available verifier-template combinations because key press latencies are formed by adding key hold and key interval latencies, so including (R, TKP) and (S, TKP) do not bring new information when (R, TKH), (R, TKI), (S, TKH) and (S, TKI) are already included in the fusion. The "A" verifier was primarily designed for key press latencies, so we included only (A, TKP) in the fusion. *Choosing weights*: we experimented with 126 weight combinations. Initially, we set $\omega_i = 0.1, i = 1, \cdots, 4$ and $\omega_5 = 0.6$. Then we incremented (or decremented) the weights in 0.1 units under the constraints: $0.1 \leq \omega_i \leq 0.6$ and $\sum_{i=1}^{5} \omega_i = 1$.

CHAPTER 4

RESULTS

4.1 Feature-level Attack Experiments

4.1.1 Baseline (Zero-effort Impostor Attack) Results

We performed baseline experiments to find the optimal number of matching pairs (M) that yield the lowest FRR and IPR values. The DET curves of "R" verifier, *i.e.*, plots 4.1(a)–4.1(b) and "S" verifier, *i.e.*, plots 4.1(c)–4.1(d) (in Figure 4.1) show that lower FRR and IPR values are achieved with high M values (i.e., 300, 350, 500, and 750). However, a high M requires typing more keystrokes to generate a verification attempt, which ultimately increases the time to output a match score. In our experiments, it took on an average of 49.36 keystrokes (14.305 seconds average typing time) to obtain a verification attempt with 20 matching pairs and 1836.42 keystrokes (545.307 seconds average typing time) to obtain a verification attempt with 750 matching pairs. Therefore, a trade-off exists between the number of matching pairs and the verification delay. DET curves for 20, 40, 60, and 80 matching pairs (not shown due to space constraints) had higher EERs than the DET curves in Figure 4.1.

In Table 4.1, we give the number of genuine and impostor attempts used for generating DET curves in Figure 4.1. In our test data, on an average there were 2100 keystrokes per user. So, for high M values like 750, 500, and 350, on average, we

Table 4.1: Total genuine attempts ($\#G$), genuine attempts/user (G/U), and impostor attempts/user (I/U) used in Figure 4.1 DETs.

M	100	120	150	300	350	500
$\#G$	570	473	371	175	141	94
G/U	11.4	9.46	7.42	3.50	2.82	1.88
I/U	555.3	456.8	360.8	168.6	140.1	90.3

needed 1836.42, 1212.42, and 853.69 keystrokes respectively to generate a verification attempt. For this reason, we had very few genuine attempts for each user when M was high (Table 4.1, second row). However, we did not have this problem for impostor attempts because, for each user, we used the keystroke data of the remaining 49 users to generate impostor attempts.

4.1.2 Snoop-forge-replay Attack Results and Analysis

In this section, we demonstrate the success rates of forged verification attempts created from snooping a user's keystroke data. We choose to use M values with baseline EERs less than 0.15. Figure 4.1 shows that this is achieved when M is 150, 300, 350, 500, and 750 for both verifiers. However, we excluded 750 matching pairs because the average number of keystrokes it required to generate a verification attempt (1836.42) was too high to be realistic. Verification threshold for each M value was selected using two heuristics: 1) threshold T_1 with the least IPR when $FRR < 0.15$ and 2) threshold T_2 for which $FRR + IPR$ is minimum (i.e., lowest point in the DET curve) when $FRR \leq 0.15$. In Table 4.2, we give the baseline (*zero-effort*) IPRs for verifiers "R" and "S" for thresholds T_1 and T_2.

The plots in Figure 4.2(a), 4.2(c), 4.2(e), and 4.2(g) show the percentage of successful forgery attempts with threshold T_1 (see Table 4.2). We considered two

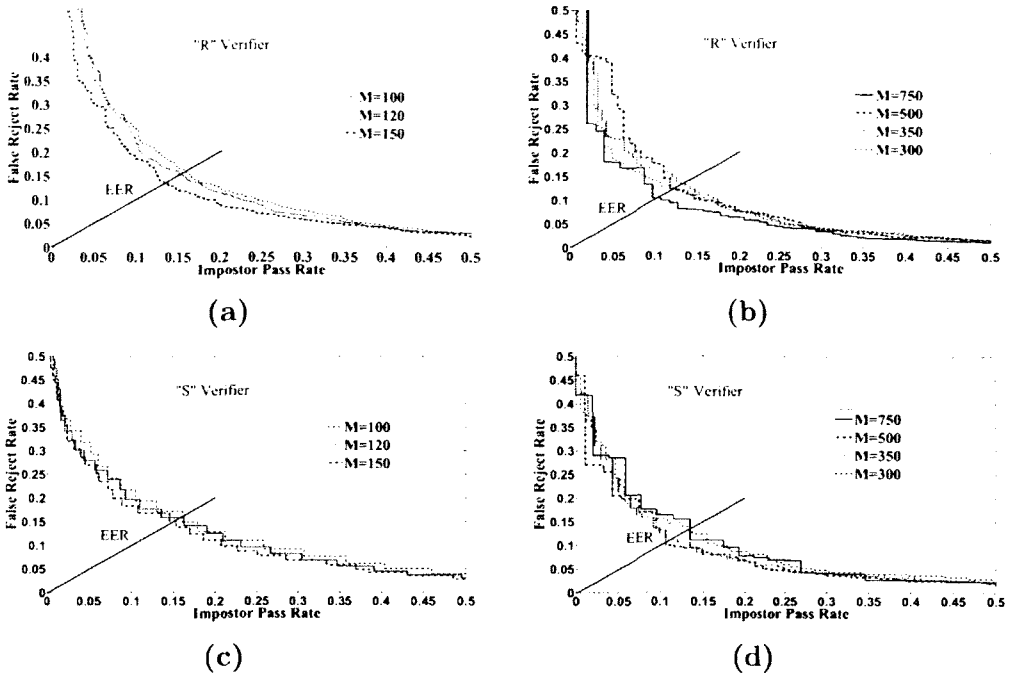


Figure 4.1: Baseline DET curves of "R" (a) & (b) and "S" (c) & (d) verifiers with M matching pairs.

factors: number of snooped keystrokes and filtering (no filtering) snooped digraph latencies greater than 300ms. When forged attempts were created by estimating the means and standard deviations of digraph latencies and replayed using a Gaussian random number generator, we observe (in plots 4.2(a) and 4.2(e)) that filtering increases the percentage of successful forged attempts. With "S" verifier, the maximum percentage of successful forgeries with filtering was 87.58 (for $M = 150$ and 200 snooped keystrokes) and 72.06% (for $M = 150$ and 150 snooped keystrokes) without filtering. With R verifier, the maximum percentage of successful forgeries was 79.32 with filtering (for $M = 500$ and 200 snooped keystrokes) and 72.73% (for $M = 500$ and 100 snooped keystrokes) without filtering. For these M values, in plots 4.2(b) and 4.2(f), we compare baseline DET curves obtained with zero-effort impostors and the DET curves

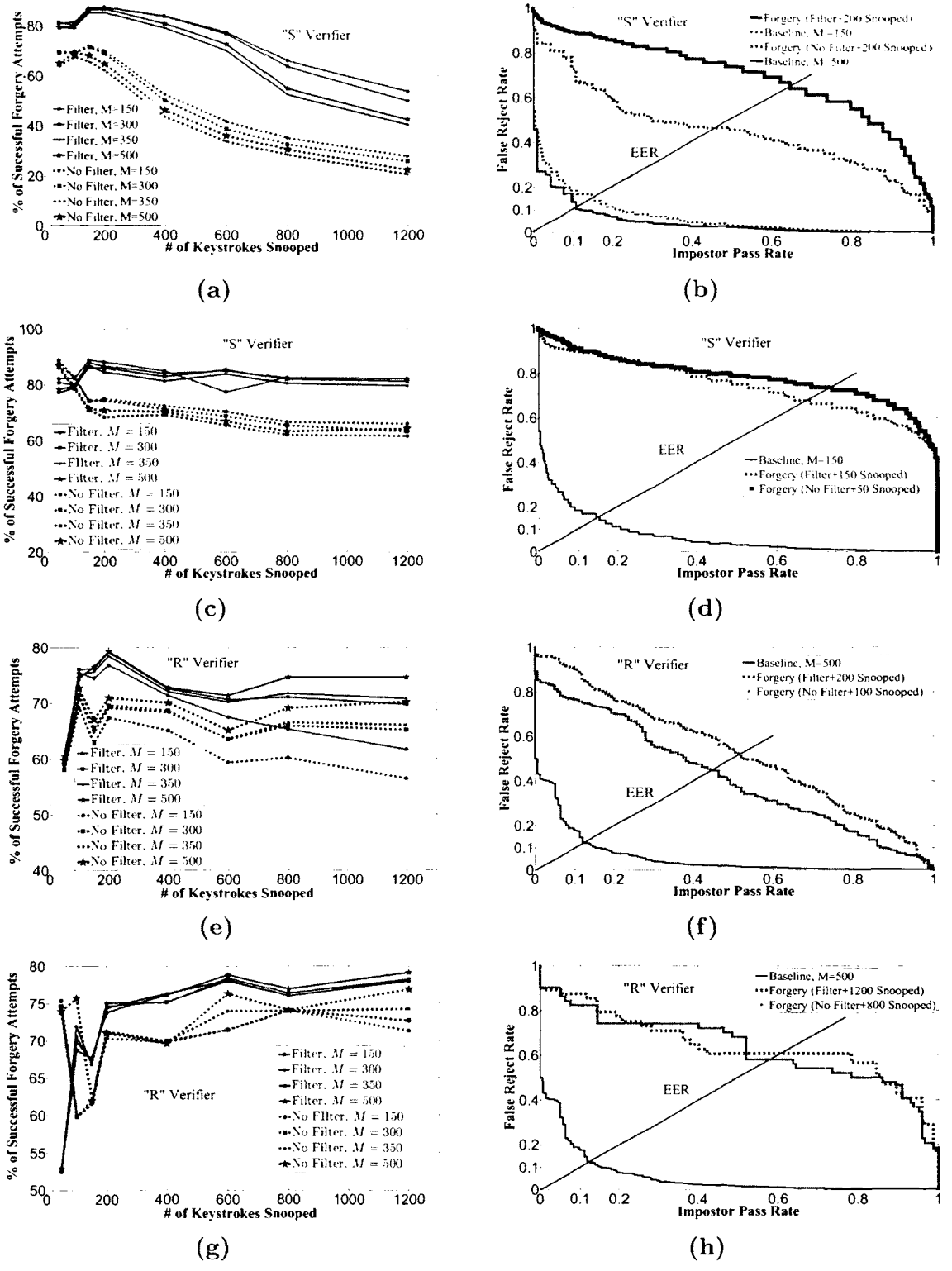


Figure 4.2: (a, c, e, g): Percentage of successful forged attempts with threshold T_1; (b, d, f, h): Comparison of EERs generated with zero-effort impostors (baseline) and forged verification attempts generated by estimating the means and standard deviations of snooped digraph latencies (b, d), and by estimating the means of the top 10 frequent snooped digraphs (f, h).

obtained with forged attempts (with 200 and 150 snooped keystrokes). The DET curves clearly show that forged attempts considerably increase the baseline EERs.

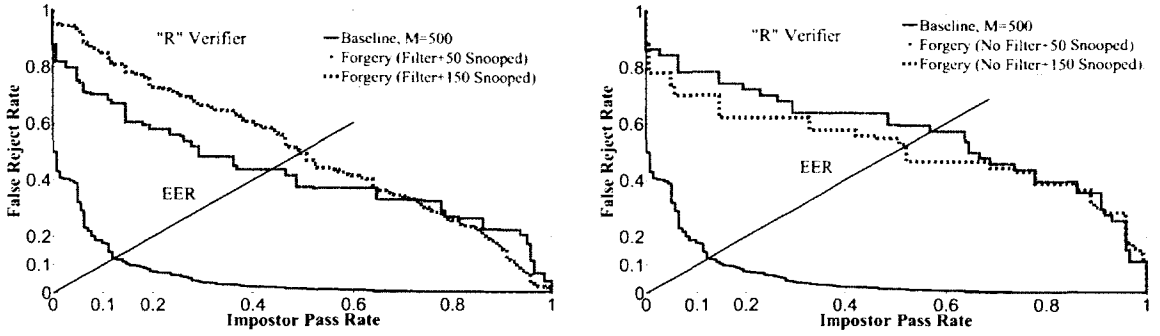


Figure 4.3: Comparison of EERs of "R" verifier with forged verifications attempts generated with 50 and 150 snooped keystroke when $M = 500$.

In "S" verifier, the percentage of successful forgeries increased as the number of snooped keystrokes increased from 50 to 150 and then began to decrease from 200 to 1200. Though less pronounced, the same behavior was observed with "R." The observation that snooping more keystroke information makes the forgery attack less effective is counter-intuitive. The reason behind this phenomenon is that the frequency of occurrence of English digraphs follow heavy-tailed (Pareto-like) distributions, implying that only a few digraphs occur most of the time. In fact, [33] showed that 25 digraphs make about third of all printed English text. As more keystrokes are snooped, the frequencies of only a few digraphs are sufficient enough to estimate the mean/standard deviations, and for the remaining digraphs, the frequencies are too low to estimate the means/standard deviations correctly. Therefore, increasing the number of snooped keystrokes increases the number of badly estimated means/standard deviations used to forge a sample. To support our argument, we created forged samples

Table 4.2: IPRs of verifiers "R" and "S" for thresholds T_1 and T_2.

M	T_1 "R"	T_2 "R"	T_1 "S"	T_2 "S"
150	0.118167	0.138081	0.152123	0.138153
300	0.109109	0.128045	0.124451	0.124451
350	0.115047	0.127630	0.080491	0.112174
500	0.102082	0.123187	0.085271	0.099889

using means estimated from the 10 most frequently occurring digraphs (and set the standard deviation to zero, i.e., created the forgeries using only the mean digraph latencies). Plots 4.2(c) and 4.2(g) show a clear improvement in the success rates of forgery attempts when the number of snooped keystrokes increase from 200-1200. We also compare the baseline DET curves obtained with zero-effort impostors when $M = 500$ and the DET curves obtained with forged attempts created using only the means of the top 10 most occurring snooped digraphs (with 1200 and 800) snooped keystrokes. The DET curves clearly show that forged attempts considerably increase the baseline EERs.

In Figure 4.3(a) we compare the baseline (zero-effort) EERs of "R" verifier ($M = 500$) with EERs of forged impostor attempts created by estimating the means and standard deviations of digraph latencies (plot 4.3(a)) and with forged attempts created by estimating the means of the top 10 most frequent digraphs (plot 4.3(b)). The plots confirm that forged attempts created by estimating the means of the top 10 most frequent digraphs have higher EERs. Additionally, we also plotted forgery success rates with the threshold T_2 (not shown due to space constraints). The plots with T_2 were similar to Figure 4.2 and offered no new insights.

Table 4.3: EERs generated for 150 users, with Set II as the training data and Set III as the verification data. The lowest EERs in each verifier group are marked in **bold**.

M	R			S			A			Fusion (F)		
	KI	KP	KH	KI	KP	KH	KI	KP	KH	Lowest EER_F	Avg. EER_F	STD_F
20	0.235	0.299	0.22	0.227	0.251	0.345	0.336	0.265	0.329	0.149	0.169	0.0165
40	0.185	0.246	0.155	0.186	0.212	0.325	0.302	0.232	0.306	0.105	0.125	0.0189
60	0.16	0.224	0.143	0.165	0.191	0.317	0.293	0.218	0.303	0.089	0.113	0.0196
80	0.153	0.206	0.132	0.158	0.177	0.311	0.274	0.209	0.294	0.08	0.105	0.0198
100	0.143	0.196	0.12	0.151	0.169	0.31	0.266	0.2	0.288	0.072	0.098	0.0217
120	0.141	0.187	0.117	0.145	0.165	0.307	0.262	0.196	0.286	0.069	0.096	0.0216
150	0.131	0.171	0.11	0.137	0.156	0.306	0.252	0.195	0.285	0.067	0.092	0.0211
300	0.123	0.155	0.105	**0.124**	0.141	0.307	0.246	0.191	0.275	0.056	0.083	0.0209
350	0.117	0.160	0.104	0.131	0.131	0.308	0.248	0.192	0.273	0.043	0.069	0.0259
500	0.116	0.155	**0.103**	0.132	0.126	0.288	0.247	**0.187**	0.258	0.042	0.086	0.0277
750	0.124	0.135	0.105	0.132	0.139	0.282	0.234	0.206	0.264	**0.032**	0.084	0.0189

Table 4.4: EERs generated for 200 users. Set II was used for training, Set III was used for generating genuine attempts, and Set IV for generating impostor attempts.

M	R			S			A			Fusion (F)		
	KI	KP	KH	KI	KP	KH	KI	KP	KH	Lowest EER_F	Avg. EER_F	STD_F
20	0.232	0.296	0.2	0.218	0.239	0.34	0.331	0.255	0.315	0.141	0.169	0.0165
40	0.179	0.243	0.146	0.178	0.201	0.321	0.296	0.222	0.293	0.099	0.122	0.0186
60	0.155	0.221	0.123	0.158	0.18	0.313	0.287	0.207	0.289	0.084	0.107	0.0194
80	0.147	0.203	0.113	0.15	0.166	0.307	0.267	0.2	0.279	0.076	0.099	0.0196
100	0.135	0.192	0.106	0.143	0.158	0.306	0.259	0.192	0.276	0.068	0.094	0.0212
120	0.133	0.184	0.101	0.137	0.156	0.303	0.255	0.189	0.276	0.067	0.092	0.0213
150	0.123	0.167	0.097	0.13	0.147	0.304	0.244	0.186	0.272	0.063	0.088	0.0209
300	0.115	0.149	0.088	0.116	0.134	0.302	0.236	0.183	0.263	0.048	0.076	0.0211
350	0.112	0.152	0.087	0.124	0.122	0.302	0.249	0.18	0.26	0.044	0.067	0.0248
500	0.112	0.148	0.096	0.121	0.123	0.288	0.237	0.179	0.246	0.041	0.081	0.0268
750	0.116	0.128	0.1	0.12	0.125	0.279	0.223	0.195	0.253	0.03	0.08	0.0185

4.2 Sample-level Attack Experiments

4.2.1 Baseline (Zero-effort Impostor Attack) Results and Analysis

Table 4.3 shows the EERs[1] of the nine verifier-template combinations and the EERs of "F" verifier. To generate EERs in Table 4.3, we used Set II to build the users templates and Set III to obtain genuine and zero-effort impostor scores. In Table 4.3, "F" verifier has three columns: 1) "Lowest EER_F" gives the lowest, 2) "Avg. EER_F" gives the average, and 3) "STD_F" gives the standard deviation of EERs obtained with 126 different weight combinations.

[1]Equal error rate (EER) or crossover error rate is the point where false reject rate (FRR) and impostor pass rate (IPR) curves intersect. To plot the FRR and IPR curves, we calculated a series of false reject rates and impostor pass rates by varying the verification threshold from 0 to 1 in increments of 0.001.

Table 4.4 is similar to Table 4.3 except we used Set IV instead of Set III to generate impostor scores. (We used the same genuine scores to generate EERs in both Table 4.3 and Table 4.4). Our observations follow.

Observation 1: For all M values, the fusion ("F") verifier outperformed individual ("R," "S," and "A") verifiers. See "Lowest EER_F" and "Avg. EER_F" columns under "F" verifier in Tables 4.3 and 4.4.

Observation 2: Irrespective of the verifier, lowest EERs were achieved at higher M values (*e.g.*, 300, 350, 500, and 750) and a trade-off exists between M and EERs (*i.e.*, as M increases, we can expect EERs to decline).

Observation 3: We computed the absolute difference between the EER in each cell in Table 4.3 and the same cell in Table 4.4 (excluding cells of columns "Avg. EER_F" and "STD_F"). The average of absolute differences between EERs in Table 4.3 and Table 4.4 is 0.00998. This means the EER, on average, changes by 0.00998 if the impostors were from Set IV instead of Set III. *This shows that EERs in Tables 4.3 and 4.4, though obtained from two different impostor populations, are not quite different.*

In Table 4.5, we give the total number of genuine and impostor verification attempts extracted from Set III and Set IV to generate EERs in Tables 4.3 and 4.4. In Table 4.5, we also give the average number of keystrokes in a verification attempt (includes genuine and zero-effort impostor) and the average time in seconds taken by the users to type a verification attempt.

Reason for not considering matching pairs beyond 750: Note from Table 4.5 that it took 14.83 seconds of typing time to generate a verification attempt

Table 4.5: "Tot. Impostor Attempts" and the "Tot. Genuine Attempts" columns give the total number of impostor and genuine verification attempts used to calculate EERs in Tables 4.3 and 4.4. "Avg. Typing Time per Attempt" column gives the average time taken to type a verification attempt (in seconds). "Avg. # of Keystrokes per Attempt" column gives the average number of keystrokes in a verification attempt.

M	Tot. Impostor Attempts	Tot. Genuine Attempts	Avg. Typing Time (in sec.) per Attempt	Avg. # of Keystrokes per Attempt
20	10950540	32630	14.83	54.65
40	5439610	16218	29.81	109.53
60	3596320	10715	44.65	163.98
80	2673265	7961	59.47	218.06
100	2119001	6320	74.27	272.03
120	1749125	5218	89.07	326.11
150	1379683	4114	111.24	406.72
300	640836	1918	221.11	805.63
350	530810	1579	257.62	937.49
500	342178	1018	364.24	1328.73
750	189141	564	547.56	2000.15

when $M = 20$ and 547.56 seconds when $M = 750$. Though all the verifiers achieved lower EERs when $M = 750$, it is impractical for a continuous verification system to use 750 matching pairs, because for each verification attempt, the verifier would have to wait for nearly 10 minutes. This is the reason why we did not consider beyond 750 matching pairs in our experiments.

4.2.2 Snoop-forge-replay Attack Parameters and Configurations

We considered four snoop-forge-replay attack parameters for sample-level attacks. They are:

1) Length of Snooped Text: is the number of keystrokes for which the attacker steals hold and interval latencies from a victim. Depending on various factors, including attacker's intent and victim's availability, the attacker can steal few or many

keystrokes. We experimented with snooped text of length 20, 50, 100, 200, 600, and 1200 to see how this parameter impacts the attack performance.

For extracting N snooped keystrokes from user U_i, we used the first N characters from U_i's typing sample in Set I. For example, when we snooped $N = 100$ characters for user U_i, we used the first 100 characters from U_i's sample in Set I.

2) Gaussian Perturbation of Snooped Latencies: Keystroke dynamics is a behavioral trait, so it is highly unlikely that two latencies of the same key (for example, two hold latencies of "a") will be exactly equal, even if "a" was typed in rapid succession. From the snooped keystrokes, assume an attacker learns that the average key hold latency of "a" is 150ms. If the attacker creates a forgery that has 150ms for every occurrence of "a," then this artifact alone can expose the forgery.

We solved the problem by adding Gaussian noise (zero mean and three standard deviations) to perturb the latency values in a forgery. We chose Gaussian because its parameters (mean and standard deviation) are easy to estimate and most programming languages can generate a Gaussian variate. However, an attacker can also choose a different perturbation model (*e.g.,* adding noise from uniform distribution), as long as the perturbation does not distort the latencies too much. We performed experiments without and with Gaussian perturbation.

3) Filtering Outliers: From the snooped latencies, the attacker can choose to remove outliers. We performed experiments with and without filtering outliers. When we filtered outliers, we discarded any latency greater than or equal to 300 milliseconds.

Table 4.6: Twenty-four attack configurations obtained with different parameter settings.

Configuration Number	Length of Snooped Text	Gaussian Perturbation	Filtering Outliers	Min. Freq. of Occurrence
1	20	YES	YES	1
2	50	YES	YES	1
3	100	YES	YES	1
4	200	YES	YES	1
5	600	YES	YES	1
6	1200	YES	YES	1
7	20	YES	NO	1
8	50	YES	NO	1
9	100	YES	NO	1
10	200	YES	NO	1
11	600	YES	NO	1
12	1200	YES	NO	1
13	20	NO	YES	1
14	50	NO	YES	1
15	100	NO	YES	1
16	200	NO	YES	1
17	600	NO	YES	2
18	1200	NO	YES	3
19	20	NO	NO	1
20	50	NO	NO	1
21	100	NO	NO	1
22	200	NO	NO	1
23	600	NO	NO	2
24	1200	NO	NO	3

4) Minimum Frequency of Digraphs in the Snooped Text: In the snooped text, if a latency (*e.g.*, key hold of "a") appeared multiple times, we used its average in the forgery. To improve the forgery, the attacker can choose to use the snooped digraphs whose average latency was computed with at least k repeats. For long snooped text lengths, *i.e.*, 600 and 1200, we chose k to be 2 and 3, respectively. However, for shorter lengths (20, 50, 100, and 200), we considered all the digraphs regardless of how many times they repeated; otherwise, we were left with too few digraphs to create a forgery.

Using the above attack parameters, we created 24 attack configurations. (In Table 4.6, we list the 24 attack configurations with their parameter values.) We

Table 4.7: Average number of snoop-forge-replay attacks generated **per user**.

M	Avg. No. of "Snoop-forge-replay" Attacks per User			
	KI	**KP**	**KH**	**F**
20	5594.98	5451.27	5325.65	4528.12
40	3448.91	3377.06	3149.48	2599.54
60	3105.98	3080.52	2946.57	2438.02
80	2839.48	2674.07	2639.32	2235.75
100	2401.59	2261.081	2233.64	1916.44
120	2012.78	1895.34	1872.46	1603.36
150	1526.72	1517.16	1498.85	1284.11
300	805.31	758.38	749.21	651.23
350	690.19	649.92	642.07	549.93
500	483.01	454.82	449.35	381.67
750	321.85	303.05	299.38	254.53

experimented with 24 attack configurations, 11 matching pairs, and 10 different verifier-template pairs. This resulted in $24 \times 11 \times 10 = 2640$ attack experiments.

Table 4.7 gives the number of snoop-forge-replay attacks we generated against each user. Values in Table 4.7 represent averages calculated from all attacks generated by the 24 attack configurations and running Algorithm 1 for 24 hours.

4.2.3 Effectiveness of the Attacks

Comparison of attack and baseline using error rate plots: In Figure 4.4, we compare the baseline (zero-effort) impostor pass rates with snoop-forge-replay attack pass rates. Panels (a), (b), (c), and (d) in Figure 4.4 show the error rate plots for "R," "S," "A," and "F" verifiers. In each panel, the two baseline IPR curves correspond to zero-effort impostor attacks with Set III and Set IV, respectively. The 24 attack IPR curves correspond to 24 attack configurations.

Showing error rates for all verifier settings is not practical because we experimented with 11 matching pairs and 10 verifier-template combinations, which gives 110

settings. So, we chose to show plots corresponding to the setting in which a verifier had achieved its lowest baseline EER. (We highlighted the lowest baseline EERs in **bold** in Table 4.3).

Table 4.8 gives the maximum, minimum, and average attack EERs and baseline EERs corresponding to the panels (a)–(d) in Figure 4.4. Table 4.8 shows that the attack IPRs (in Figure 4.4) *markedly increased* the EERs for verifiers which had the lowest EERs in our baseline experiments. The results in Figure 4.4 and Table 4.8 also illustrate the *wide discrepancy* between the snoop-forge-replay attack EERs and baseline EERs.

Comparison of attack and baseline using EER plots: Here we summarize how the 24 attack configurations perform against 10 verifier-template combinations and 11 matching pair settings. In Figures 4.5 and 4.6, panels correspond to 10 verifier-template combinations. In each panel, we show the maximum, minimum, average, and standard deviation (error bars) of attack EERs and baseline EERs for 11 matching pair (M) settings. The maximum, minimum, average, and standard deviations of attack EERs were computed from EERs corresponding to 24 attack configurations. From the panels, we observe the following:

1) The "Maximum Attack EER" curves are remarkably higher than zero-effort "Baseline EER" curves (see Table 4.9 for percentage increase in attack EERs over the baseline). This shows that all 11 matching pairs and 10 verifier-template combinations were vulnerable to the snoop-forge-replay attack;

2) The fusion "F" verifier (see Figure 4.6, panel *(b)*), which had the lowest EERs in our baseline experiments has the highest maximum, minimum, and average

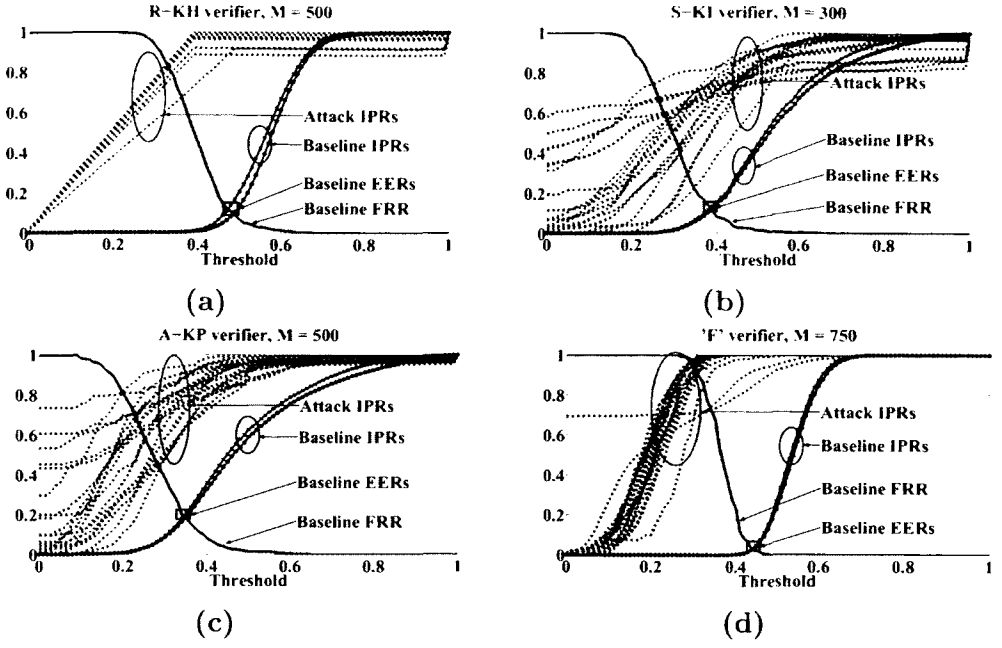


Figure 4.4: False reject rate (Baseline FRR), zero-effort impostor pass rates (Baseline IPRs) *(highlighted by the smaller circles)*, and 24 snoop-forge-replay attack IPR curves *(highlighted by the large circles)* achieved with "R" *(a)*, "S" *(b)*, "A" *(c)* verifiers paired with KH, KI, and KP templates respectively and "F" *(d)* verifier. In each plot, the Baseline EERs (crossover points between Baseline FRR and Baseline IPR curves) are marked by a box.

attack EERs. This demonstrates that the best performing verifier under zero-effort attacks could turn out to be the most vulnerable verifier under a non-zero effort attack;

3) The maximum and minimum attack EER curves and error bars indicate that some attack configurations are more effective than the others. In Section 4.2.4, we discuss which attack configurations are more effective; and

4) The "Minimum Attack EER" curves show that, even for the worst performing attack configurations, the "R," "A," and "F" verifiers had considerably high attack EERs compared to their baseline EERs.

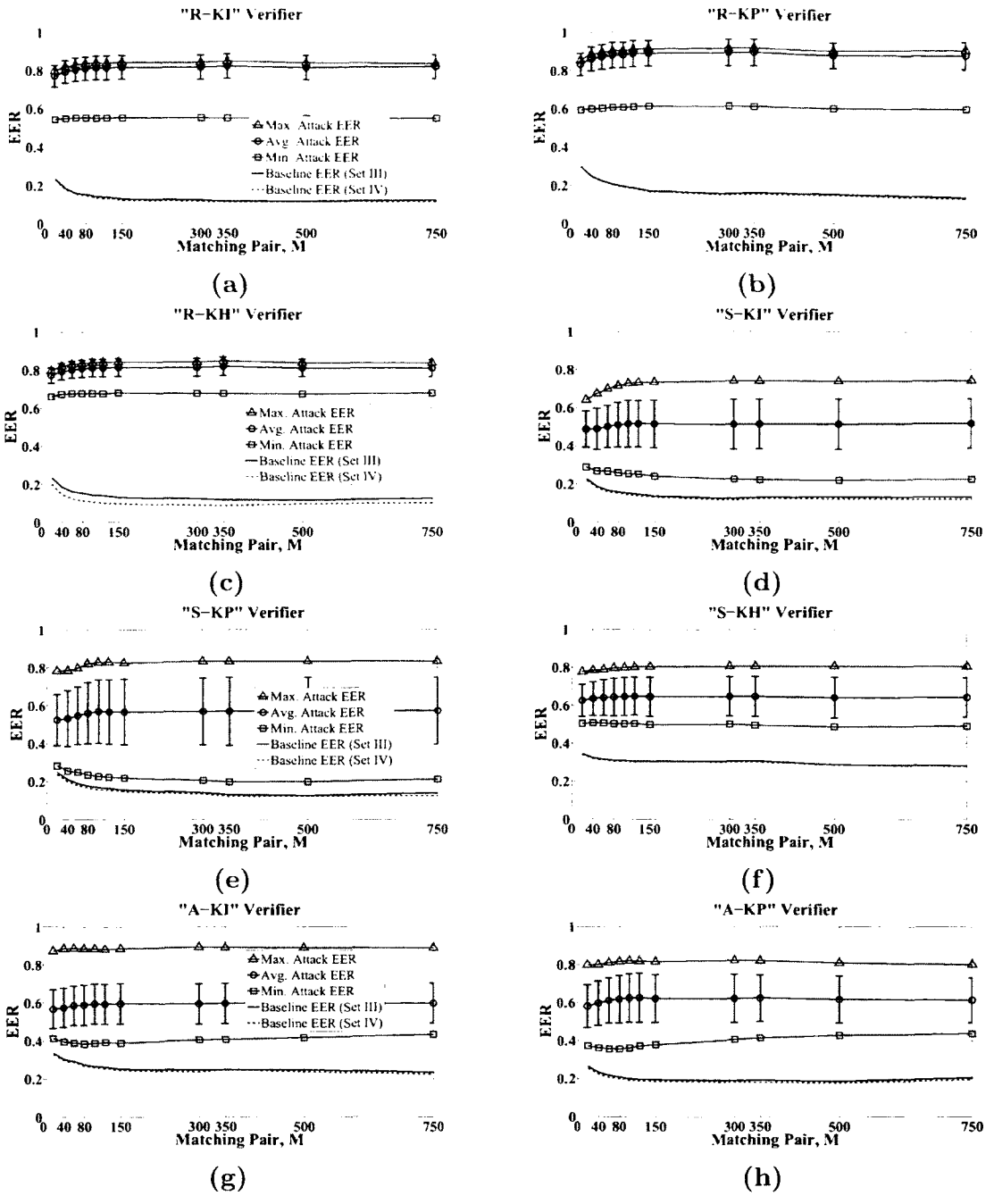
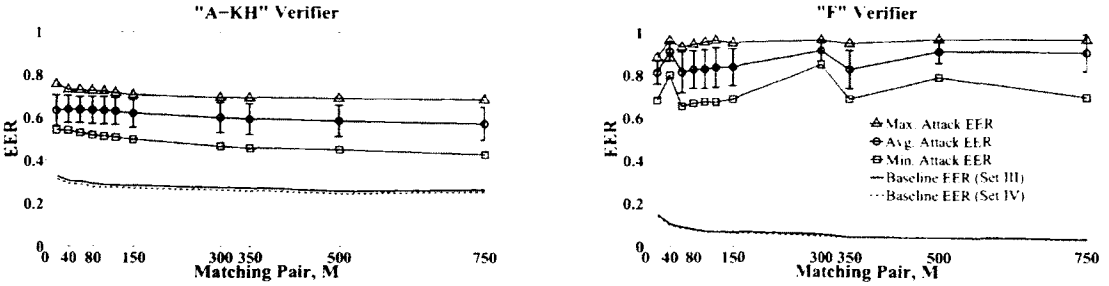


Figure 4.5: Maximum ("Max. Attack EER" curves), average ("Avg. Attack EER" curves), minimum ("Min. Attack EER" curves), and standard deviations (Error bars) of attack EERs achieved from 24 snoop-forge-replay attack configurations against "R" *(a - c)*, "S" *(d - f)*, "A" *(g - h)* verifiers and KI, KP, and KH templates. Two Baseline EER curves "Baseline EER (Set III)" and "Baseline EER (Set IV)" represent EERs from Tables 4.3 and 4.4. *("Baseline EER (Set III)" and "Baseline EER (Set IV)" curves are overlapping in most panels.)* Legends are the same for panels *(a)* through *(j)*.



(a) (b)

Figure 4.6: Maximum ("Max. Attack EER" curves), average ("Avg. Attack EER" curves), minimum ("Min. Attack EER" curves), and standard deviations (Error bars) of attack EERs achieved from 24 snoop-forge-replay attack configurations against "A" *(a)* verifier and KH template and "F" *(b)* verifier. Two Baseline EER curves "Baseline EER (Set III)" and "Baseline EER (Set IV)" represent EERs from Tables 4.3 and 4.4. *("Baseline EER (Set III)" and "Baseline EER (Set IV)" curves are overlapping in most panels.)* Legends are the same for panels *(a)* through *(j)*.

4.2.4 Performance Analysis of Attack Parameters

To observe the performance of attack parameters, in Figure 4.7, we give the attack EERs of "R" (panels a and b), "S" (panels c and d), "A" (panels e and f), and "F" (panels g and h) verifiers configured with 750 matching pairs and different attack parameters. The EERs with the remaining matching pairs (40, 60, 80, 100, 120, 150, 300, 350, and 500) behave the same way as the panels in Figure 4.7. See Appendix B for the EER plots with the remaining matching pairs.

In Figure 4.7, panels a, c, e, and g, "Filtering + Gaussian" (solid lines) correspond to configurations when the outliers were *filtered*, latencies were *perturbed* with Gaussian, and *all* snooped latencies were used to compute forgeries, irrespective of their frequency of occurrence. "No Filtering + Gaussian" (dashed lines) correspond to similar settings except outliers were *not filtered*. In panels b, d, f, and h, "Filtering + Min. Frequency" (solid lines) correspond to configurations when the outliers were *filtered*, latencies were *not perturbed* with Gaussian, and only the latencies that

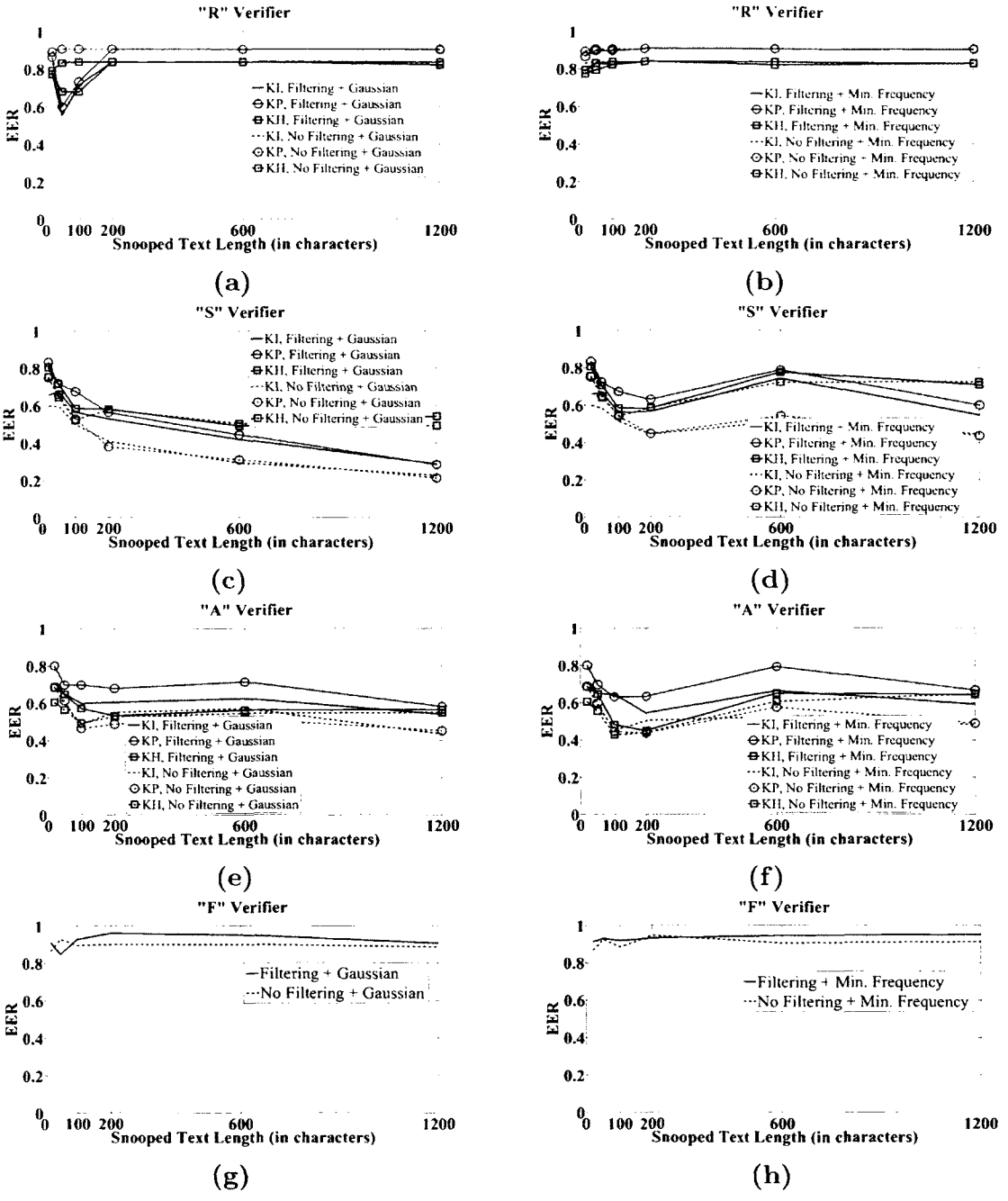


Figure 4.7: Attack EERs under different attack configurations with "R" ($a - b$), "S" ($c - d$), "A" ($e - f$), and "F" ($g - h$) verifiers. EERs are computed using $M = 750$. Solid lines represent attack EERs when the outliers are *filtered* and the dashed lines represent attack EERs when the outliers are *not filtered*.

Table 4.8: Attack EERs of the top performing verifier settings in baseline experiments.

Verifier	Baseline EERs (Set III & IV)	Maximum Attack EER	Minimum Attack EER	Average Attack EER
R-KH, $M = 500$	0.116, 0.096	0.837	0.673	0.81
S-KI, $M = 300$	0.124, 0.116	0.742	0.224	0.514
A-KP, $M = 500$	0.187, 0.179	0.811	0.429	0.618
F, $M = 750$	0.032, 0.03	0.965	0.679	0.905

occurred *more than once* in the snooped text were used to compute forged latencies. "No Filtering + Min. Frequency" (dashed lines) correspond to similar settings except outliers were *not filtered*. Our observations follow.

1) Forgeries using short snooped text of 20 to 100 characters achieved *very high* EERs ranging from lowest 0.43 (with "A-KH" in Figure 4.7*(f)*) to highest 0.92 (with "F" in Figure 4.7*(h)*). However, the limitation with short snooped text is that it produces forgeries in which few characters repeat many times. Consequently, the forged text may contain misspelled words, linguistically meaningless text, and grammatical violations. However, given that current keystroke-based continuous verification systems do not impose any checks on the text typed by the user, the forgeries created with short snooped text are still *effective*.

In panels *(c)* and *(e)* in Figure 4.7, note that as the snooped length increases, the attacks become *less effective* against "S" and "A" verifiers. We explain why this happens in Section 4.2.6.

2) Overall, we achieved higher attack EERs, ranging from lowest 0.43 (with "A-KH" in Figure 4.7*(f)*) to highest 0.965 (with "F" in Figure 4.7*(h)*), when the outliers in the snooped text were *filtered*, latencies were *not perturbed* with Gaussian noise,

Table 4.9: Minimum to maximum percentage increase in attack EERs over baseline EERs across all the matching pairs.

	Max. Attack EER	Avg. Attack EER	Min. Attack EER
Baseline (Set III)	125.5% to% 2915.62%	69.33% to 2730.55	12.84% to 2075.41%
Baseline (Set IV)	128.8% to 3116.67%	71.89% to 2919.26%	18.51% to 2220.44%

and only the latencies that occurred *more than once* in the snooped text were used to compute forged latencies. Attack EERs corresponding to these parameters are shown as solid curves in panels *b*, *d*, *f*, and *h* in Figure 4.7.

In comparison, we achieved lower attack EERs, ranging from the lowest 0.21 (with "S-KP" in Figure 4.7(c)) to highest 0.932 (with "F" in Figure 4.7(g)), when the outliers were *not filtered*, latencies were *perturbed* with Gaussian, and *all* snooped latencies were used to compute forgeries, irrespective of their frequency of occurrence. Attack EERs corresponding to these parameters are shown as dashed curves in panels *a*, *c*, *e*, and *g* in Figure 4.7.

4.2.5 Analysis of Attacks Against the "R" Verifier

In the solid curves of Figure 4.7(a), note the drop in EERs when the snooped text lengths are 20, 50, and 100. When we used, 1) 20, 50, and 100 snooped text lengths, 2) Gaussian perturbation, and 3) outlier filtering, we achieved lower attack EERs with "R" verifier. This occurred because the normalized disorder score (see "R" verifier, Section 3.6, Page 34) is sensitive to the size of A_{train} and A_{test}. We explain with the following example.

Assume A_{train} and A_{test} contain five digraphs. If there is one mismatch between A_{train} and A_{test}, it means 40% or two out of the five digraphs do not have the same rank. This gives a normalized disorder score of 0.16667. Now, assume A_{train} and A_{test} contain 20 digraphs. One mismatch between A_{train} and A_{test} leads to having two out of 20 (or 10%) digraphs that do not have the same rank. This gives a normalized disorder score of 0.01. So, even if A_{train} and A_{test} have the same number of mismatches, the normalized disorder between A_{train} and A_{test} is higher when the array sizes are lower.

Filtering outliers from snooped texts of length 20, 50, and 100 decreased the size of A_{train} and A_{test} arrays. Gaussian perturbation further increased the disorder score. The two parameters together increased the normalized disorder score between the template and the forgery and thus lowered the attack EERs. However, EERs for snooped text of lengths 200, 600, and 1200 were high in spite of adding Gaussian perturbation because the arrays were large even after outlier filtering.

When the outliers were not filtered, the size of the arrays were large for all snooped text lengths. So the attack EERs were high for all snooped text lengths (see dashed curves in Figures 4.7(a) and 4.7(b)). Therefore, except when we performed both Gaussian perturbation and outlier filtering with snooped text of length 20, 50, and 100, the attack EERs for "R" verifier were high for all configurations.

4.2.6 Analysis of Attacks Against "S" and "A" Verifiers

In Figures 4.7(c) and 4.7(e), we observe that the attack EERs decreased as the snooped text length increased. The reason why EERs decreased when snooped text length increased is related to the distribution of digraphs in English text. In Figure 4.8, we show the probability distribution of the digraphs. To plot the distribution,

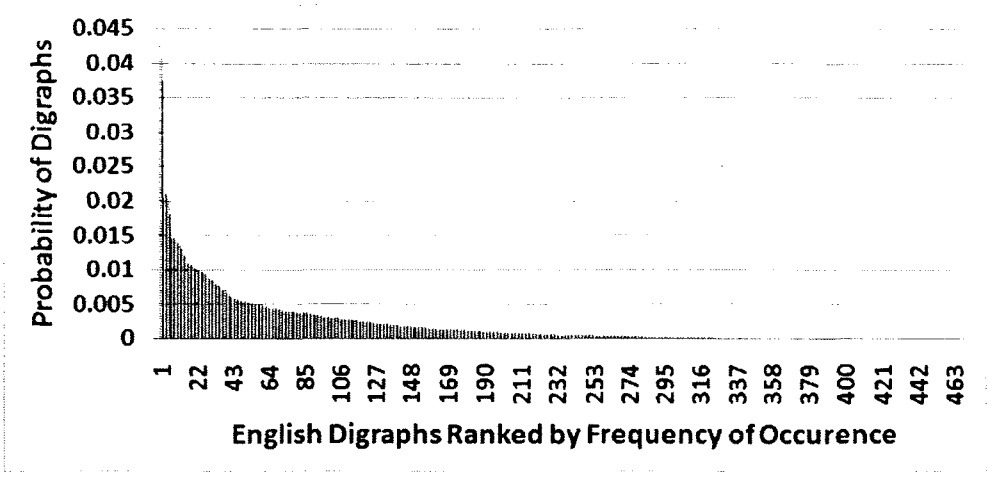


Figure 4.8: Probability distribution of English digraphs. Probabilities were computed from the digraph frequencies given in [44].

we used digraph frequencies in [44], which calculated the frequencies by considering 997,380 digraph instances. In Figure 4.8, notice that the first few digraphs cover a major portion of the probability distribution compared with the rest of the digraphs. In fact, the 40 most frequent digraphs account for 50.71% of all digraph instances. So, when the snooped text is long (600 and 1200 characters), the majority of the digraphs occur only once. For example, from the distribution shown in Figure 4.8, if the snooped text has 600 digraphs, out of the 676 possible English digraphs, we can expect only 94 digraphs to occur more than once in the snooped text and the remaining 582 digraphs either occur once or do not occur at all.

For the digraphs that occurred only once in the snooped text, we used a single snooped latency value in the forgery. This latency value could have been an inaccurate estimate of the "true" latency value. During the attack, when we replayed single latency values, the majority of the digraphs were considered "invalid" by the "S" and "A" verifiers. This lowered the attack EERs.

We mitigated the problem by replaying only those digraph latencies that occurred at least two or three times in the snooped text. This gave us a better estimate of the mean of the victim user's latencies, and as a result, the forgery had a better chance of being considered as "valid." In the right side panels (*b*, *d*, *f*, and *h*) of Figure 4.7, with 600 characters snooped text, we used latencies of those digraphs that occurred at least twice and with 1200 characters, we used digraphs that occurred at least thrice. Consequently, from Figures 4.7(d) and 4.7(f), we see that the attack EERs are high at 600 and 1200 snooped text lengths.

Attack EERs increased when we filtered the outliers in the snooped text. In Figures 4.7(c)–4.7(f), the solid curves (which represent filtering outliers) are above their corresponding dashed curves (which represent no outlier filtering). By filtering the outliers we were able to forge latencies that were closer to the victim user's latencies, and therefore, were able to increase the attack pass rates.

4.2.7 Analysis of Attacks Against the "F" Verifier

The "F" verifier, which had the lowest EERs against zero-effort impostor attacks had surprisingly high attack EERs. From Figures 4.7(g) and 4.7(h), we observe the following: as in "R," the EERs for "F" verifier were high for all attack configurations. A contributing factor for this is that the "R" verifier was weighted more in the "F" verifier.

In Figures 4.7(g) and 4.7(h), the solid curves are over the dashed curves, which shows that filtering the outliers increased the attack EERs. In Figure 4.7(g), at a few points the dashed curves are over the solid curves when the snooped text length is

short. This indicates the influence of "R" verifier, which had lower attack EERs for short snooped text with Gaussian perturbation and outlier filtering.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this dissertation, we presented a new attack called "snoop-forge-replay" attack that synthesizes keystroke forgeries using timing information stolen from victim users. Our results from feature-level and 2640 sample-level attack experiments (involving 150 users, four state-of-the-art continuous verifiers, three types of keystroke latencies, and 24 attack configurations) reveal that snoop-forge-replay attacks are very effective in increasing EERs. With 20 to 1200 snooped keystrokes, the average sample-level snoop-forge-replay attack EERs were between 0.487 and 0.912. In comparison, the baseline EERs with zero-effort impostor attacks were between 0.03 and 0.285 (*i.e.*, the attack increased EERs from between 69.33% to 2730.55%). Our results additionally show that effective keystroke forgeries can be created with a) as low as 20 to 100 characters of snooped text and b) old legacy keystroke timing information.

The main reason for the success of snoop-forge-replay attack is that keystroke-based continuous verification methods solely rely on users' latency information, which can be easily forged, as demonstrated in this dissertation. We opine that by integrating text-based and language-based traits into the verification process such as – 1) the rate at which a user misspells words or repeats letters, 2) type of words for which the user has latency outliers, 3) how the user revises text *i.e.*, revision pattern, and so

on, the impact of the attack can be mitigated. In our future work, we will pursue the problem of designing keystroke based verification systems that are resilient to snoop-forge-replay attacks.

APPENDIX A

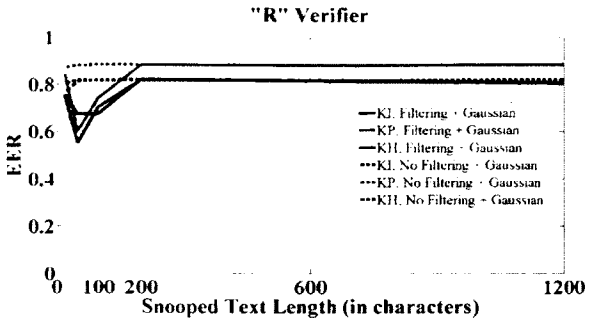
ADDRESSES OF WEB PAGES USED AS "DUMMY TEXT"

Web addresses of 20 Wikipedia pages used in our "dummy text" file

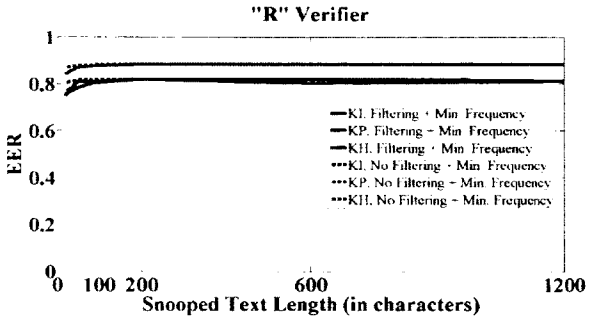
1. *en.wikipedia.org/wiki/History_of_United_States*
2. *en.wikipedia.org/wiki/World_War_II*
3. *en.wikipedia.org/wiki/Air_warfare_of_World_War_II*
4. *en.wikipedia.org/wiki/Effects_of_World_War_II*
5. *en.wikipedia.org/wiki/United_Nations*
6. *en.wikipedia.org/wiki/World_War_I*
7. *en.wikipedia.org/wiki/Causes_of_World_War_I*
8. *en.wikipedia.org/wiki/Cold_War*
9. *en.wikipedia.org/wiki/Great_Pyramid_of_Giza*
10. *en.wikipedia.org/wiki/Stonehenge*
11. *en.wikipedia.org/wiki/Colosseum*
12. *en.wikipedia.org/wiki/Great_Wall_of_China*
13. *en.wikipedia.org/wiki/War_on_Terror*
14. *en.wikipedia.org/wiki/Gulf_War*
15. *en.wikipedia.org/wiki/Vietnam_War*
16. *en.wikipedia.org/wiki/Grand_Canyon*
17. *en.wikipedia.org/wiki/Christopher_Columbus*
18. *en.wikipedia.org/wiki/Albert_Einstein*
19. *en.wikipedia.org/wiki/Isaac_Newton*
20. *en.wikipedia.org/wiki/NASA*

APPENDIX B

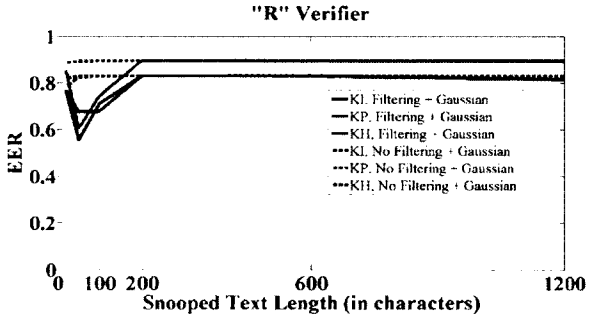
EER PLOTS UNDER DIFFERENT ATTACK CONFIGURATIONS



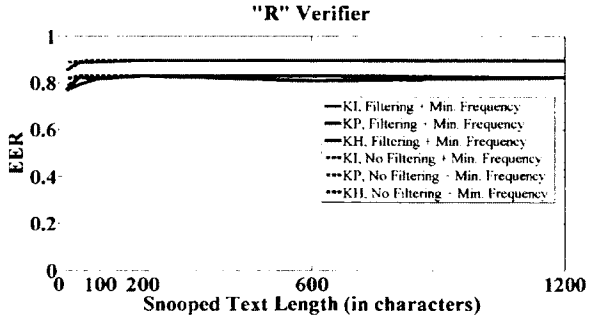
(a)



(b)



(c)



(d)

Figure B.1: Comparison of attack EERs using attack configurations 1–12 (plots *a, c*) and attack configurations 13–24 (plots *b, d*) for "R" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 40$, plots *c* and *d* correspond to $M = 60$.

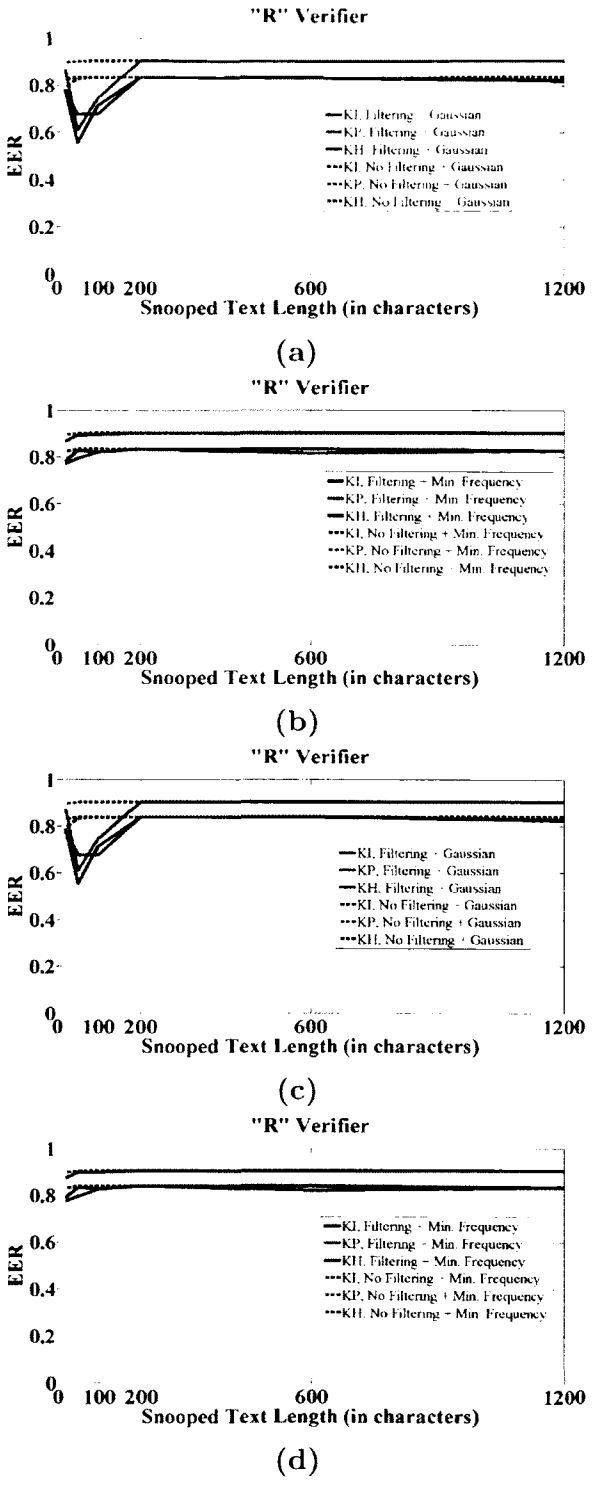


Figure B.2: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "R" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 80$, plots *c* and *d* correspond to $M = 100$.

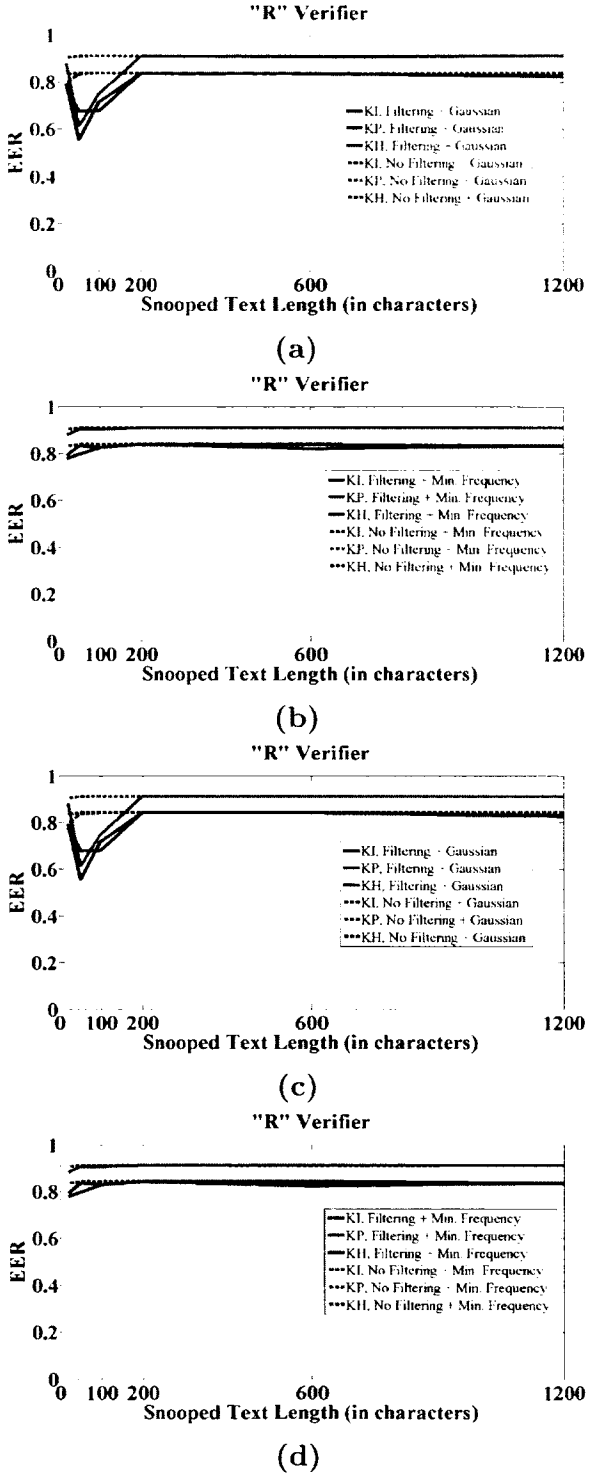
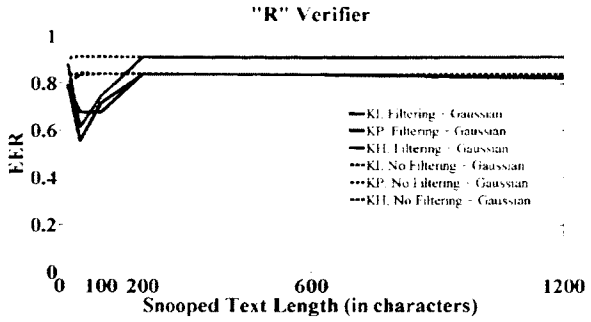
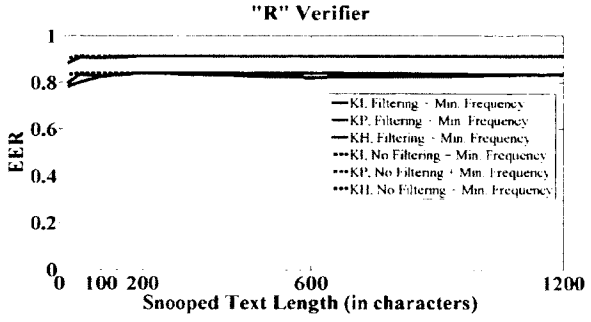


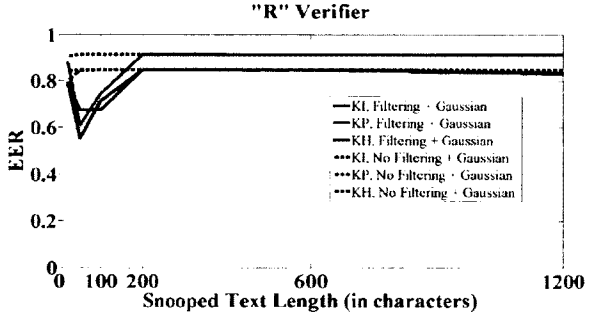
Figure B.3: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "R" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 120$, plots *c* and *d* correspond to $M = 150$.



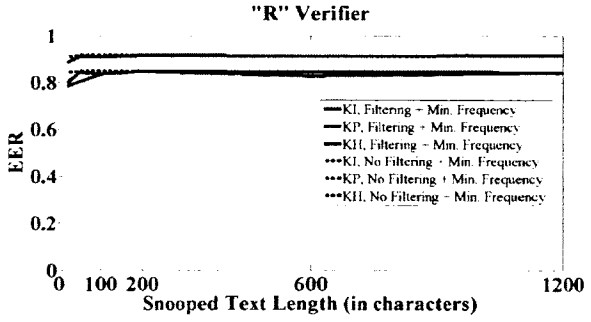
(a)



(b)

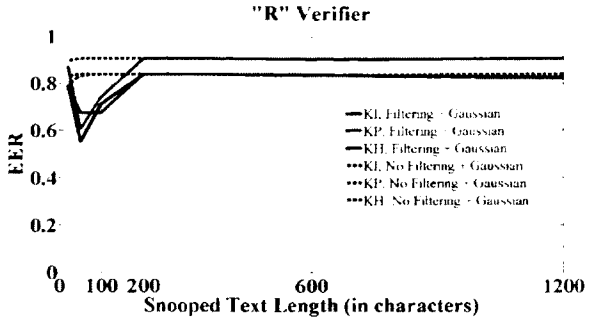


(c)

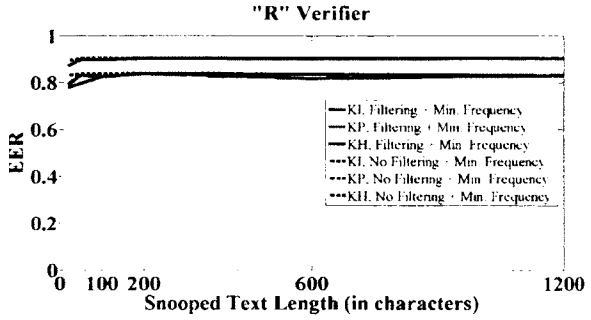


(d)

Figure B.4: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "R" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 300$, plots *c* and *d* correspond to $M = 350$.

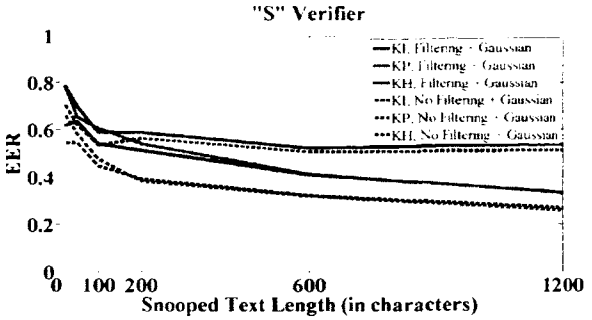


(a)

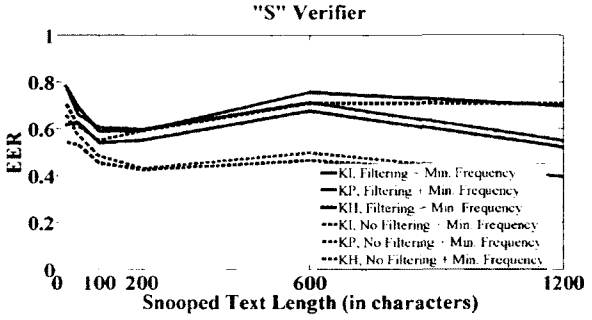


(b)

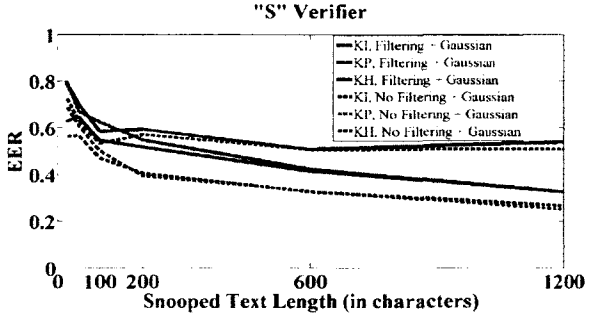
Figure B.5: Comparison of attack EERs using attack configurations 1–12 (plots *a*) and attack configurations 13–24 (plots *b*) for "R" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots correspond to $M = 500$.



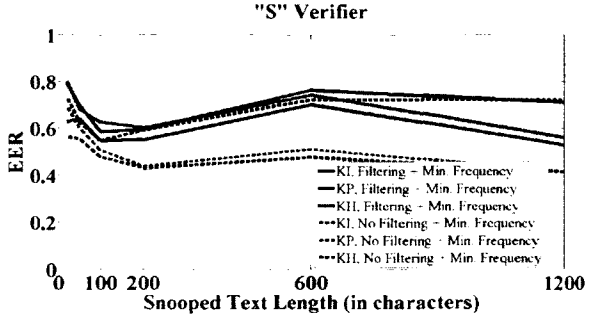
(a)



(b)

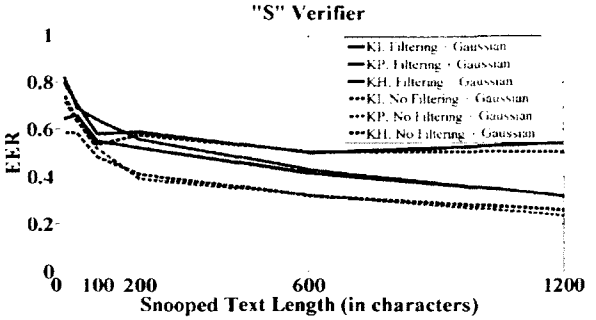


(c)

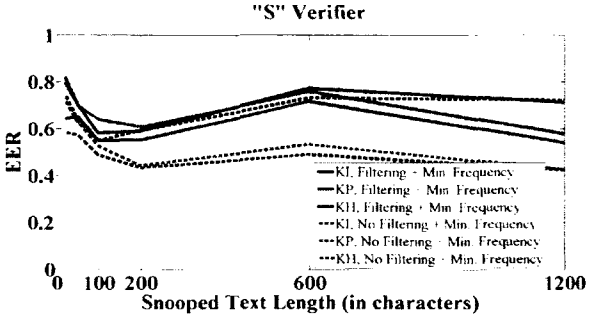


(d)

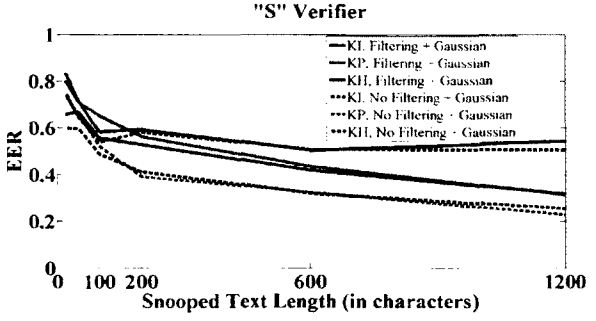
Figure B.6: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "S" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 40$, plots *c* and *d* correspond to $M = 60$.



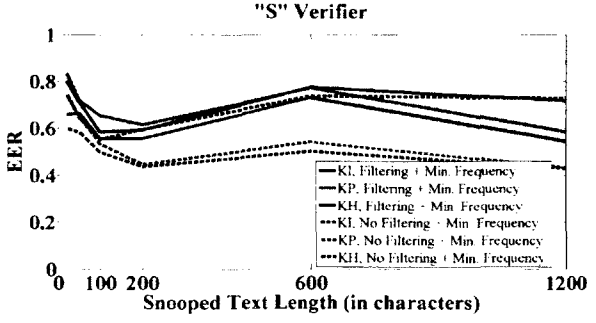
(a)



(b)

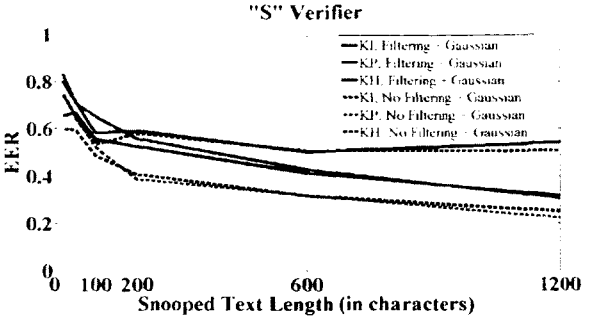


(c)

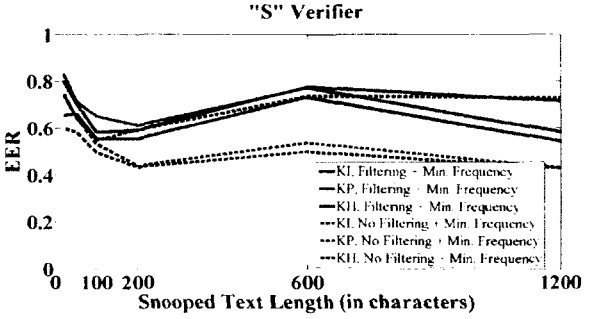


(d)

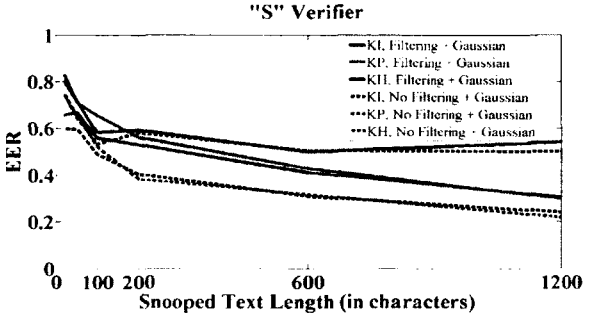
Figure B.7: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "S" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 80$, plots *c* and *d* correspond to $M = 100$.



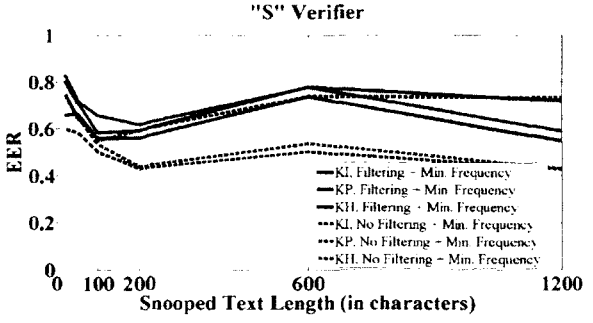
(a)



(b)

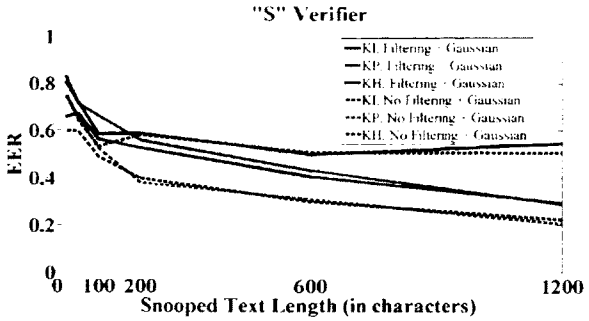


(c)

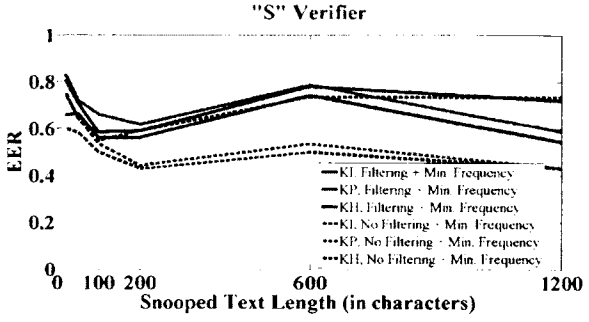


(d)

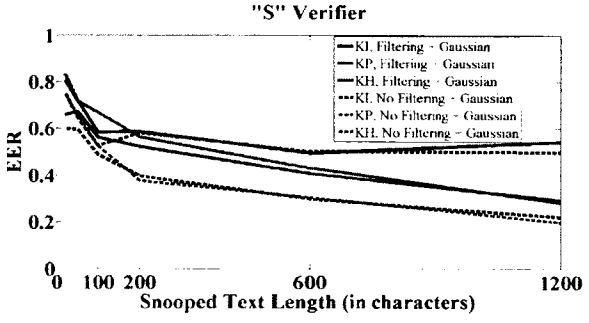
Figure B.8: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "S" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 120$, plots *c* and *d* correspond to $M = 150$.



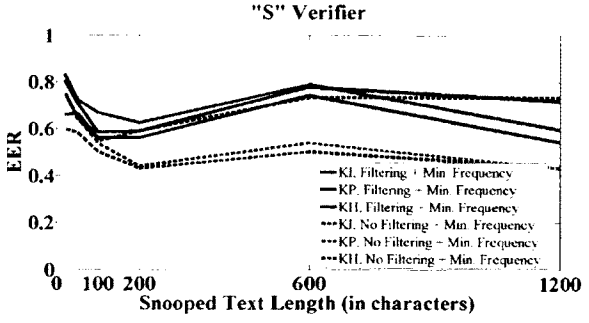
(a)



(b)

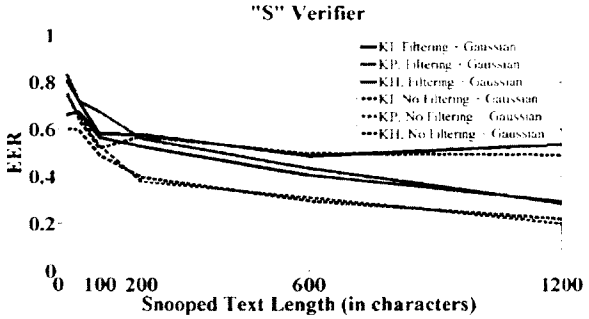


(c)

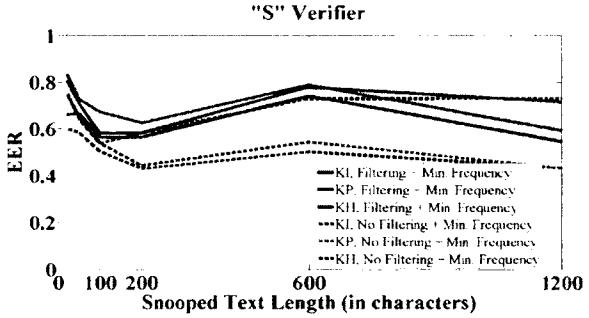


(d)

Figure B.9: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "S" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 300$, plots *c* and *d* correspond to $M = 350$.



(a)



(b)

Figure B.10: Comparison of attack EERs using attack configurations 1–12 (plots *a*) and attack configurations 13–24 (plots *b*) for "S" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots correspond to $M = 500$.

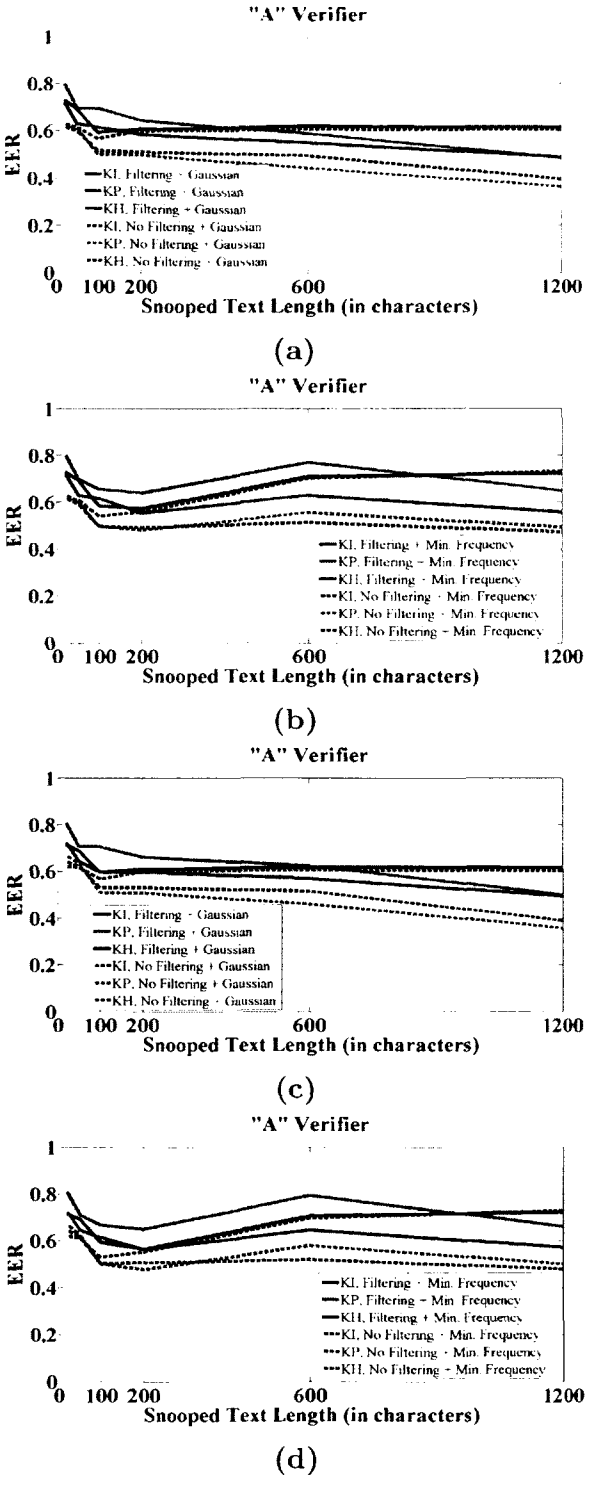


Figure B.11: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "A" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 40$, plots *c* and *d* correspond to $M = 60$.

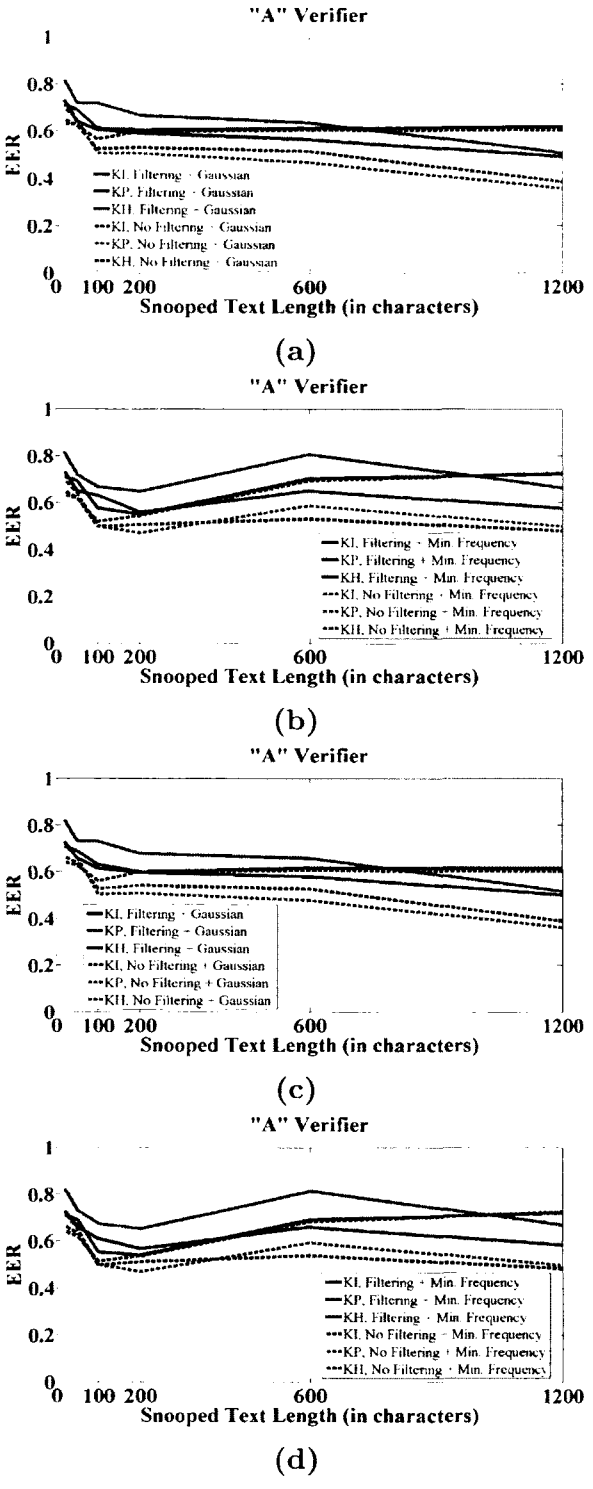


Figure B.12: Comparison of attack EERs using attack configurations 1–12 (plots *a, c*) and attack configurations 13–24 (plots *b, d*) for "A" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 80$, plots *c* and *d* correspond to $M = 100$.

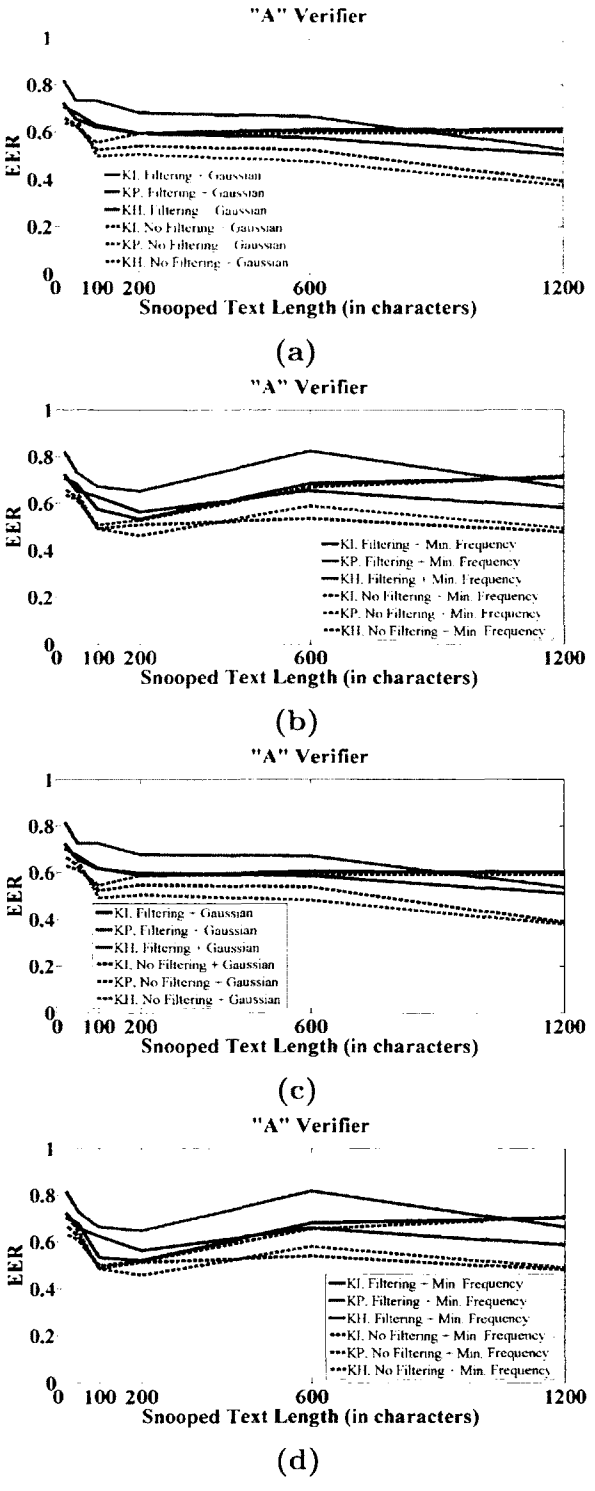
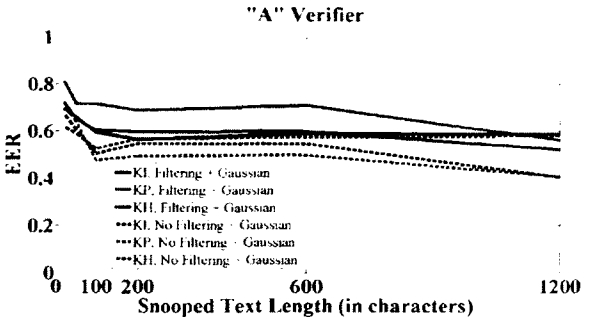
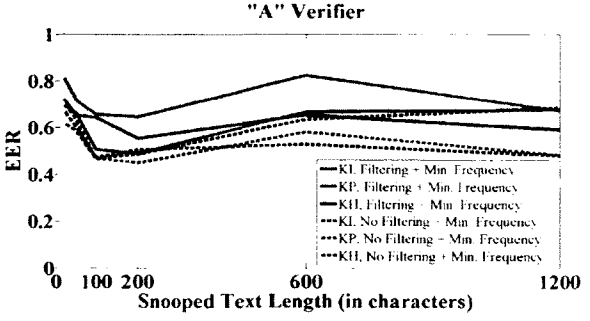


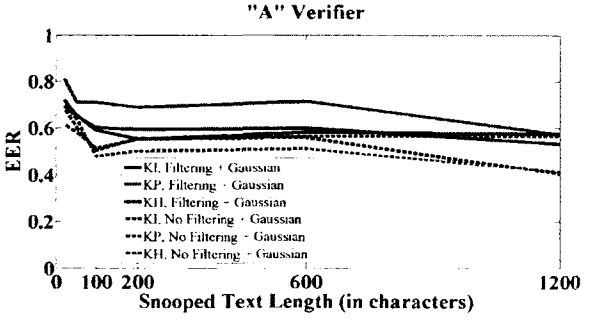
Figure B.13: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "A" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 120$, plots *c* and *d* correspond to $M = 150$.



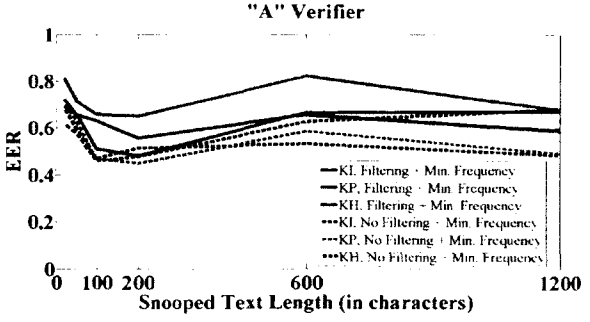
(a)



(b)

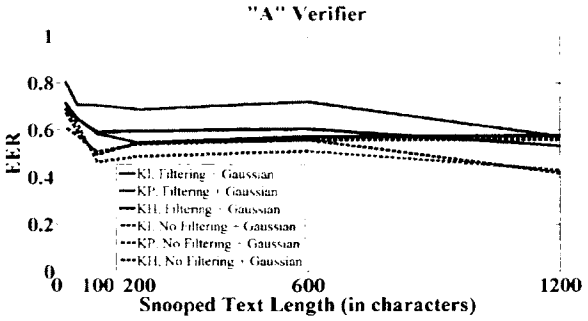


(c)

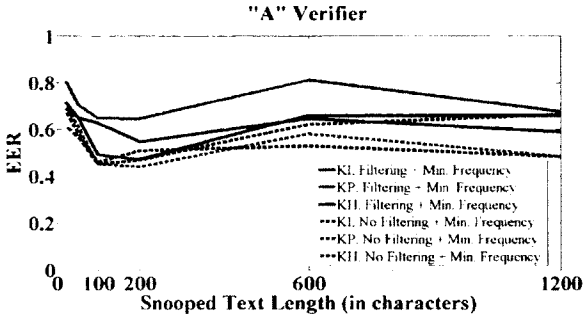


(d)

Figure B.14: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "A" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 300$, plots *c* and *d* correspond to $M = 350$.



(a)



(b)

Figure B.15: Comparison of attack EERs using attack configurations 1–12 (plots *a*) and attack configurations 13–24 (plots *b*) for "A" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots correspond to $M = 500$.

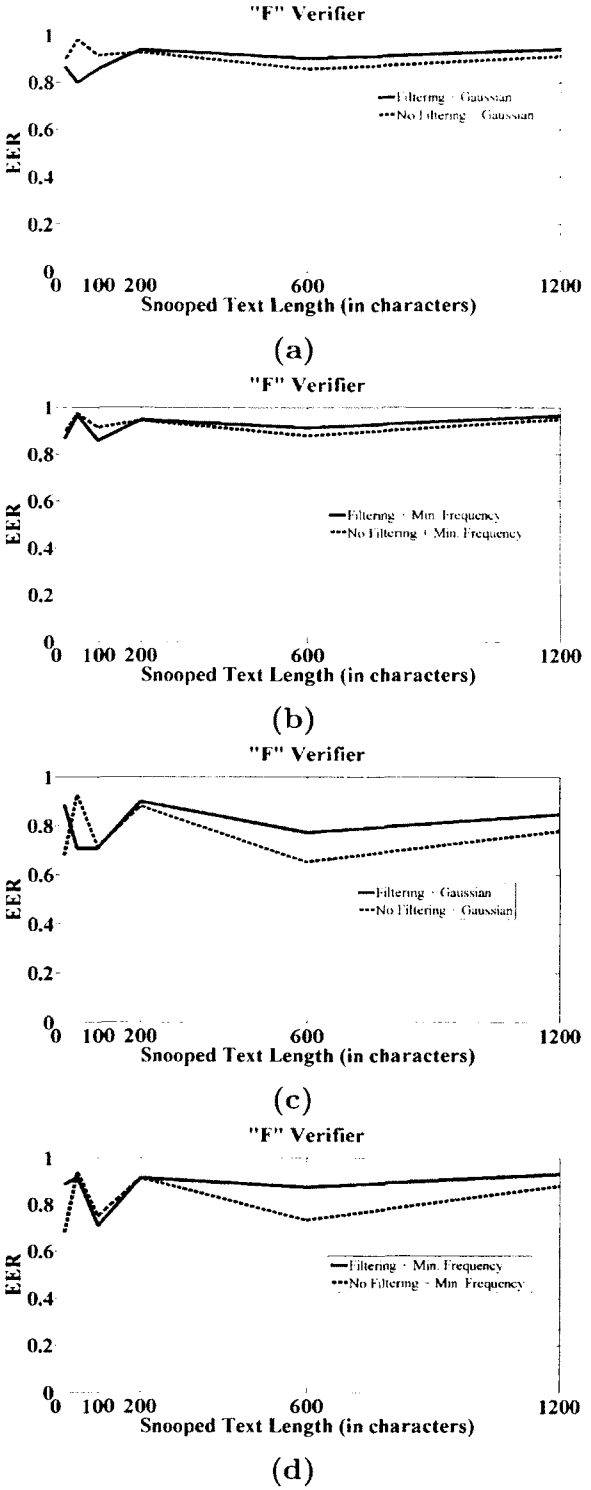


Figure B.16: Comparison of attack EERs using attack configurations 1–12 (plots *a, c*) and attack configurations 13–24 (plots *b, d*) for "F" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 40$, plots *c* and *d* correspond to $M = 60$.

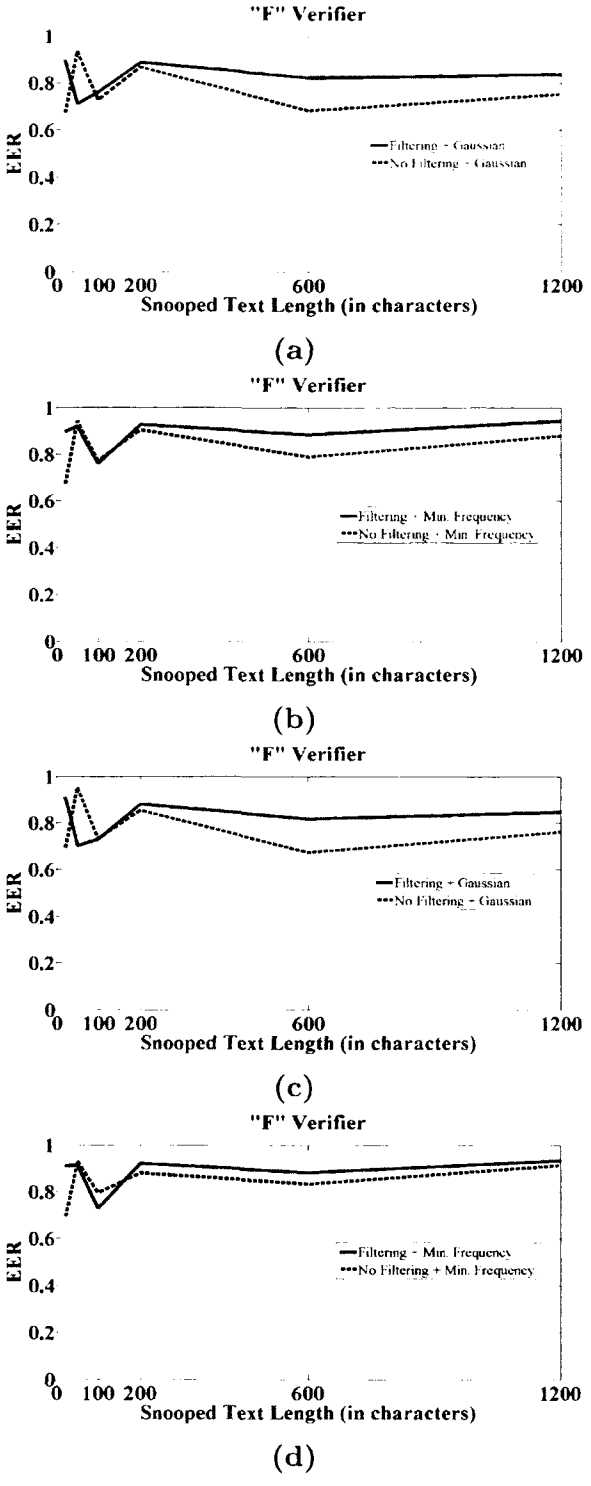


Figure B.17: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "F" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 80$, plots *c* and *d* correspond to $M = 100$.

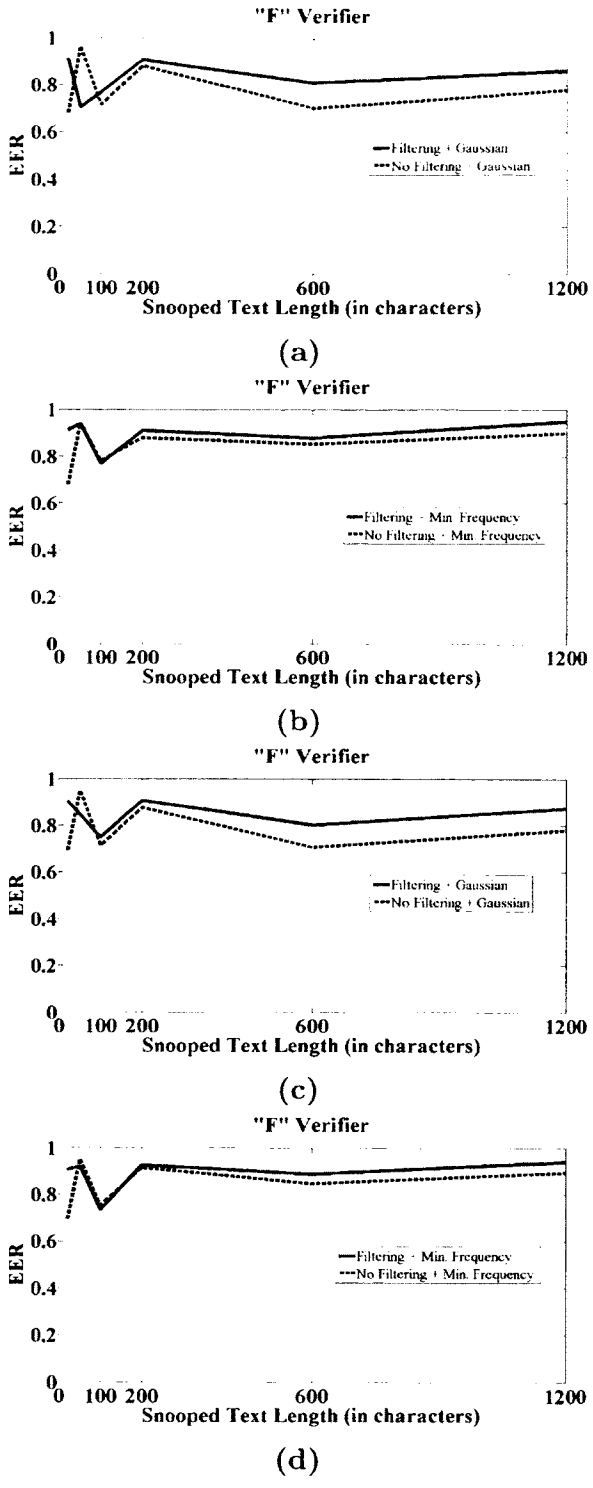


Figure B.18: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "F" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 120$, plots *c* and *d* correspond to $M = 150$.

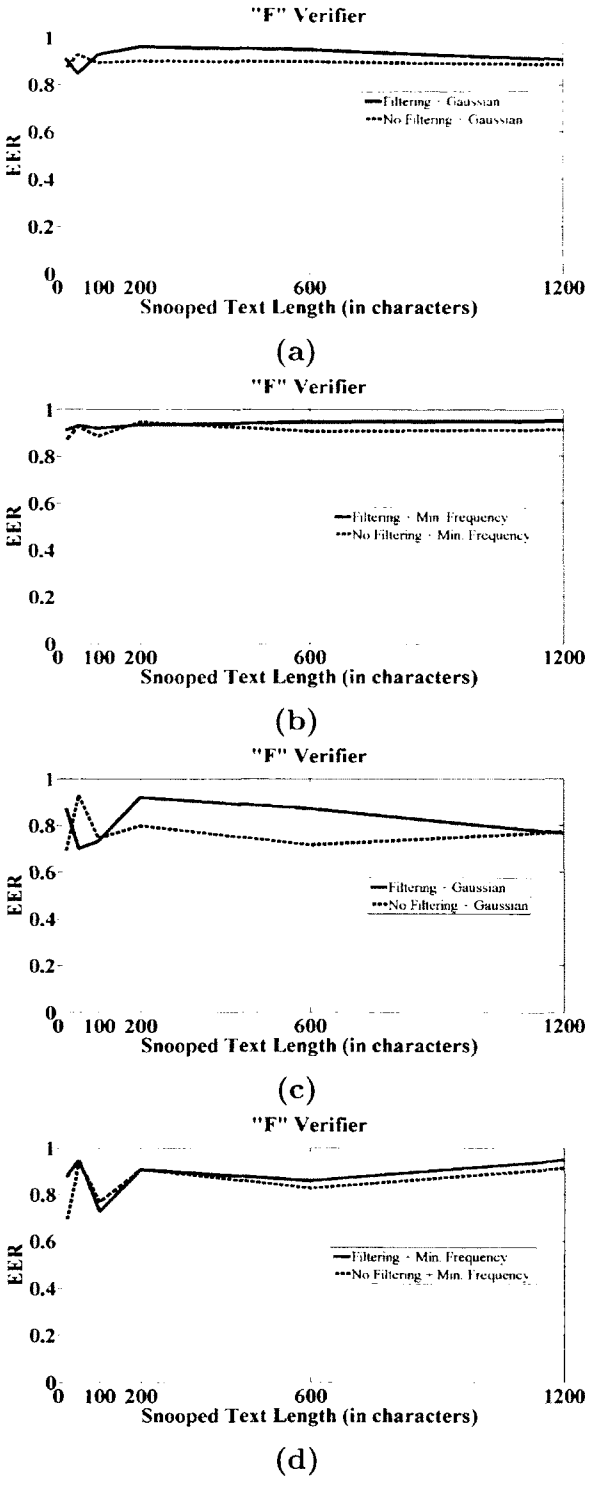


Figure B.19: Comparison of attack EERs using attack configurations 1–12 (plots *a*, *c*) and attack configurations 13–24 (plots *b*, *d*) for "F" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots *a* and *b* correspond to $M = 300$, plots *c* and *d* correspond to $M = 350$.

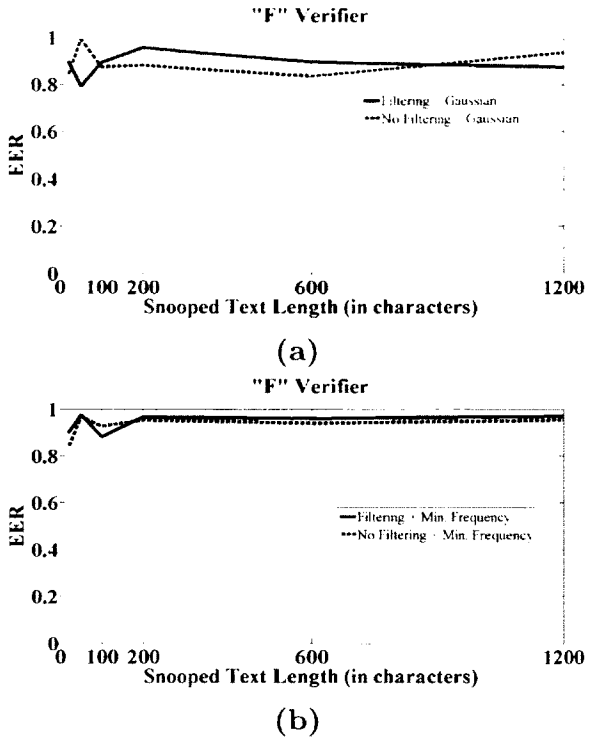


Figure B.20: Comparison of attack EERs using attack configurations 1–12 (plots *a*) and attack configurations 13–24 (plots *b*) for "F" verifier. Refer the Table 4.6 to see the parameter values used in each configuration. The solid lines represent attack EERs when the outliers are filtered and the dashed lines represent attack EERs when the outliers are not filtered. Plots correspond to $M = 500$.

BIBLIOGRAPHY

[1] D. Gunetti and C. Picardi, "Keystroke analysis of free text," ACM Trans. Inf. Syst. Secur., vol. 8, no. 3, pp. 312-347, Aug. 2005.

[2] F. Monrose and A. Rubin, "Authentication via keystroke dynamics," in Proc. of the 4th ACM Conf. on Computer and Commun. Security, 1997, pp. 48-56.

[3] P. Dowland, H. Singh, and S. Furnell, "A preliminary investigation of user authentication using continuous keystroke analysis," in Proc. of the 8th IFIP Conf. on Infor. Security Mgmt. and Small Sys. Security, Nevada, USA, 2001.

[4] T. Shimshon, R. Moskovitch, L. Rokach, and Y. Elovici, "Continuous verification using keystroke dynamics," in Intl. Conf. on Computational Intel. and Security, Los Alamitos, CA, 2010, pp. 411-415.

[5] Niinuma, K., Unsang Park, and Jain, A.K., "Soft Biometric Traits for Continuous User Authentication," in IEEE Trans. on Information Forensics and Security, 2010, vol. 5, no. 4, pp. 771-780.

[6] R. S. Zack, C. C. Tappert, and S.-H. Cha, "Performance of a long-textinput keystroke biometric authentication system using an improved knearest-neighbor classification method," in 2010 Fourth IEEE Intl. Conf. on Theory Applications and Systems (BTAS), 2010, pp. 1-6.

[7] A. Messerman, T. Mustafic, S. A. Camtepe, and S. Albayrak, "Continuous and non-intrusive identity verification in real-time environments based on free-text keystroke dynamics," in IEEE Int. Joint Conf. on Biometrics, 2011.

[8] Yong Sheng, Vir V. Phoha, and S. M. Rovnyak, "A parallel decision tree-based method for user authentication based on keystroke patterns," in IEEE Trans. on SMC Part B: Cybernetics, vol. 35, no. 4, 2005, pp. 826-833.

[9] E. Yu and S. Cho, "GA-SVM Wrapper Approach for Feature Subset Selection in Keystroke Dynamics Identity Verification," in Intl. Joint Conf. on Neu. Nets, 2003, pp. 2253-2257.

[10] B. Francesco, D. Gunetti, and C. Picardi, "User authentication through keystroke dynamics," in ACM Trans. Inf. Syst. Secur, 2002, vol. 5, no. 4, pp. 367-397.

[11] D. Hosseinzade and S. Krishnan, "Gaussian Mixture Modeling of Keystroke Patterns for Biometric Applications," in IEEE Trans. on Systems, Man, and Cybernetics-Part C', 2008, vol. 38, no. 6, pp. 816-826.

[12] C. Shen, Z. Cai, X. Guan, and J. Cai, "A hypo-optimum feature selection strategy for mouse dynamics in continuous identity authentication and monitoring," in IEEE Intl. Conf. on Infor. Theory and Infor. Secur, 2010, pp. 349-353.

[13] M. Pusara and C.E. Brodley, "User Re-Authentication via Mouse Movements," in Proc. ACM Workshop Visualization and Data Mining for Computer Security (VizSec/DMSEC 04), 2004, pp. 1-8.

[14] Ahmed, A.A.E. and Traore, I., "A New Biometric Technology Based on Mouse Dynamics," in IEEE Trans. on Dependable and Secure Computing, 2007, vol. 4, no. 3, pp. 165-179.

[15] Y. C. Yang, "Web user behavioral profiling for user identification," in Decision Support Sys. 49, 3, 2010, pp. 261-271.

[16] R. Gaines, W. Lisowski, S. Press, and N. Shapiro, "Authentication by keystroke timing: Some preliminary results," Tech. Rep. R-256-NSF, Rand corporation, Santa Monica, CA. May, 1980.

[17] D. Umphress and G. Williams, "Identity verification through keyboard characteristics," International Journal of Man-Machine Studies 23, 3, pp. 263-273, 1985.

[18] SendInput [Online]. Available: msdn.microsoft.com.

[19] xsendkeycode [Online]. Available: http://manpages.ubuntu.com/manpages/gutsy/man8/xsendkeycode.8.html.

[20] U. Uludag and A. K. Jain, "Attacks on biometric systems: a case study in fingerprints," in SPIE Security, Steganography and Watermarking of Multimedia Contents VI, vol. 5306, January 2004, pp. 622-633.

[21] R. Maxion and K. Killourhy, "Keystroke biometrics with number-pad input," in 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 28 2010-july 1 2010, pp. 201-210.

[22] S. Joshi, "Naive bayes and similarity based methods for identifying computer users using keystroke patterns," Ph.D. dissertation, Louisiana Tech University, Ruston, Louisiana, 2009.

[23] M. Nisenson, I. Yariv, R. El-Yaniv, and R. Meir, "Towards Behaviometric Security Systems: Learning to Identify a Typist," in Knowledge Discovery in Databases: PKDD 2003, volume 2838 of LNCS, pages 363374. 2003.

[24] K. A. Rahman, K. S. Balagani, and V. V. Phoha, "Making impostor pass rates meaningless: A case of snoop-forge-replay attack on continuous cyber-behavioral verification with keystrokes," in 2011 IEEE Computer Vision and Pattern Recognition Workshops (CVPRW), Colorado, USA, June 2011, pp. 31-38.

[25] L. C. F. Araujo, L. H. R. Sucupira, M. Lizarraga, L. Ling, and J. B. T. Yabu-uti, "User authentication through typing biometrics features," IEEE Trans. on Signal Processing, vol. 53, pp. 851-855, 2005.

[26] K. Killourhy and R. Maxion, "Why did my detector do that?!: Predicting keystroke-dynamics error rates," in Recent Adv. in Intrusion Detection, Canada, 2010, pp. 256276.

[27] D. Stefan, X. Shu, and D. Yao, "Robustness of keystroke-dynamics based biometrics against synthetic forgeries," Computers and Security, vol. 31, no. 1, pp. 109-121, February 2012.

[28] A. Serwadda, V. V. Phoha, and A. Kiremire, "Using global knowledge of users typing traits to attack keystroke biometrics templates," in Proceedings of the thirteenth ACM Workshop on Multimedia and Security, USA, 2011, pp. 51-60.

[29] L. Ballard, D. Lopresti, and F. Monrose, "Forgery quality and its implications for behavioral biometric security," IEEE Trans. on Systems, Man, and Cybernetics-Part B, vol. 37, no. 5, pp. 1107-1118, 2007.

[30] D. Gafurov, E. Snekkenes, and P. Bours, "Spoof attacks on gait authentication system," IEEE Trans. on Information Forensics and Security, vol. 2, no. 3, pp. 491-502, 2007.

[31] Anti-Phishing Working Group. "Phishing Activity Trends Report for the Month of February, 2007" www.antiphishing.org/reports/apwg report february 2007.pdf, February 2007.

[32] Mark Davies, "Word frequency data from the Corpus of Contemporary American English (COCA)," http://www.wordfrequency.info. Last accessed on 15 May 2011.. February 2008.

[33] E. Fry, "Developing a Word List," Elementary English, 34 (7), pp. 456-458, 2007.

[34] [Online]. Available: www.vmware.com/products/vsphere.

[35] [Online]. Available: http://research.cs.wisc.edu/condor.

[36] [Online]. Available: www.ushistory.org/declaration/document/.

[37] [Online]. Available: www.cs.virginia.edu/ robins/YouAndYourResearch.html.

[38] Charles Dickens, "David Copperfield," Penguin Classics, 1850.

[39] Leslie Stephen, "Samuel Johnson," Harper and Brothers, 1879.

[40] Henry David Thoreau,"Leslie Stephen," Houghton, Mifflin, and Company, 1854.

[41] V. Phoha and S. Joshi, "Methods of identifying users based on text entered on keyboard," Patent Pending, 2010.

[42] Y. Wang, T. Tan, and A. Jain, "Combining face and iris biometrics for identity verification," in Proceedings of Fourth International Conference on AVBPA, 2003, pp. 805-813.

[43] A. Jain, K. Nandakumar, and A. Ross, "Score normalization in multimodal biometric systems," Pattern Recognition, vol. 38, no. 12, pp. 2270-2285, 2005.

[44] D. R. Ridley and M. Lively, "English letter frequencies and their applications: Part iidigraph frequencies," Psychological Reports, vol. 95, no. 3, pp. 787-794, July 2004.