cryptography

MDPI

Article

PassT5: A Transformer-Based Deep Learning Model for Credential Tweaking Attacks

Dhuha Alharbi [1,*], Abdullah Alshahrani [1] and Adel Alshamrani [2]

[1] Department of Computer Science and Artificial Intelligence, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia; asalshahrani2@uj.edu.sa
[2] Department of Cybersecurity, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia; asalshamrani@uj.edu.sa
* Correspondence: dhuha.alhrbi@gmail.com

Abstract

Users frequently reuse a password across websites with only minor modifications, so the compromise of one password lets an adversary guess the others by predicting the modification. Existing credential-tweaking models achieve limited accuracy. We present PassT5, which, to our knowledge, is the first application of the pre-trained T5 encoder–decoder Transformer to credential tweaking, framed as a sequence-to-sequence task. We fine-tune T5-Small on password-transformation pairs rebuilt from the 4iQ leak, keeping only pairs that represent a genuine modification, and measure the Success Rate at K (SR@K), the percentage of tweaked passwords guessed within the top-K candidates. Across three training runs under a user-level 70/15/15 split, PassT5 recovers 44.69% ($\pm$0.32) of passwords at $K = 10$, 56.51% ($\pm$0.11) at $K = 100$, and 62.80% ($\pm$0.39) at $K = 1000$. An input-agnostic popularity baseline reaches only 0.68%, 1.40%, and 2.02% at $K = 10$, 100, and 1000, and a stronger input-dependent rule-based baseline 36.72%, 40.72%, and 41.26%, indicating that PassT5 learns password-specific transformations beyond both frequency and hand-crafted rules. The highest SR@1000 previously reported on 4iQ is 23.52% (PassTrans); because samples and splits differ, this is a same-dataset rather than controlled comparison, and on an unfiltered all-pairs benchmark PassT5 scores 18.36% versus PassTrans's 23.52% at K = 1000, so its much larger tweak-only margin reflects the well-posed task rather than a universal model advantage. Encoder–decoder Transformers therefore model credential tweaking effectively and can inform stronger password defences.

Keywords: credential tweaking attack; deep learning; password authentication; password guessing attack; password reuse

Check for updates

Academic Editor: Aleksandr Ometov

Received: 22 October 2026
Revised: 23 November 2026
Accepted: 24 November 2026
Published: 26 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Password-based authentication remains the dominant method for protecting online accounts. As more services move online, users must manage hundreds of accounts, and they frequently cope by reusing the same password or applying simple, predictable modifications to it. At the same time, data breaches have grown more frequent, giving attackers direct access to large collections of real passwords. These leaked datasets are a valuable resource for learning the patterns by which users modify their passwords across sites [1], which in turn makes credential-tweaking attacks substantially easier. Traditional guessing approaches have become inadequate against this threat. Rule-based methods and pattern matching, such as those used in tools like HashCat [2] and John the Ripper [3], rely on hand-crafted rules that must be specified in advance. As a result, they do not generalise well to the diverse and

evolving ways users transform their passwords [4]. Deep learning models have improved on these approaches, yet current password-guessing models such as Pass2Path [5] and Pass2Edit [6] still struggle to capture the complexity of real-world password transformations. Sequence-to-sequence Transformers offer a promising alternative. Pre-trained models such as T5 [7] learn structured input–output mappings and, through transfer learning, have proven effective across a range of NLP tasks such as machine translation and paraphrase generation—properties that appear well suited to modelling password transformations. Despite this potential, T5 has not yet been applied to credential tweaking, and existing Transformer-based models achieve only limited success on the task.

To address these limitations, we propose PassT5, an encoder–decoder model that fine-tunes T5-Small to generate credential-tweaking candidates directly from a leaked password.

We evaluate with the Success Rate at K (SR@K) metric. Averaged over three independent seeds, PassT5 reaches a mean SR@1000 of 62.80% ($\pm$0.39) on a user-level test subset of our 4iQ-derived data, against the highest SR@1000 previously reported on 4iQ, 23.52% (PassTrans). Because the prior results come from their original papers under different samples and splits, we read this comparison as anchored to a common dataset rather than a controlled re-evaluation (Section 5.3). On an unfiltered all-pairs benchmark PassT5 instead scores 18.36% at $K = 1000$ (Section 5.4), so this large margin is specific to the well-posed tweak-only task.

This paper makes the following contributions:

- We frame credential tweaking as a sequence-to-sequence problem and fine-tune the T5-Small encoder–decoder to generate tweaked passwords directly, instead of predicting edit operations or labelling tokens as earlier recurrent and encoder-only models do.
- We rebuild the credential-tweaking dataset from the 4iQ leak and keep a pair only when its normalised edit distance is at most 0.5, so that every pair is a real modification rather than two unrelated passwords.
- We show that the model learns genuine transformations rather than password popularity: it clearly outperforms both a frequency-only popularity baseline that ignores the input entirely (2.02% SR@1000) and a more demanding hand-crafted rule-based baseline in the spirit of Das et al. (41.26% SR@1000).
- We ablate the main design choices: pre-training contributes roughly twelve points of SR@1000 over an identically sized model trained from scratch, and we further compare subword against byte-level tokenisation (ByT5), vary the sampling hyperparameters (k, τ), and compare decoding strategies (top-k sampling versus beam and greedy search).
- We evaluate with SR@K over frequency-ranked unique candidates, user-level splits, three-seed means with standard deviations, and Wilson confidence intervals. We isolate the effect of the tweak-only filter from model capability through an unfiltered all-pairs evaluation. We frame the comparison with prior work as anchored to a common dataset (4iQ) rather than a controlled re-evaluation.
- We analyse where the model succeeds and fails, breaking results down by transformation type and by edit distance, where 90.5% of failures fall within three single-character edits of a generated candidate.

The remainder of the paper is organized as follows. Section 2 reviews background on Transformers and credential tweaking, and Section 3 surveys related work. Section 4 describes the PassT5 model, and Section 5 reports the experimental results, which are discussed in Section 6. Section 7 covers ethical considerations and Section 8 covers the defensive applications of the model, before Section 9 concludes the paper.

https://doi.org/10.3390/cryptography10060085

2. Background

2.1. Transformer Models

Transformers, introduced by Vaswani et al. [8] in 2017, are an encoder–decoder architecture built on self-attention and now standard for sequence tasks such as translation and summarisation. The encoder maps the input sequence into a context-aware representation through stacked self-attention and feed-forward layers, with each token weighing its relation to all others; self-attention thereby captures the long-range dependencies that recurrent networks tend to lose. The decoder generates the output token by token, using cross-attention to stay grounded in the encoder's representation of the input [9]. These properties—direct modelling of long-range structure and a conditional encoder–decoder formulation—motivate our use of a pre-trained Transformer for password transformation.

2.2. Credential Tweaking Attack

Password reuse is one of the most common security weaknesses among Internet users: many people reuse a password across sites, or make small, predictable changes to an existing one rather than creating a new password for each account. In a credential-tweaking attack, the adversary obtains a user's leaked password from one site and tries to guess that same user's password on another site by applying likely transformations, submitting candidate guesses one by one until access succeeds or the attempts are exhausted [5]. Das et al. [10] were the first to study this behaviour, in 2014. They examined several hundred thousand leaked passwords from eleven sites and found that 43–51% of users reuse a password across sites. When users did change a password, the changes followed simple patterns, such as capitalizing the first letter, appending a digit, or adding a common symbol. Their cross-site guessing algorithm recovered 30% of transformed passwords within 100 attempts, compared with 14% for a standard guesser. This showed that credential tweaking is a distinct threat, and it motivated the models reviewed in Section 3.

3. Related Work

Research on credential tweaking has progressed along three broad lines: models that learn edit or reuse operations directly, methods that cast guessing as ranking or classification, and generative or hybrid approaches that model the overall password distribution. We review each in turn. Because reported success rates differ widely in dataset, guess budget, and task, Table 1 lists only the directly comparable results—edit- and reuse-based models on 4iQ at 10^3 guesses—while the remaining methods, each at its own setting, are collected in Appendix A (Table A1).

Table 1. Edit- and reuse-based password-guessing models on 4iQ at 10^3 guesses—the closest comparable setting for PassT5. Reported figures are taken from each method's own paper and differ in sample size and split, so this is an anchored same-dataset comparison rather than a controlled re-evaluation.

Model	Approach	Key Innovation	Dataset	Guesses	Reported SR
Pass2Path [5]	LSTM seq2seq	Neural transformation prediction	4iQ	10^3	14.83% [a]
Pass2Edit [6]	RNN/GRU	Multi-step edit classification	4iQ (20k)	10^3	17.82%
PointerGuess [11]	Pointer-generator	Character-level copy mechanism	4iQ (20k)	10^3	20.43%
PassBERT [4]	BERT encoder	Pre-training framework	4iQ (100k)	10^3	17.96%
PassTrans [12]	Transformer	Full Transformer architecture	4iQ (100k)	10^3	23.52%
PassT5 (ours)	Enc–dec (T5)	Seq2seq transformation generation	4iQ (5k)	10^3	**62.80%**

[a] 14.83% is PassTrans's re-run of Pass2Path, not Pass2Path's own reported figure (over 16%).

3.1. Edit- and Reuse-Based Models

The dominant line models the transformation between a known password and its tweaked variant. Pal et al. [5] proposed Pass2Path, an LSTM sequence-to-sequence model

that predicts the sequence of edits turning one password into a similar one; trained on 1.4 billion email–password pairs, it compromised over 16% of accounts within 10^3 guesses when one of the user's passwords is known. Pass2Edit [6] reframed the task as multi-step decision making, using an RNN/GRU to classify which edit to apply at each step, improving over Pass2Path. These models, however, characterise reuse through syntactic similarity measures (e.g., edit distance) rather than generating the transformation itself, which limits them on structurally non-trivial but systematic edits. PointerGuess [11] addressed this with a pointer mechanism that separates the characters a user keeps from those they change, modeling reuse from both personal and population-wide perspectives and reaching an average success rate of 25.85% (range 10.08–45.00%), ahead of Pass2Edit and TarGuess-II [13].

All of these rely on recurrent architectures, which capture limited long-range structure within a password. Transformer-based models followed. He et al. (PassTrans) [12] applied a Transformer to password reuse and credential tweaking, cracking 23.52% of the 4iQ dataset within 10^3 guesses—the highest figure previously reported on 4iQ, and our main point of comparison. PassBERT [4] used a bidirectional (encoder-only) Transformer in a pre-train/fine-tune framework with three attack modes—conditional, targeted, and adaptive rule-based guessing—and, in the targeted setting, improved over Pass2Path on the BreachCompilation (4iQ) and Collection datasets. Most recently, Zou et al. [14] scaled to billion-parameter models with PassLLM, fine-tuning a decoder-only large language model (Mistral-7B) with LoRA; in the reuse (credential-tweaking) setting, where one of a user's other passwords is known, it cracks 8.27–9.66% more passwords than Pass2Edit within 10^3 guesses from a single known password, confirming that large pre-trained models carry useful transformation signal—though through a decoder-only rather than an encoder–decoder formulation. Per-dataset figures appear in Table 1 (4iQ-comparable models) and Table A1 (other settings).

3.2. Ranking- and Classification-Based Methods

A second line treats guessing as ranking or classification. RankGuess [15] frames password creation as a Markov decision process and uses adversarial ranking to order guesses, applying to trawling, PII-based, and conditional guessing; with known PII it reaches 58.21–91.95% success for common users, albeit at a very large (10^{12}) guess budget. RFGuess [16] applies a Random Forest to re-encoded password characters, with trawling, PII, and reuse variants; its reuse variant improves 7–22% over Pass2Path on English datasets while remaining below TarGuess-II [13]. These methods broaden the toolkit, but each models a single technique in isolation.

3.3. Generative and Hybrid Approaches

A third line learns the overall password distribution or combines models. PassGAN [17] generates passwords with a Generative Adversarial Network and, combined with HashCat [2], matches 51–73% more passwords than HashCat alone. GENPass [18] targets cross-site generalisation with a PCFG+LSTM generator, improving cross-site match rates by 16–30% over a single-dataset LSTM. Pasquini et al. [19] refined deep-generative modeling with a "strong locality" latent space for conditional generation, reaching 51.80% on RockYou at 5×10^{10} guesses (versus 34.19% for PassGAN). GuessFuse [20] fuses PCFG, Markov, FLA [21], and RFGuess guess lists through multi-view learning to improve cracking within 10^7 guesses. These approaches, however, target untargeted or distribution-level guessing rather than the per-user transformation that credential tweaking requires.

3.4. Summary and Gap

Table 1 lists the directly comparable 4iQ results, with the broader set of methods in Appendix A. Two observations motivate the present work. First, none of them uses a full encoder–decoder Transformer to generate the tweaked password directly in a sequence-to-sequence manner; they rely on recurrent, encoder-only, or decoder-only architectures. Second, even the strongest prior result on 4iQ—23.52% at 10^3 guesses (PassTrans)—leaves most tweaked passwords unrecovered, so success rates remain well below what the threat warrants and there is substantial room for improvement. These two gaps motivate PassT5, which adapts a publicly available pre-trained encoder–decoder checkpoint (T5) to the credential-tweaking task. Seen more abstractly, credential tweaking is a question of information leakage: one of a user's passwords reveals a great deal about the rest. This links our problem to the formal study of information flow in security. Denning's lattice model [22] and the non-interference framework of Goguen and Meseguer [23] set out when one part of a system may influence, or reveal information about, another, and interference-based behavioural equivalences later carried these ideas into concurrent systems [24]. Persistent formulations strengthen this notion by requiring security to hold not merely at the outset but in every reachable state of the system, so that no intermediate configuration leaks a protected secret [25]. A leaked password is an observation made at an intermediate point of a user's history, so this persistent view fits credential tweaking closely: resistance to reuse is precisely the demand that such an intermediate observation not reveal the user's other passwords at any later state. These notions remain binary, however—information either flows or it does not—while password reuse is a matter of degree, which is the province of quantitative information flow [26]: not whether one secret leaks another, but how much. Our results naturally fall into this picture. SR@K quantifies this leakage: given one known password of a user, how much a model can recover about another, hidden password of the same user. The defensive directions we outline—reuse checkers and personalised strength meters—are then ways to bound that leakage.

4. PassT5: The Proposed Model

4.1. Overall Framework

PassT5 is a Transformer-based model that generates credential-tweaking candidates by learning transformation patterns between leaked and tweaked passwords. As illustrated in Figure 1, the model follows a seven-stage pipeline: building a dataset of paired leaked and tweaked passwords, splitting it into training, validation, and test sets, tokenising them using the T5 sub-word tokeniser, fine-tuning a pre-trained T5 encoder–decoder architecture, generating up to K distinct candidate passwords through stochastic sampling, searching against the target password to determine a match, and aggregating success rates across the held-out test set.

4.2. Transformer-Based Credential-Tweaking Model

Figure 2 presents the overall architecture of PassT5. We describe the main components in detail below.

4.2.1. Input Preparation: Tokenisation and Embedding

Before the leaked password is processed by the encoder, it must be prepared through two basic processes: tokenisation and embedding. Tokenisation: the password is not processed as a single continuous string. SentencePiece is used instead to divide it into understandable units called tokens [27,28]. For instance, our tokeniser breaks the password `Admin2024!` into five pieces rather than ten distinct characters: a leading whitespace

marker (SentencePiece's convention for marking the start of a word) followed by [Admin][20][24][!]. The end-of-sequence token </s> is then appended. The method uses a sub-word vocabulary of roughly 32,000 tokens rather than a character-level vocabulary of only about 100 distinct symbols (letters, digits, and special characters), encoding frequent multi-character patterns as single tokens, and therefore increasing processing efficiency while preserving the semantic structure of the inputs.

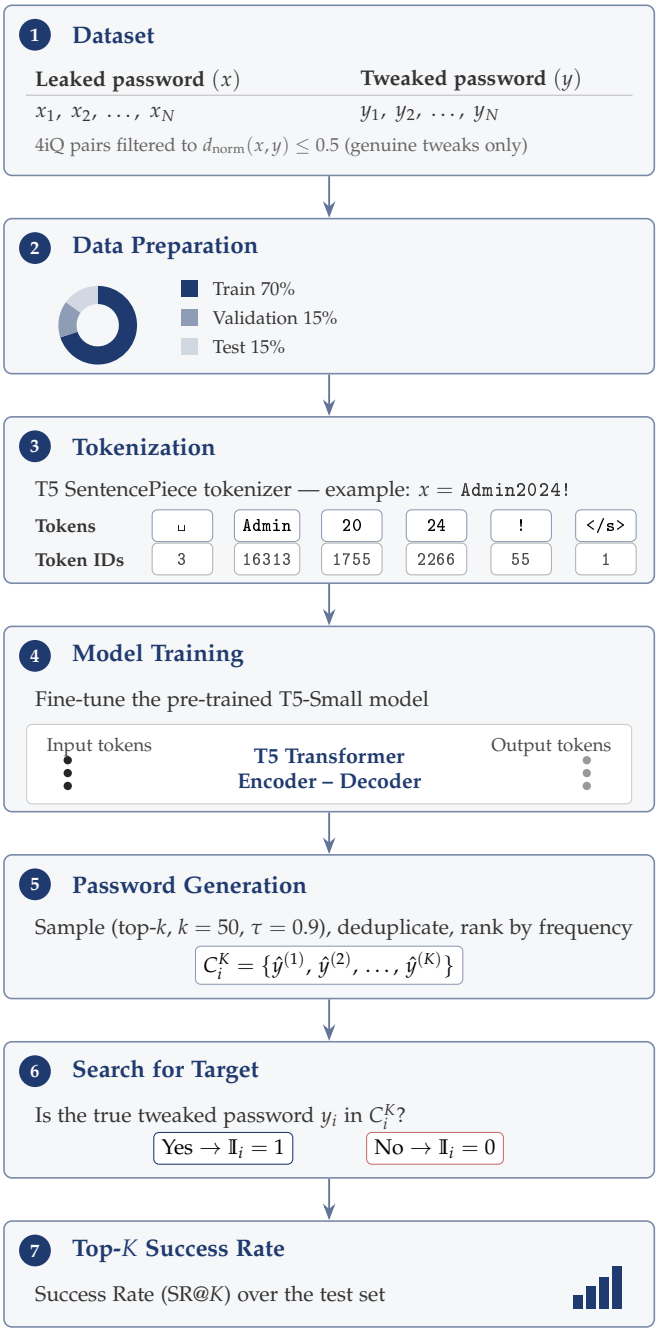


Figure 1. Overview of the PassT5 pipeline. The data is split 70%/15%/15% for training, validation, and testing. SentencePiece tokenises each password (e.g., `Admin2024!` → [␣][Admin][20][24][!][</s>]); the T5 encoder–decoder is fine-tuned on these pairs, then samples candidate passwords for each input, which are deduplicated and ranked by sampling frequency to form the top-K list. Each candidate set is checked against the true password, and the results are aggregated into the Top-K Success Rate.

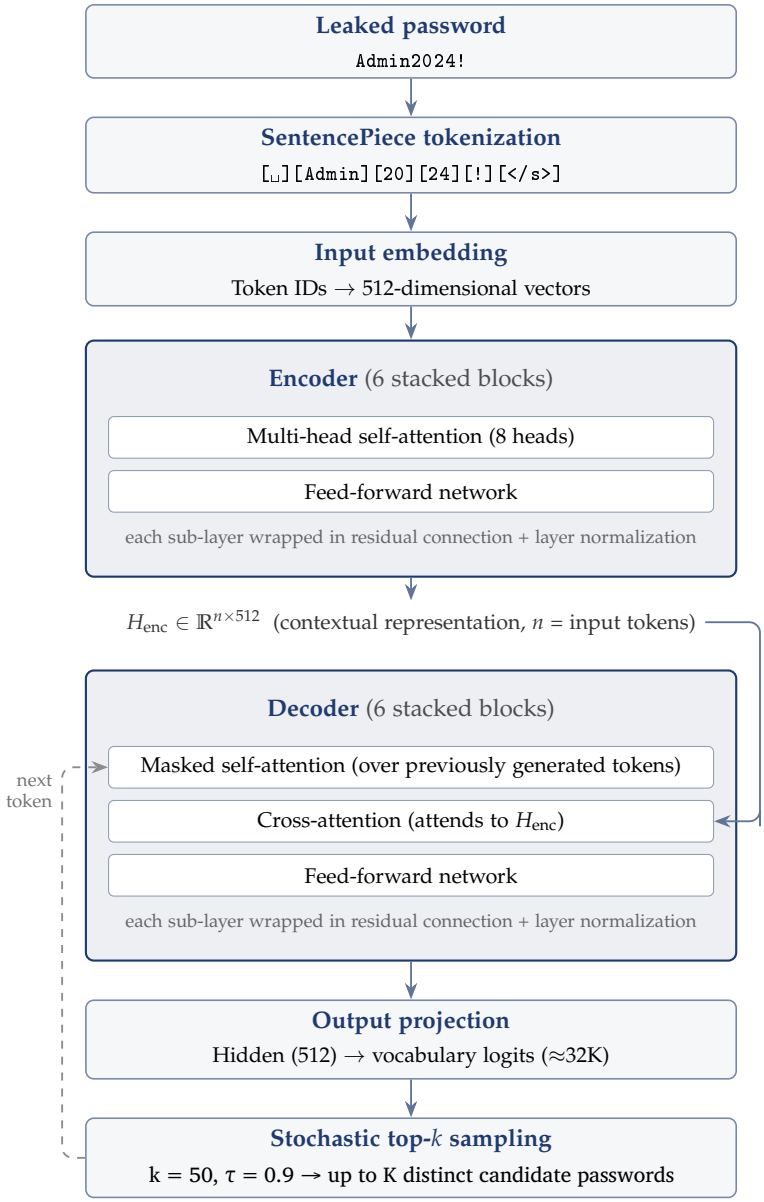


Figure 2. PassT5 architecture for credential tweaking attacks. The leaked password is first tokenised using SentencePiece and converted to 512-dimensional embeddings. The encoder processes these embeddings through 6 stacked Transformer blocks with 8 attention heads each, producing a contextualised representation H_{enc}. The decoder generates candidate passwords autoregressively, using self-attention over previously generated tokens and cross-attention to H_{enc}. Finally, the output projection layer converts hidden states to vocabulary logits, and stochastic top-k sampling ($k = 50$, $\tau = 0.9$) produces up to K distinct candidate passwords.

SentencePiece's vocabulary was built from ordinary English web text (the C4 corpus), so it does not include tokens for a few ASCII symbols that appear in passwords. Encoding each password and decoding it back shows that eight characters do not survive this round trip: the tab character, <, \, ^, `, {, }, and ~. If one of these characters appears in a target password, the model cannot reproduce it exactly, no matter how well it has learned the underlying transformation (for example, `sweetheart<3` is decoded as `sweetheart3`). In our dataset the impact is limited: although we filter out non-ASCII pairs during data construction, these eight characters are themselves valid ASCII and are therefore retained, as they occur in real passwords. Across the full test set, they affect only 108 of 17,280 pairs (0.63%), of which 63 (0.36%) have an unreachable target; we count these as model failures.

Still, this is a real limitation of using a tokeniser built for general text on passwords; a character- or byte-level tokeniser avoids this issue; we evaluate one such model (ByT5) in Section 5.11.2, and a tokeniser purpose-built for passwords remains a direction for future work.

Embedding: Following tokenisation, each token is transformed into a 512-dimensional dense vector from a straightforward numerical identifier (token ID). The Transformer cannot process abstract numbers directly; hence, this conversion is required. Each dimension of the vector encodes a lexical feature learned during training. These features might provide details about the character's kind (letter or digit), its relative position, or other password transformation patterns. After these two processes, the encoder receives not the leaked text but a dense vector representation of the input password. The encoder processes these vectors through multiple layers of self-attention and deep networks, resulting in an overall contextual representation of the characteristics of the leaked password. This representation is what the decoder later uses when generating the candidate passwords.

4.2.2. Encoder: Understanding the Leaked Password

The encoder interprets the leaked password's structure and patterns. Instead of memorizing individual characters, it analyses the password as a whole, identifying patterns, such as the way the letters in `Admin` are followed by numbers in `2024`. By using 8 attention heads, the approach enables various encoder components to concentrate on distinct elements at the same time, such as digit patterns, capitalisation, and unique character locations. The final representation encapsulates the key semantic and structural elements of the compromised password.

4.2.3. Decoder: Generating Candidate Passwords

The decoder generates the target password in an autoregressive manner, one token at a time. It must choose whether to produce a letter, number, or special character at each stage. To access the encoder's representation of the compromised password at each generation stage, the decoder employs cross-attention rather than operating independently. By doing this, the generated candidates are guaranteed to retain structural coherence with the input. For example, if the encoder identified the pattern [letters][digits] in `Admin2024!`, the decoder will assign high probabilities to digits when continuing after `Admin`. About 32,000 tokens in the vocabulary are available for selection, and their probabilities are computed based on both: (1) what has already been generated (via self-attention), and (2) the encoder's understanding of the leaked password (via cross-attention). This back-and-forth between encoder and decoder enables the model to generate plausible password transformations while maintaining structural consistency with the original.

4.2.4. Token Selection: Top-k Sampling

We employ top-k sampling (with sampling cutoff $k = 50$, distinct from the guess-count $K \in \{10, 100, 1000\}$ used in SR@K) instead of choosing the token with the highest probability directly, which could result in a single, deterministic outcome. At each decoding step, the probability mass is restricted to the $k = 50$ most likely tokens, and the next token is sampled from this reduced set according to their temperature-scaled probabilities, using a temperature of $\tau = 0.9$ to balance diversity and reliability. This gives us a variety of realistic candidates instead of just one.

4.2.5. Generating Multiple Candidates

Rather than relying on a single candidate, we draw many samples from the model for each input. Because sampling is stochastic, repeated draws yield different candidates

but also many duplicates. We therefore deduplicate the samples and rank the distinct candidates by how often each was sampled, which approximates their probability under the model, and compute SR@K over the top-K distinct candidates. When the model produces fewer than K distinct candidates for an input, the ranked list is simply shorter than K (its length is $\min(K, |U_i|)$); no padding or filler candidates are added. A prediction counts as successful if the tweaked password appears among these top-K candidates.

4.3. Mathematical Formulation of the Proposed Framework

This section expresses the framework described above formally. The following steps translate the encoder–decoder mechanism and the candidate generation strategy into a set of definitions.

4.3.1. Dataset Definition

Let $D = \{(x_i, y_i)\}_{i=1}^{N}$ denote the password transformation dataset, where x_i represents the leaked password, y_i represents the tweaked password of the same user, and N is the total number of password pairs. Each pair (x_i, y_i) captures a real-world instance of one of a user's passwords and a related variant of it—the exact behaviour our model aims to learn. The leak carries no reliable timestamps, so a pair represents two related passwords of the same user rather than a dated before-and-after change.

4.3.2. Dataset Construction

Our data is extracted from the 4iQ leak, also known as the BreachCompilation dataset. Each user may have multiple leaked passwords. We group passwords by their corresponding email addresses, which are first normalised (by removing spaces and converting to lower case) so that all case-insensitive variations of the same address are linked to a single user. A naive construction that pairs every password belonging to a user with every other (the 'all-pairs' scheme) produces numerous pairs, but many of these do not represent genuine transformations: two passwords chosen independently by the same user (such as `summer2010` and `xk92!qLp`) are unrelated strings, not modified versions of one another. Including such pairs confuses credential tweaking with the random co-occurrence of passwords, and makes a frequency-only baseline—which ignores the input—appear competitive.

To ensure every pair (x_i, y_i) reflects an actual modification, we filter pairs by their normalised edit distance, defined as the Levenshtein distance between x_i and y_i divided by the length of the longer string:

$$d_{\text{norm}}(x_i, y_i) = \frac{\text{Lev}(x_i, y_i)}{\max(|x_i|, |y_i|)}. \tag{1}$$

We retain a pair only if $d_{\text{norm}}(x_i, y_i) \leq 0.5$, i.e., y_i shares at least half of its structure with x_i, discarding pairs with no genuine transformation relationship. For example, `password` → `password1` has $d_{\text{norm}} \approx 0.11$ and is retained, whereas `summer2010` → `xk92!qLp` has $d_{\text{norm}} \approx 0.9$ and is discarded. This keeps the systematic edits that characterise credential tweaking—appends, truncations, case changes, and small substitutions—while removing unrelated pairs. We stress that edit distance is used only to define which pairs constitute a genuine tweak when constructing the dataset, not as the model's similarity mechanism: PassT5 learns to generate transformations directly, without relying on any explicit similarity score at inference time. After filtering, the dataset comprises 114,519 pairs. For each user, pairs are formed between distinct passwords as ordered pairs, so a password is never paired with itself ($d_{\text{norm}} > 0$ for every retained pair) and both directions (a, b) and (b, a) of a modification are generated. Users were identified by a single deterministic pass over the corpus: we traversed the BreachCompilation files in sorted order and kept the first 50,000 users with

at least one pair satisfying the filter ($d_{\text{norm}} \leq 0.5$), then dropped users whose remaining pairs contained non-ASCII characters, leaving 48,630 users. This selection is a fixed traversal of the leak rather than a uniform random sample from it; the train/validation/test split was subsequently randomised at the user level.

To verify that the filtered dataset is not trivially solvable by target frequency alone, we additionally evaluate a popularity baseline that ignores the input and always guesses the most frequent training targets; its results are reported alongside PassT5 in Section 5.2.

4.3.3. Dataset Splitting

$$|D_{\text{tr}}| \approx 0.70|D|, \quad |D_{\text{val}}| \approx 0.15|D|, \quad |D_{\text{test}}| \approx 0.15|D| \tag{2}$$

where D_{tr}, D_{val}, and D_{test} refer to subsets for training, validation, and testing, respectively. Applied to our 114,519 pairs, this yields 80,070 training, 17,169 validation, and 17,280 test pairs. The split is performed at the user level, so that all pairs from a single user fall in the same partition; because users contribute differing numbers of pairs, the resulting partition sizes approximate rather than exactly match these proportions. The Transformer is trained on password modification patterns using the training subset; the validation subset is used for model selection and early stopping during training; and final performance is assessed on the held-out test subset, which is not used at any point during training or hyperparameter selection. To account for training variance, we repeat this procedure across three independent random seeds (see Training Configuration below) and report test-set results as mean ± standard deviation. Because partitioning is at the user level, no exact pair is shared across partitions: none of the 17,280 test pairs (x_i, y_i) occurs in the training set. A leaked password x can still recur across partitions when it is a common string (5.3% of test inputs), but its paired target y differs, so this does not constitute pair-level leakage.

4.3.4. Tokenisation

Each password pair is mapped to sequences of subword tokens by the SentencePiece tokeniser $T(\cdot)$:

$$T(D) = \{(T(x_i), T(y_i))\}_{i=1}^{N} \tag{3}$$

where $T(\cdot)$ draws tokens from a vocabulary of roughly 32,000 subword units. A worked example and a discussion of character coverage are given in Section 4.2.1.

4.3.5. Conditional Probability of Password Generation

Let f_θ denote the fine-tuned T5 model with learned parameters θ. Given an input password x_i, the model estimates the conditional probability of generating the target (tweaked) password y_i:

$$f_\theta(x_i) = P(y_i \mid x_i; \theta) = \prod_{t=1}^{L} P(y_t \mid y_{<t}, x_i; \theta) \tag{4}$$

where:

- $f_\theta(\cdot)$ is the fine-tuned T5 encoder–decoder model [7] with learned parameters θ,
- y_t is the token generated at decoding step t,
- $y_{<t}$ denotes all tokens generated before step t,
- θ represents the learned model parameters,
- L is the length of the generated sequence.

This equation formalises the autoregressive generation process described earlier. At each step t, the decoder does not generate tokens independently but conditions on both the encoder's representation of the input password x_i (via cross-attention) and all previously

generated tokens $y_{<t}$ (via self-attention). This is exactly the back-and-forth between encoder and decoder mentioned in the conceptual description.

4.3.6. Stochastic Candidate Generation

For each input x_i, we draw stochastic samples from the model, continuing until $K_{\max}$ distinct candidates are collected or a fixed sampling budget B is reached (we use $K_{\max} = 1000$ and $B = 6000$). Let $S_i = (y_i^{(1)}, \ldots, y_i^{(M_i)})$ be the M_i samples drawn:

$$y_i^{(m)} \sim P(y \mid x_i; \theta), \qquad m = 1, \ldots, M_i, \tag{5}$$

where the distribution is shaped by top-k sampling (cutoff $k = 50$) with temperature $\tau = 0.9$. We deduplicate the samples into the set of distinct candidates U_i and score each by its sampling frequency, a Monte-Carlo estimate of its model probability:

$$U_i = \text{unique}(S_i), \qquad \text{freq}_i(u) = \frac{1}{M_i} \sum_{m=1}^{M_i} 1\big[y_i^{(m)} = u\big]. \tag{6}$$

The candidate list C_i^K is formed by taking the K candidates in U_i with the highest freq_i (all of U_i when $|U_i| < K$):

$$\left|C_i^K\right| = \min\left(K, |U_i|\right), \qquad K \in \{10, 100, 1000\}. \tag{7}$$

When the model yields fewer than K distinct candidates, the list is shorter than K; no training passwords or other filler are appended. This formalises the top-k sampling [29] procedure described in the conceptual section.

4.3.7. Password Matching

We establish a threshold function that detects successful predictions for every test sample:

$$\mathbb{I}_i = \begin{cases} 1, & \text{if } y_i \in C_i^K \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

Put differently, if the corresponding candidate set C_i^K consists of any instances of the real tweaked password y_i, the prediction is regarded as successful.

4.3.8. Top-K Success Rate

PassT5's overall performance is evaluated using the Top-K Success Rate:

$$SR@K = \frac{100}{|D_{\text{test}}|} \sum_{i=1}^{|D_{\text{test}}|} \mathbb{I}_i \tag{9}$$

This metric directly reflects the practical effectiveness of the model: it measures the percentage of tweaked passwords that PassT5 can successfully predict within a given number of guesses K. In our experiments this average is computed over the fixed evaluation subset described in Section 5.1.

4.3.9. Overall Workflow

The complete pipeline can be expressed compactly as:

$$D \to (D_{\text{tr}}, D_{\text{val}}, D_{\text{test}}) \to T(\cdot) \to f_\theta(\cdot) \to C^K \to SR@K \tag{10}$$

where each candidate set C^K is drawn by stochastic sampling from f_θ (Equation (5)) and scored with $SR@K$ (Equation (9)).

4.3.10. Training Configuration

The model is trained using the following hyperparameters:

- **Optimiser**: AdamW
- **Learning rate**: 5×10^{-5}
- **Batch size**: 16
- **Number of epochs**: 30 (with early stopping, patience = 5)
- **Loss function**: Cross-entropy loss
- **Random seeds**: 42, 123, 456 (three independent training runs; results reported as mean ± standard deviation across seeds)

Figure 3 shows the training and validation loss for the seed-42 run. The validation loss plateaus after roughly epoch 17 and does not rise thereafter; training selects the checkpoint with the lowest validation loss (epoch 25), and early stopping (patience = 5) ends training before any over-fitting appears. The other two seeds showed a similar convergence pattern and were likewise halted early by the same criterion.

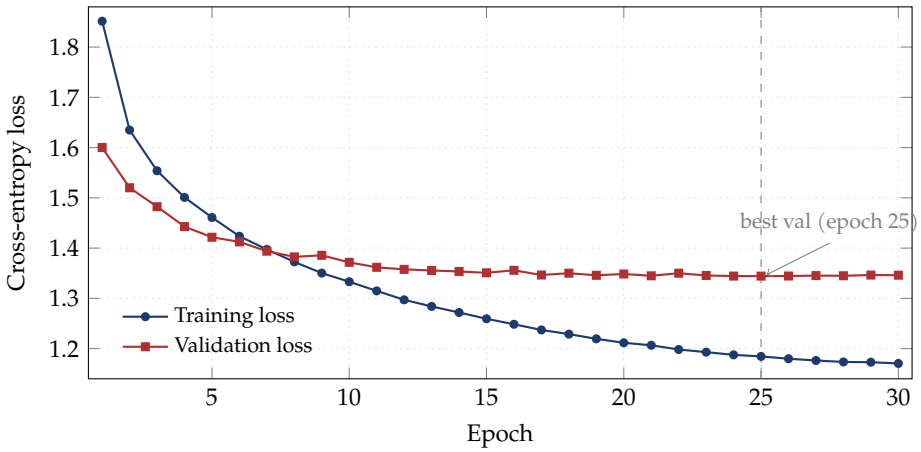


Figure 3. Training and validation loss over 30 epochs (seed 42). The validation loss plateaus after roughly epoch 17 and is lowest at epoch 25, with no subsequent rise; the epoch-25 checkpoint is retained.

4.4. Evaluation Metrics

To evaluate the effectiveness of PassT5 in predicting password transformations, we adopt the **Success Rate @ K (SR@K)** as our primary evaluation metric. This metric measures the proportion of held-out test samples for which the true tweaked password appears within the top-K candidates generated by the model. $SR@K$ is defined in Equation (9). A prediction is considered successful if the true tweaked password y_i appears anywhere within the set of up to K distinct generated candidates C_i^K, regardless of its rank within that set. We evaluate PassT5 at three different values of the guess count: $K \in \{10, 100, 1000\}$. Each value corresponds to a distinct attack scenario that reflects realistic conditions in the field of credential security:

- $\boldsymbol{K = 10}$: Represents a **strict online attack** scenario, where the attacker is limited to a small number of guesses before being blocked by account lockout mechanisms or rate-limiting policies. This is the most challenging scenario, as it requires the model's top predictions to be highly accurate.
- $\boldsymbol{K = 100}$: Represents a **moderate attack** scenario, applicable to systems with weaker rate-limiting or when the attacker has multiple attempts across different sessions. This scenario evaluates whether the model can capture common transformation patterns.

- ***K* = 1000**: Represents an **offline attack** scenario, where the attacker has obtained a leaked password database and can perform an unlimited number of guesses without triggering online defences. Importantly, ***K* = 1000** has been adopted as the standard evaluation count in most prior work on credential tweaking attacks. We therefore include it to allow our results to be placed alongside figures reported in prior work on 4iQ, while noting (Section 5.3) that these come from different setups.

We choose *SR@K* over alternative metrics (such as BLEU score or perplexity) for two main reasons. First, *SR@K* directly reflects the **practical effectiveness** of the model in real-world password guessing attacks—it answers the exact question a security analyst would ask: How many tweaked passwords can the model successfully predict within *K* attempts? Second, *SR@K* is widely adopted in prior password guessing research, enabling comparison with results reported in prior work such as Pass2Path [5], PassBERT [4], and Pass2Edit [6]. By reporting results at all three values of *K*, we provide a comprehensive evaluation that captures PassT5's performance across the full spectrum of practical attack scenarios—from tightly constrained online attacks to unrestricted offline guessing.

5. Results

This section evaluates PassT5 from several angles. We start with its overall success rate and how it compares with related work, then probe the robustness of that result on an unfiltered all-pairs test set and on users held out from the training pool, and show how the same model can serve defensively as a password-reuse checker. We then break the results down by transformation type and edit distance, walk through qualitative examples, look at performance across password lengths, and close with ablation studies that isolate the contributions of pre-training, tokenisation, and the sampling and decoding choices.

Two conventions hold throughout. First, the overall success rates and the comparison with related work are averaged over three independent training seeds (42, 123, and 456) and reported as mean ± standard deviation. The remaining per-category analyses—transformation type, edit distance, qualitative examples, and password length—are considerably more expensive to repeat for every seed, so we compute them from a single checkpoint (seed 42) on the same 5000-pair subset used for the headline result. This single run's overall SR@1000 (63.10%) is close to the three-seed mean, so it is representative of PassT5's averaged performance; the figure we cite as PassT5's overall performance remains the three-seed mean reported in Section 5.2. Second, wherever we report a success rate for a subgroup we accompany it with a 95% confidence interval computed using the Wilson interval [30], which remains well-behaved even when a rate is close to 0% or 100%—as it is for the strongest transformation categories (e.g., head truncation at 98.3%).

5.1. Evaluation Setup

We evaluate on the held-out test partition described in Section 4.3.2 (17,280 user-level test pairs). To account for training variance, we repeat the full training and evaluation pipeline across three independent random seeds (42, 123, 456) and report the mean ± standard deviation. Because building the ranked candidate list requires sampling up to 6000 candidates per input, evaluating the entire test set is computationally prohibitive; we therefore report SR@K on a fixed random subset of 5000 test pairs selected at the user level (2136 users; subset seed = 42)—whole users are drawn at random until the target size is reached, so no user's pairs are split across the subset boundary—held constant across all seeds so that the reported variation reflects model randomness rather than differences in the evaluated examples.

5.2. Overall Performance

On this evaluation subset, PassT5 achieves the following success rates:

- SR@10: 44.69% ± 0.32
- SR@100: 56.51% ± 0.11
- SR@1000: 62.80% ± 0.39

SR@K rises by 11.82 percentage points from $K = 10$ to $K = 100$ and by a further 6.29 points to $K = 1000$. The standard deviation across seeds is below 0.4 points at every K.

As a control, a popularity baseline that ignores the input and always guesses the most frequent training passwords—evaluated on the same 5000-pair subset used for PassT5—reaches 0.68%, 1.40%, and 2.02% at $K = 10$, $K = 100$, and $K = 1000$, respectively (95% CI [1.67–2.45] at $K = 1000$). As a second, input-dependent control, we evaluate a rule-based baseline in the spirit of earlier transformation attacks [10]: for each leaked password we generate candidates from a fixed catalogue of hand-crafted rules—case changes, common digit and symbol suffixes and prefixes, leet substitutions, and short truncations (about 180 candidates per input)—and check the target against them. On the same subset, with the rules applied in order of how often each produces the correct tweak in the training data, it reaches 36.72%, 40.72%, and 41.26% at $K = 10$, $K = 100$, and $K = 1000$: far above the popularity baseline—hand-crafted rules do capture real tweak patterns—but below PassT5 at every budget. The ordering matters only at small budgets: applying the same catalogue in a fixed, hand-chosen order instead gives 13.18% and 15.68% at $K = 10$ and $K = 100$, with the $K = 1000$ value unchanged at 41.26% (the candidate set is identical; only its ordering differs). Second, replacing our hand-crafted catalogue with Hashcat's standard `best64` rule set [2] (77 rules) reaches 9.38%, 23.40%, and 23.40%—below our catalogue at every budget—which confirms that the baseline we compare against is already a strong hand-crafted attack rather than a weak strawman. PassT5 exceeds all of these variants at every budget; the frequency-ordered catalogue is listed in Appendix B.

Figure 4 plots all three curves across the guess budgets.

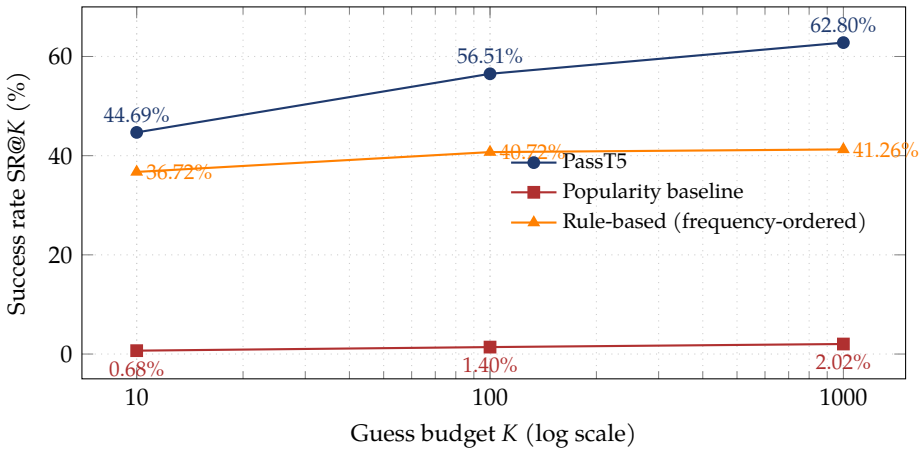


Figure 4. Success rate SR@*K* of PassT5 (mean over three seeds) versus the popularity and rule-based baselines across guess budgets.

Because candidates are sampled independently, repeated draws frequently reproduce the same strings. We therefore deduplicate each candidate set and rank the resulting unique candidates by their sampling frequency, computing SR@K over the top-K unique candidates (so that SR@10 ≤ SR@100 ≤ SR@1000 by construction). This removes the duplicate inflation inherent in naive sampling. It also exposes a related property of the model: for many inputs it cannot produce as many as 1000 distinct plausible tweaks. On this subset it yields on average about 730 unique candidates per input (ranging from 1 to over 1000), and 57.8%

of inputs produce fewer than 1000. SR@1000 should therefore be read as the success rate within up to 1000 unique candidates (mean ≈ 730 per input), rather than within a full 1000: where the model yields fewer than 1000 distinct tweaks the list is correspondingly shorter rather than padded, so SR@1000 reflects only the model's own generations.

5.3. Comparison with Prior Work

We next place PassT5 against prior work. Other models such as RFGuess [16], PassGAN [17], and GENPass [18] report strong results but on different datasets (e.g., RockYou, Neopets), which limits direct comparability; Table 2 therefore reports only models evaluated on the 4iQ dataset, so that the comparison is at least anchored to a common data source. These prior-work figures are not rerun under our setup—each is taken from a published source, as indicated in the Source column (the model's own paper, or a later paper's re-run using that model's public implementation). The listed models were all tested on 4iQ but may differ in data sampling, pre-processing, and train/test partitioning, so the table should be read as a comparison with previously reported results on the same dataset rather than a strictly controlled comparison under identical conditions.

Table 2. PassT5 performance compared with state-of-the-art password transformation models.

Study	$K = 10$	$K = 100$	$K = 1000$	Source	Evaluation Setup
PassBERT [a]	12.68%	15.71%	17.96%	From paper	80/20, held-out 20% of 4iQ (100k accounts)
PassTrans [b]	13.13%	18.41%	23.52%	From paper	80/20 (4iQts), 100k users
Pass2Path [b]	8.07%	11.71%	14.83%	Re-run (by PassTrans)	80/20 (4iQts), 100k users
Pass2Edit [c]	10.25%	14.81%	17.82%	From paper (via PointerGuess)	80/20, 20k-pair test subsample
PointerGuess [c]	10.49%	15.70%	20.43%	From paper	80/20, 20k-pair test subsample
PassT5 (Our Work)	**44.69% ± 0.32**	**56.51% ± 0.11**	**62.80% ± 0.39**	This work (3-seed re-run)	user-level 70/15/15, tweak-only ($d_{\text{norm}} \leq 0.5$), 5k test subset

[a] Xu et al. (Fudan University), USENIX Security 2023—a different research group from Pass2Edit/PointerGuess below. The paper samples 100,000 accounts from the held-out 20% of 4iQ for this evaluation. [b] Both rows from PassTrans's Table 2 (same 4iQts split); Pass2Path's number is PassTrans's authors' own re-run of Pass2Path's public model, not a number from Pass2Path's original paper. Full dataset: ∼146M paired users. [c] As cited in PointerGuess. Both papers state a 116.8M/29.2M train/test split on 4iQ, then randomly subsample 20,000 test pairs when a test set exceeds that size; the exact per-K breakdown for the 4iQ scenario alone is not given as a standalone table in either paper.

Table 2 places PassT5 alongside five prior methods from the credential-tweaking literature. Across all three guess budgets, PassT5 reports the highest success rate—44.69% at $K = 10$, 56.51% at $K = 100$, and 62.80% at $K = 1000$—ahead of the strongest prior method at each point (PassTrans, at 13.13%, 18.41%, and 23.52%, respectively).

These margins are large, and we think they should be read with care rather than as evidence that PassT5 is straightforwardly the stronger model. Each prior method was evaluated on its own 4iQ subset, whereas our numbers come from a test set we reconstructed with a normalised edit-distance filter (Section 4.3.2) that keeps only pairs representing a genuine tweak. That filter removes the unrelated password pairs an all-pairs construction introduces, so our evaluation set is plausibly more learnable than the test sets used in prior work: by design, every pair in it is a real modification. Part of the gap therefore reflects a cleaner, tweak-only evaluation and not model capability alone. We present this comparison as anchored to a common data source (4iQ) rather than as a comparison conducted under identical evaluation conditions. To see how much of the gap this filter explains, we also ran PassT5 on an unfiltered all-pairs test set, built the way earlier work builds theirs (Section 5.4). There it reaches about 18% at $K = 1000$, a little below PassTrans's 23.52%. So most of the lead in Table 2 comes from the tweak-only filter, not from PassT5 being a stronger model across the board; where it clearly wins is on the task it was built for.

The difference in evaluation-set size also follows from our protocol rather than a reduced test. Prior work on 4iQ uses an 80/20 train/test split, whereas we adopt a user-level 70/15/15 split that reserves a separate 15% validation partition for model selection and early stopping and keeps all of a user's pairs within a single partition to prevent cross-partition leakage. This yields a 17,280-pair held-out test set, from which we draw the 5000-pair user-level subset used for SR@K for computational tractability—much as the prior methods above report figures on a sampled 20,000 to 100,000 pairs rather than their full test split. Because SR@K is a proportion rather than a count, the size of this subset affects only the width of the confidence interval, not the expected rate; the across-seed standard deviation ($\leq$0.39) confirms the estimate is stable.

5.4. Evaluation on an Unfiltered All-Pairs Test Set

Our main test set keeps only pairs within a normalised edit distance of 0.5, so that every pair is a real tweak. This fits the task, but it also leaves out the harder, unrelated pairs that an all-pairs construction includes, which makes our test set easier than the ones earlier work uses and inflates the gap in Table 2. To see how much of that gap is down to the filter rather than the model, we re-ran PassT5 on an unfiltered all-pairs test set, built the way prior work builds theirs: every ordered same-user pair on the same test users, with no edit-distance filter. Table 3 reports what we found. On the all-pairs set PassT5 reaches 11.54%, 16.40%, and 18.36% at $K = 10$, 100, and 1000. That is about three times the popularity baseline on this set (6.02% at $K = 1000$)—itself far stronger here than on the tweak-only task, since unrelated same-user passwords are often common ones—but on this unfiltered set it no longer leads prior work: PassTrans reports 13.13%, 18.41%, and 23.52% at the same budgets, so PassT5 sits a little below PassTrans across all three. Most of the large margin on the filtered task therefore comes from the clean, tweak-only evaluation, not from PassT5 being a stronger model everywhere. Where it clearly helps is on the task it was built for—real tweaks—where it reaches 63.10% at $K = 1000$.

Table 3. PassT5 on the unfiltered all-pairs test set, next to the filtered tweak-only set (seed 42, $n = 5000$). PassTrans is the closest comparable prior result on an all-pairs setting.

Setting	SR@10	SR@100	SR@1000
Popularity baseline (all-pairs)	1.96%	4.96%	6.02%
PassTrans (reported)	13.13%	18.41%	23.52%
PassT5—all-pairs (unfiltered)	11.54%	16.40%	18.36%
PassT5—tweak-only (filtered)	45.04%	56.52%	63.10%

5.5. Generalisation to Users Outside the Training Pool

Our main test set is drawn, like the training set, from the first 50,000 users of the 4iQ leak. To check that the results are not specific to that slice of users, we built a second evaluation set of 5000 tweak pairs from users sampled outside the first 50,000—accounts that play no part in the main split—and scored the same seed-42 model on it under the identical procedure (top-k sampling, $k = 50$, $\tau = 0.9$; SR@K over frequency-ranked unique candidates).

Table 4 compares the two. On the held-out users PassT5 reaches 41.29%, 53.83%, and 60.25% at $K = 10$, 100, and 1000, a little below the 45.04%, 56.52%, and 63.10% of the matched seed-42 run on the main set—a gap of about three points at every budget. Performance therefore carries over to users the model never saw during development, with only a small drop that is consistent with these accounts forming a slightly different and marginally harder sample rather than with any overfitting to the original pool.

Table 4. PassT5 (seed 42) on the main test users (within the first 50,000) versus a fresh set of users sampled outside the first 50,000; each set is 5000 tweak pairs scored with the same generation and ranking procedure.

Test Users	SR@10	SR@100	SR@1000
Main (within first 50,000)	45.04%	56.52%	63.10%
Outside first 50,000 (held-out)	41.29%	53.83%	60.25%

5.6. Defensive Use as a Password-Reuse Checker

The same model can be read defensively rather than offensively. A service that already knows a user's leaked password can run PassT5 on it when that user sets a new password, and warn or block if the proposed password appears among the model's top-K generations—that is, if it is a predictable tweak of the compromised credential. Used this way PassT5 becomes a reuse checker, and the quantity that matters is no longer only how many genuine tweaks it catches, but how often it wrongly flags a password the user deliberately chose to be different.

Measuring that second quantity requires pairs that are not tweaks, which the filtered tweak-only set excludes by construction. We therefore reuse the unfiltered all-pairs set of Section 5.4, which contains both kinds of pair, and label each pair by the normalised edit distance d between its two passwords: $d \leq 0.5$ marks a genuine tweak (a password the checker should block) and $d > 0.5$ marks an essentially unrelated choice (one it should not). Of the 5000 pairs, 1538 are tweaks and 3462 are unrelated. For each pair we take the model's existing output—whether it generated the target, and at what rank—and treat the proposed password as flagged when that rank is at most K. No re-generation is needed: both rates below are read off the same runs as Section 5.4.

Table 5 reports the outcome. On genuine tweaks the detection rate grows with the budget, reaching 56.76% at $K = 1000$, while the false-rejection rate on unrelated choices stays very low—0.26%, 0.81%, and 1.30% at $K = 10$, 100, and 1000. A budget of 1000 generations therefore lets the checker catch most lazy reuse while mistakenly rejecting only about one in seventy-seven genuinely different passwords. Because a deployed service also enforces a composition policy, we recompute the false-rejection rate on the unrelated pairs whose target satisfies one: it remains 1.34% under a length- ≥ 8 rule and 2.07% under a rule that additionally requires a letter and a digit, so policy screening does not frequently reject valid passwords. This positions PassT5 not only as an attack model but as a low-false-alarm defensive screen.

Table 5. PassT5 used as a defensive reuse checker on the all-pairs set (seed 42, $n = 5000$): detection on genuine tweaks ($d \leq 0.5$, 1538 pairs) versus false rejection on unrelated choices ($d > 0.5$, 3462 pairs). Lower false rejection is better.

Rate	K = 10	K = 100	K = 1000
Detection (tweaks, should block)	36.93%	51.50%	56.76%
False rejection (unrelated, should allow)	0.26%	0.81%	1.30%
False rejection under a composition policy ($K = 1000$):			
length ≥ 8	–	–	1.34%
length ≥ 8, with letter and digit	–	–	2.07%

5.7. Transformation-Type Analysis

To see what kinds of password changes PassT5 handles well or poorly, we sort every leaked/tweaked pair (X, Y) in the test set into one of eight categories based on how Y relates to X: append means Y is X with characters added at the end (e.g., `pass1` → `pass123`); prepend is the same but at the start; truncate (tail/head) means the opposite—characters

removed from the end or the start; case change means X and Y have the same characters but different capitalisation; substitution (≤2 chars) means X and Y are the same length and differ in at most two character positions; reorder/same-length covers same-length pairs that differ in more than two positions; and heavy/unrelated is the fallback for any pair that does not fit the rules above.

Table 6 reports SR@K by transformation category. Every category contains at least 297 pairs. At $K = 1000$, the categories rank as follows: head truncation 98.3% ($n = 641$), tail truncation 94.7% ($n = 606$), case change 91.5% ($n = 646$), prepend 68.0% ($n = 643$), substitution 50.8% ($n = 297$), append 46.2% ($n = 634$), heavy/unrelated 32.6% ($n = 1190$), and reorder 26.5% ($n = 343$), against an overall rate of 63.10%. The same ordering holds at $K = 10$ and $K = 100$.

Table 6. Success rate (%) by transformation type (seed 42, $n = 5000$).

Transformation Type	Pairs (n)	$K = 10$	$K = 100$	$K = 1000$	95% CI ($K = 1000$)
Truncation (head removed)	641	92.5%	97.2%	98.3%	[97.0–99.0]
Truncation (tail removed)	606	77.4%	90.3%	94.7%	[92.6–96.2]
Case change only	646	77.2%	87.9%	91.5%	[89.1–93.4]
Prepend (prefix added)	643	48.2%	60.7%	68.0%	[64.3–71.5]
Substitution (≤2 chars)	297	13.8%	38.0%	50.8%	[45.2–56.5]
Append (suffix added)	634	17.0%	34.2%	46.2%	[42.4–50.1]
Heavy/unrelated rewrite	1190	16.5%	25.1%	32.6%	[30.0–35.3]
Reorder/same-length	343	10.5%	20.1%	26.5%	[22.1–31.4]
Overall (this run)	5000	45.04%	56.52%	63.10%	–

5.8. Edit Distance Analysis

For every test pair where PassT5's Top-1000 candidates missed the true password, we look at how close the model actually got. Specifically, we compute the edit distance—the minimum number of single-character insertions, deletions, or substitutions needed to turn one string into another, also known as Levenshtein distance—between the true password and the closest of the model's generated candidates. A small edit distance means the model's best guess was structurally very close to the real password even though it did not match exactly.

Table 7 measures proximity to success in the 1845 test pairs where PassT5 did not place the true password among the top-1000 unique candidates. The right password was within three single-character modifications of the closest generated candidate in 90.5% of failures (1670 of 1845, 95% CI [89.1–91.8]).

Table 7. Distribution of failures by edit distance to closest candidate (seed 42, $n = 5000$).

Min. Edit Distance to Target	Miss Count	% of Misses	95% CI
1	896	48.6%	[46.3–50.8]
2	491	26.6%	[24.6–28.7]
3	283	15.3%	[13.8–17.1]
≥4	175	9.5%	[8.2–10.9]
Total Misses	1845	100%	–

5.9. Qualitative Examples

Table 8 presents five examples from the seed-42 run. The "Model's Top Prediction" column reports the rank-1 generated candidate; for the deep-match and miss rows, this is not necessarily the candidate closest to the target. Cases 1–3 are rank-1 successes (append, case change, head truncation). Case 4 is an append recovered at rank 367, with a different

string at rank 1. Case 5 is a miss (case change plus an internal digit edit) whose closest generated candidate is one edit from the target.

Table 8. Qualitative examples of PassT5 predictions (seed 42, $n = 5000$).

Input (X)	Target (Y)	Model's Top Prediction	Outcome
cocogirl	cocogirl1	cocogirl1	success @ rank 1 (append)
eQoq9kCT6E	eqoq9kct6e	eqoq9kct6e	success @ rank 1 (case change)
qq466922025	466922025	466922025	success @ rank 1 (head truncation)
412363	412363Q	0412363	success @ rank 367 (append; deep match)
Gamenut89	gamenut188	gamenut89	Miss (min. edit dist. = 1)

5.10. Password Length Analysis

We examine performance by input-password length, dividing test pairs into Short (≤8 characters, $n = 2596$), Medium (9–12 characters, $n = 1989$), and Long (13+ characters, $n = 415$).

As shown in Figure 5, SR@1000, computed within each length group, is 59.59% (Short, 95% CI [57.7–61.5]), 67.27% (Medium, [65.2–69.3]), and 65.06% (Long, [60.4–69.5]). The medium group's Wilson interval does not overlap the short group's, so medium-length inputs are recovered at a reliably higher rate than short ones; the long group, with far fewer pairs, sits between the two and its wider interval overlaps both.

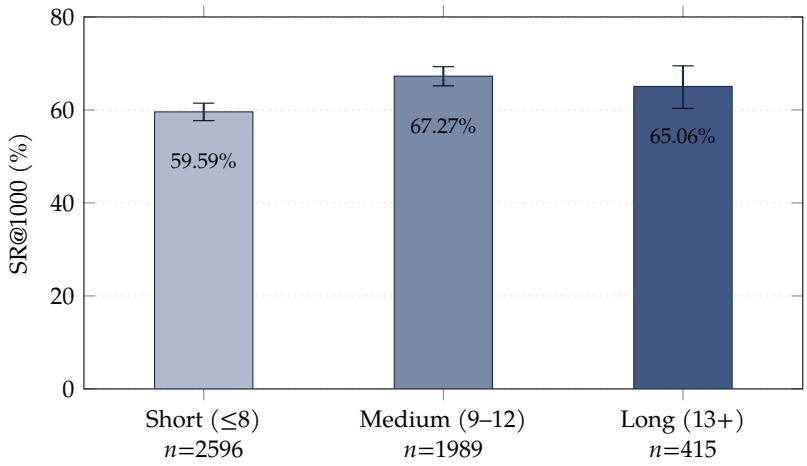


Figure 5. SR@1000 of PassT5 by password length category (seed 42). Error bars show 95% Wilson confidence intervals, computed within each length group.

5.11. Ablation Studies

To understand where PassT5's performance comes from and to justify our main design choices, we run four ablations: the effect of pre-training, the tokenisation scheme, the sampling hyperparameters, and the decoding strategy. All use a single seed (42) on a frozen seed-42 subset of the test set, held fixed across all ablations for exact reproducibility; absolute SR values therefore differ from Section 5.2 by at most a few tenths of a point. The pre-training and tokeniser ablations use the full 5000-pair subset; the sampling and decoding studies use smaller fixed subsets of it (500 and 300 pairs) for tractability.

5.11.1. Effect of Pre-Training

PassT5 fine-tunes a T5-Small checkpoint pre-trained on general text. To isolate the contribution of that pre-training, we train an identical T5-Small architecture from scratch (random initialisation, no pre-trained weights) on the same pairs, under the same user-level

split, hyperparameters, and seed (42), and evaluate it with the same SR@K procedure on the fixed subset. As Table 9 shows, pre-training adds roughly eleven to twelve percentage points at every budget (63.10% vs. 50.72% at $K = 1000$). Notably, the from-scratch model still reaches 50.72% SR@1000—far above the 2.02% popularity baseline—so the architecture learns genuine transformations even without pre-training, while pre-training provides a substantial additional gain consistent with transfer from general-text sequence modelling. Both models use the same learning rate (5×10^{-5}), which was chosen for fine-tuning a pre-trained checkpoint and may be suboptimal for from-scratch training, which often benefits from a larger learning rate or a longer schedule; the pre-training gain reported here should therefore be read as an upper bound.

Table 9. Effect of pre-training: PassT5 fine-tuned from a pre-trained checkpoint vs. the same architecture trained from scratch (seed 42, $n = 5000$, fixed subset).

Model	SR@10	SR@100	SR@1000
T5-Small from scratch (no pre-training)	33.76%	44.64%	50.72%
PassT5 (pre-trained + fine-tuned)	**45.04%**	**56.52%**	**63.10%**
(pre-training gain)	+11.28	+11.88	+12.38

5.11.2. Byte-Level Tokeniser (ByT5)

Section 4.2.1 noted that PassT5's SentencePiece tokeniser, built for general English text, cannot reproduce eight ASCII characters that occur in passwords. To test whether a byte-level tokeniser removes this limitation and how it affects overall performance, we fine-tune ByT5-Small—a byte-level encoder–decoder—under identical settings (same pairs, user-level split, hyperparameters, seed 42; top-k sampling with $k = 50$, $\tau = 0.9$) and score it with the same SR@K procedure on the fixed subset. We note that ByT5-Small is larger than T5-Small (about 300 M versus about 60 M parameters, as byte-level modelling needs more layers to compensate for shorter tokens), so part of the improvement reported below may reflect this additional capacity rather than tokenisation alone.

Table 10 reports the comparison. ByT5 improves SR@K at every budget by roughly five percentage points. The gain holds across all transformation types and is largest on edits that add or change individual characters, such as appended suffixes, where a byte-level model can compose any character directly. ByT5 also produces more distinct candidates (mean 820 vs. 730; only 38.8% of inputs short of 1000, vs. 57.8%), which partly eases the candidate-diversity limitation of Section 6.4. Most directly, it closes the coverage gap: of the 19 subset targets containing one of the eight characters SentencePiece cannot emit, ByT5 recovers 11 (57.9%) within $K = 1000$—several at rank 1—whereas PassT5 cannot produce these targets by construction. A byte-level tokeniser is thus a practical remedy for the coverage limitation, but at a substantial efficiency cost: ByT5-Small has roughly five times the parameters of T5-Small (about 300 M vs. about 60 M) and decodes longer byte sequences, making candidate generation several times slower. We therefore retain T5-Small as our main model—its favourable accuracy-to-cost trade-off matters under the up-to-6000-sample budget per input—and present ByT5 as a promising direction when byte-level coverage or peak accuracy is the priority.

Table 10. PassT5 (subword) vs. ByT5 (byte-level) under identical settings (seed 42, $n = 5000$, fixed subset).

Model (Tokeniser)	SR@10	SR@100	SR@1000
PassT5 (SentencePiece subword)	45.04%	56.52%	63.10%
ByT5 (byte-level)	**49.14%**	**61.50%**	**68.62%**

5.11.3. Sampling Hyperparameters

To check sensitivity to the sampling settings, we vary the temperature τ (at $k = 50$) and the top-k cutoff (at $\tau = 0.9$) on a 500-pair subset (Table 11, Figure 6). Performance is stable across the grid, varying by at most about three points. Temperature mainly controls candidate diversity—the mean number of distinct candidates rises from 281 at $\tau = 0.7$ to 797 at $\tau = 1.1$—so higher temperatures slightly raise SR@1000 (from 56.4% to 60.4%) by filling larger budgets, while SR@100 peaks at $\tau = 0.9$. The top-k cutoff has little effect ($k \in \{25, 50, 100\}$ differ by at most one point). Our default ($k = 50$, $\tau = 0.9$) gives the highest SR@100 on this subset, although the differences across settings are small relative to their confidence intervals, and it stays within about one point of the best setting at every budget—consistent with a balanced, near-optimal choice rather than a tuned outlier.

Table 11. Sampling-hyperparameter sensitivity on a 500-pair subset. At this size the 95% Wilson interval is about ±4 points, so the differences across settings are not statistically significant.

k	τ	Avg. Unique	SR@10	SR@100	SR@1000
50	0.7	281	41.4%	52.8%	56.4%
50	0.8	410	43.2%	54.6%	58.6%
50	**0.9**	**548**	**43.6%**	**55.0%**	**59.4%**
50	1.0	682	44.4%	54.4%	60.2%
50	1.1	797	44.4%	54.8%	60.4%
25	0.9	458	43.2%	54.4%	59.2%
100	0.9	606	43.4%	53.4%	60.4%

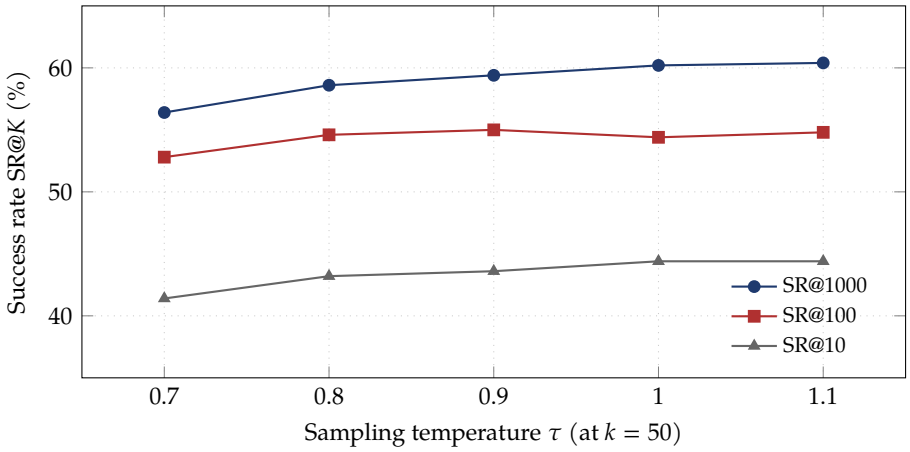


Figure 6. Sensitivity of SR@K to sampling temperature τ (top-k fixed at $k = 50$; $n = 500$). Our default $\tau = 0.9$ is at or near the optimum at every budget.

5.11.4. Decoding and Candidate Ranking

Most prior credential-tweaking models decode with beam search, whereas PassT5 uses top-k sampling. To justify this choice, we compare greedy, beam-search, diverse beam search, and sampling decoding on a common 300-pair subset (Table 12); each rate is reported with a 95% Wilson confidence interval. Greedy decoding recovers 27.0% of targets with its single best guess. At matched budgets, sampling matches or exceeds beam search—44.67% vs. 43.67% at $K = 10$ and 56.33% vs. 52.00% at $K = 100$—and both clearly beat diverse beam search (38.33% and 42.67%), whose diversity penalty forces the beams toward lower-probability sequences and hurts recovery. Beam search and diverse beam search do not scale to $K = 1000$: a beam width of 1000 is computationally prohibitive and yields near-duplicate, low-diversity sequences, whereas sampling continues to 60.33% at $K = 1000$ (broadly consistent with the full-subset result in Section 5.2, up to the sampling

variance expected on 300 pairs). Sampling is therefore the appropriate decoder for the large-budget, offline setting this work targets.

Table 12. Decoding-strategy comparison on a common 300-pair subset, with 95% Wilson confidence intervals.

Decoding	SR@1	SR@10	SR@100	SR@1000
Greedy	27.0%	—	—	—
Beam search	—	43.67%	52.00%	— *
Diverse beam search	—	38.33%	42.67%	— *
Top-k sampling (ours)	—	**44.67%**	**56.33%**	**60.33%**

"—" marks budgets not reported for a decoder: greedy yields a single guess, so it is listed only at $K = 1$, and sampling is the large-budget method. * Beam and diverse beam at $K = 1000$ are computationally prohibitive. 95% Wilson CIs at $K = 100$: sampling [50.7, 61.8], beam [46.4, 57.6], diverse beam [37.2, 48.3].

We also check whether ordering the generated candidates by sampling frequency—the choice used throughout—loses anything relative to ordering them by the model's own log-probability. On a 2000-pair subset we score every generated candidate under the model with a teacher-forcing pass over the full vocabulary and recompute SR@K under both orderings. The two are statistically indistinguishable at every budget: frequency ranking reaches 45.85%, 57.55%, and 63.50% at $K = 10, 100, 1000$, against 44.70%, 56.80%, and 63.55% for log-probability ranking, with overlapping 95% Wilson confidence intervals throughout. This matches the equivalence anticipated by our candidate-generation procedure—sampling frequency is a Monte-Carlo estimate of the model's probability—so we retain frequency ranking, which needs no additional scoring pass.

6. Discussion

6.1. Does the Model Learn Transformations, or Just Guess Common Passwords?

A natural worry with any model trained on leaked passwords is that it may not be learning how people change their passwords at all, and may instead be leaning on the passwords that simply occur most often in the data. Our results show it is the former. At $K = 1000$, PassT5 finds 62.80% of the tweaked passwords, while a popularity baseline that ignores the leaked password and only guesses the most common ones finds just 2.02%. The gap is the same at every budget: 44.69% against 0.68% at $K = 10$, and 56.51% against 1.40% at $K = 100$. So guessing by frequency alone works on about one case in fifty, while PassT5 works on almost two in three. The only difference between the two is that PassT5 looks at the leaked password, so this gap shows that its success comes from learning how each password was changed, not from guessing passwords that are popular in general. A rule-based attack makes the same point from the other side. A fixed catalogue of common transformations recovers 41.26% of targets at $K = 1000$—well above frequency guessing, since these rules do encode genuine tweak behaviour—yet PassT5 reaches 62.80%. The model therefore captures transformation structure that a hand-written rule set misses, not just the most common edits.

A second, independent sign points the same way. During training, the validation loss falls steadily, plateaus after roughly epoch 17, and then stays flat rather than rising (Figure 3), and early stopping retains the checkpoint with the lowest validation loss, measured on users unseen in training. A model that had merely memorised frequent training passwords would not continue to generalise to held-out users in this way. We read the training loss only as evidence of stable, non-overfitted convergence; the model's practical ability is measured separately by SR@K, and it is that metric—not the loss—on which all our performance claims rest. The loss likewise does not explain why particular transformation types fail; that question is addressed by the transformation-type and edit-distance analyses in Sections 5.7

and 5.8. A third line of evidence comes from pre-training itself. If PassT5 were mostly recalling frequent passwords absorbed during general-text pre-training, removing that pre-training should collapse its performance toward the popularity baseline. It does not. Trained from scratch on the same pairs, the identical architecture still reaches 50.72% SR@1000 (Section 5.11.1)—far above the 2.02% popularity baseline—so the sequence-to-sequence formulation learns genuine transformations on its own. Pre-training then adds a further twelve points. PassT5's performance thus decomposes into two separable contributions: a strong task formulation and a substantial transfer benefit from pre-training, neither of which reduces to guessing common passwords.

6.2. Which Changes the Model Handles Well

PassT5's performance varies with the type of change shown in Table 6, and where each one falls turns out to be telling: the edits it handles are those that follow a set rule, and the ones it misses are those that do not. It is strongest when the target keeps the same characters and only drops some or changes their case. Head truncation reaches 98.3% at $K = 1000$, tail truncation 94.7%, and case change 91.5%, all well above the overall rate of 63.10%. In these cases the target is fixed once the rule is known, so a model that has learned the rule can reproduce it almost every time. The very high success on head truncation deserves a caveat. These targets are overwhelmingly short numeric strings (close to 90% are all digits) carrying a brief alphabetic prefix—often a single character, as in `w1044114814`→`1044114814`. Recovering them only requires stripping a short, highly predictable prefix, so the near-perfect rate reflects this common prefixing habit in the data rather than an intrinsically difficult edit.

The model does less well when it has to add new characters. Prepend is just above average at 68.0%, while substitution and append are below it at 50.8% and 46.2%, because here the model has to work out not just where the change happens but also which characters were added or swapped. The hardest changes are the ones with no clear rule: heavy or unrelated rewrites reach 32.6%, and same-length reorderings only 26.5%.

The adjustments it handles best are the simple ones—users shorten passwords, change capitalisation, or add a couple of characters—and its accuracy drops as an edit forces it to generate more new content. If a model were only picking up on simple surface regularities, it would be hard to explain why it follows this difficulty so closely.

6.3. How the Model Fails: Near-Misses vs. Hard Cases

The misses are worth a closer look as well. Whenever the true password did not appear among the top-1000 candidates, we used edit distance to measure how far the model's nearest guess landed from it (Table 7). In 90.5% of the 1845 misses, the closest candidate was within three single-character edits of the true password. In most failures, then, the model found the right kind of change and only got a few characters wrong, rather than missing the password completely. To calibrate these distances, the input and the true target are themselves a median of 3 edits apart among the missed pairs, so the closest candidate—a median of 2 edits from the target—is typically closer to the true password than the original input. Even the failures therefore converge toward the target rather than lingering at the distance of the leaked password.

Only 9.5% of the misses are four or more edits away. In Table 6, the lowest score in the table was recorded for heavy and reordered edit cases, which are really difficult, as the leaked and structurally modified passwords differ and there is no rule relating them. That most failures cluster near the target, and only a few land far from it, is one more sign that the model has picked up genuine transformation rules. It also indicates something useful: since many misses are only a character or two away, it is possible to recover some of these

misses by expanding the top candidates with minor character-level edits. We have not built or tested this, so we note it as a possible future step.

6.4. The Limit of Candidate Diversity

One feature of how PassT5 generates guesses is worth discussing. Because we keep only distinct candidates, the model often runs out of new guesses before reaching 1000. On the evaluation subset it produces about 730 distinct candidates per input on average, and for 57.8% of inputs it produces fewer than 1000. Where it falls short, the ranked list is simply shorter than 1000 rather than padded. SR@1000 should therefore be read as success within up to 1000 distinct model-generated guesses, not within a full 1000. This only affects the largest budget: at $K = 10$ and $K = 100$ the model almost always produces more than enough distinct candidates, with just 0.9% of inputs falling short of 10 and 4.9% short of 100.

This is a real limit of stochastic sampling: for many inputs the model is confident about a smaller set of likely tweaks and keeps returning them, rather than offering a wide range of distinct guesses. For the offline, high-budget setting this work targets, where an attacker can try many guesses, the diversity it does provide is still useful, and its strongest results come at the larger budgets. But the saturation caps how many genuinely different guesses the model can contribute at $K = 1000$, and closing this gap would need a generation method that produces more varied candidates while keeping them realistic tweaks of the input. A byte-level variant (Section 5.11.2) partially addresses this, producing more distinct candidates per input (about 820 vs. 730).

6.5. Positioning Relative to Prior Work

Table 2 shows PassT5 well ahead of the prior methods reported on 4iQ, at every guess budget. These methods were all tested on the 4iQ leak, but each used its own split and pre-processing, as is usual in this area, so the table is anchored to a common data source rather than a single shared protocol. Our own test set keeps only pairs that are real modifications, since credential tweaking is about how a password is changed, not about two unrelated passwords that happen to belong to the same user. That choice fits the task we set out to study, but it also makes our test set easier than the all-pairs sets earlier work uses. When we drop the filter and score PassT5 on all pairs, as in Section 5.4, it lands a little below PassTrans (about 18% against 23.52% at $K = 1000$). So the large margin in the table belongs to the tweak-only task; on the broader all-pairs setting PassT5 is no longer ahead of prior work.

Read together with the popularity baseline, one point stands out: PassT5's success comes from learning the transformation, not from guessing common passwords. It recovers far more tweaks than a frequency-only guesser at every budget—62.80% against 2.02% at $K = 1000$—and since the only thing that baseline lacks is the leaked password itself, the gap comes down to actually modelling the change.

6.6. Capabilities and Limitations

We close by summarising what PassT5 does well and where it falls short.

Its main strengths are threefold. First, it recovers a large share of tweaked passwords across all three guess budgets, making it useful both when guesses are tightly limited and when many are allowed. Second, it learns real transformation behaviour: it is strongest on the edits that follow a fixed rule—truncation, case changes, and prepends—and it beats a popularity baseline by a wide margin, showing that its success does not come from simply guessing common passwords. Third, even its failures are mostly near-misses, with 90.5% of missed targets lying within three single-character edits of a generated candidate.

Its main limitations are the following:

- **Limited candidate diversity at large budgets.** At $K = 10$ and $K = 100$ this is not an issue—the model almost always produces more than enough distinct candidates (only 0.9% of inputs fall short of 10 and 4.9% short of 100). At $K = 1000$, however, it often produces fewer than 1000 distinct candidates (about 730 on average, with 57.8% of inputs short of 1000), so for those inputs SR@1000 is measured over fewer than 1000 model-generated guesses rather than a full 1000.
- **Structurally divergent edits.** Heavy rewrites and same-length reorderings, which follow no simple rule, remain hard and account for most of the truly distant failures.
- **Tokeniser coverage.** PassT5's SentencePiece tokeniser, built for general English text, cannot reproduce a few ASCII symbols exactly. This affects very few passwords in our data, and it is specific to the subword tokeniser: our byte-level ablation (ByT5; Section 5.11.2) reproduces these characters, at the cost of longer sequences and slower generation.
- **Scope and comparison.** The main evaluation is filtered to genuine tweaks; for comparability we additionally report an unfiltered all-pairs evaluation (Section 5.4). The comparison with prior work is anchored to the 4iQ dataset rather than run under an identical protocol, and the per-category analyses are computed from a single seed.
- **Single corpus.** Our evaluation uses only the 4iQ/BreachCompilation corpus. Because credential tweaking requires same-user passwords observed across sites, single-site leaks such as RockYou and LinkedIn do not provide tweak pairs; evaluation on an independent multi-site compilation is left to future work.
- **Minimal threat model.** We assume a single-step attack from one known password per user. Richer and more realistic settings—multi-round iterative guessing, site-specific password-policy constraints, and fusion of multiple leaked passwords per user—are likely to affect attack success and are left to future work.
- **Defensive applications only partly evaluated.** We quantify the reuse checker in Section 5.6, but the remaining applications in Section 8 (context-aware strength estimation and informing online guessing limits) are presented as motivation and not evaluated, which would require separate validation.

Taken together, these results show that a sequence-to-sequence encoder–decoder is an effective way to model credential tweaking, while also marking the points—candidate diversity and structurally divergent edits—where the approach can still be improved.

7. Ethical Considerations

As this work concerns an offensive-security topic, we outline the precautions taken in handling the data and the intended use of our results. Our data is derived from the BreachCompilation corpus (the 4iQ leak), a publicly available breach and the same source used to build earlier datasets such as Pass2Path [5]. From this corpus we construct the leaked/tweaked password pairs ourselves (Section 4.3.2). Email addresses appear in the raw corpus only as a per-user key; we use them solely to group each user's passwords when forming pairs, after which no email or other identifying information is retained—each pair is tagged only with an anonymous integer user id. All processing and training were carried out offline on private compute resources, with no data shared with any third party. Neither the fine-tuned model nor the generated candidate passwords are released; the results we report are aggregate success rates over the held-out test set, and beyond the small set of illustrative examples in Table 8, no individual predicted passwords are disclosed.

8. Defensive Applications

Our work is defensively oriented. Because PassT5 models how users transform a leaked password, it could support defences that measure or reduce the predictability of

user-changed passwords. In the information-flow terms of Section 3, each of these defences is an attempt to bound the leakage from one of a user's passwords to another, rather than to prevent exact reuse alone. We outline three potential applications; the first—a reuse checker—we evaluate quantitatively in Section 5.6, while the other two we leave as directions that would each require separate validation.

- **Reuse-aware password policies.** A service that can link a user to a previously breached password could generate the top-K variants of that password and reject, or warn against, a new password that appears among them. Breached-password blocklists primarily target exact reuse [5], whereas our results show that, on the evaluated data, a substantial share of tweaked passwords are guessed within a small budget. We quantify the usability cost of such a policy in Section 5.6: at $K = 1000$ the checker flags 56.76% of genuine tweaks while wrongly rejecting only 1.30% of unrelated password choices (1.34–2.07% when the new password must also satisfy a composition policy).
- **Context-aware strength estimation.** The rank of a new password within the candidates generated from an old one could serve as a context-aware estimate of its guessability, complementing conventional strength meters that score a password in isolation, in the spirit of the personalised strength meter proposed by [5]. A password absent from the candidate list is not thereby shown to be strong; it was only not guessed within the generation budget.
- **Informing online guessing limits.** Our low-budget results (SR@10) suggest that, for users who tweak a breached password, many new passwords fall within a few guesses, supporting strict limits on online attempts for accounts linked to known breaches. These figures are conditioned on the evaluated password pairs and do not estimate the proportion of accounts at risk in a general population.

To limit the risk of misuse, we release neither the fine-tuned model nor the generated candidates, reporting only aggregate success rates. Our goal is to provide a basis for defences, not for attacks.

9. Conclusions

This research offers an improved model that leverages the Transformer T5 architecture to model credential tweaking across different accounts. Although prior research has established that deep learning models can identify behavioural patterns in password transformation, PassT5 demonstrates that Transformer-based designs provide an effective and efficient approach. Averaged across three independent seeds under a 70%/15%/15% train/validation/test split, the model achieved a mean 44.69% $\pm$ 0.32 SR@10, 56.51% $\pm$ 0.11 SR@100, and 62.80% $\pm$ 0.39 SR@1000 on the held-out test set of the 4iQ dataset. This SR@1000 is well above the highest figure previously reported on the 4iQ dataset (23.52% for PassTrans [12]), though, as discussed in Section 5.3, that prior figure comes from a larger sample under a different split, so the comparison is anchored to a common dataset rather than a controlled re-evaluation. On an all-pairs test set built as in earlier work, it instead scores about 18% (Section 5.4), a little below that prior figure—so the model's edge shows up on the tweak task it was designed for, not as a gain across the board. Overall, a pre-trained encoder–decoder is a strong fit for credential tweaking, with its clearest gains on genuine tweaks rather than on unrelated password pairs.

While this work is about password prediction, other security problems, such as analysing user behaviours, finding password reuse patterns, and identifying linked accounts, can also be solved using the Transformer T5 architecture. Analysing these patterns helps build stronger defensive systems and also identifies potential blind spots in attack surface modeling that attackers can exploit. To the best of our knowledge, PassT5 is the first work to adapt the T5 Transformer to credential-tweaking attack scenarios.

Building on the limitations identified in this study, we see three main directions for future work:

- **Improving candidate diversity.** Candidate diversity is an area that could be further improved for very large budgets. Because sampling saturates, for many inputs PassT5 produces fewer than 1000 distinct candidates. Hence, at $K = 1000$ the effective guess list is shorter than the budget, and a decoding strategy that yields more diverse but still realistic candidates could use the full budget to further increase SR@1000.
- **A password-oriented tokeniser.** The current tokeniser is designed for general English text; as such, there are gaps in reproducing some ASCII characters. Our ByT5 ablation (Section 5.11.2) confirms that a byte-level model closes this gap; a tokeniser purpose-built for passwords—rather than a general byte-level one—is a natural next step and would also let the model insert any character found in a password.
- **Managing structurally divergent edits.** The hardest cases are still heavy rewrites and same-length reordering. One way to model character position and ordering more explicitly would be to use a Transformer-based model with an additional corrector layer to help recover some heavy rewrites.

Author Contributions: Conceptualisation, D.A., A.A. (Abdullah Alshahrani) and A.A. (Adel Alshamrani); methodology, D.A., A.A. (Abdullah Alshahrani) and A.A. (Adel Alshamrani); software, D.A.; validation, D.A., A.A. (Abdullah Alshahrani) and A.A. (Adel Alshamrani); formal analysis, D.A.; investigation, D.A.; data curation, D.A.; writing—original draft preparation, D.A.; writing—review and editing, D.A., A.A. (Abdullah Alshahrani) and A.A. (Adel Alshamrani); visualisation, D.A.; supervision, A.A. (Abdullah Alshahrani) and A.A. (Adel Alshamrani). All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: This study uses the publicly circulated BreachCompilation (4iQ) corpus. To limit the risk of misuse, the derived password pairs, the fine-tuned model, and the generated candidate passwords are not publicly released. They are available from the corresponding author upon reasonable request for verification and peer-review purposes, subject to ethical restrictions.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

NLP	Natural Language Processing
PII	Personally Identifiable Information
SR@K	Success Rate at K guesses
T5	Text-to-Text Transfer Transformer

Appendix A. Other Password-Guessing Methods Surveyed

Table A1. Other surveyed methods, each at its own reported setting; datasets, budgets, and tasks differ, so these are not directly comparable to the 4iQ results in Table 1.

Model	Approach	Key Innovation	Dataset	Guesses	Reported SR
Das et al. [10]	Rule-based	First cross-site guessing algorithm	11-site leak	10^2	30% [a]
PassLLM [14]	Decoder-only LLM	Billion-parameter fine-tuning (LoRA)	4iQ (reuse)	10^3	+8.27–9.66% [b]
RankGuess [15]	Markov + ranking	Adversarial ranking optimisation	12 pw + 6 PII	10^{12}	58.21–91.95% [c]
RFGuess [16]	Random Forest	Traditional ML for reuse	English sets	—	+7–22% [b]
PassGAN [17]	GAN	Generative password distribution	trawling	—	+51–73% [b]

Table A1. *Cont.*

Model	Approach	Key Innovation	Dataset	Guesses	Reported SR
GENPass [18]	PCFG + LSTM	Cross-site generalisation	cross-site	—	+16–30% [b]
Pasquini et al. [19]	Deep generative	Latent-space conditional generation	RockYou	5×10^{10}	51.80% [d]
GuessFuse [20]	Hybrid	Multi-view ensemble	6 sets (54M)	10^7	+4.70–59.62% [b]

[a] Cross-site guessing at 100 attempts (the origin work). [b] Relative improvement over a baseline; no absolute 4iQ@10^3 figure reported. [c] Targeted guessing with known PII at a 10^{12} budget. [d] Trawling on RockYou at 5×10^{10} guesses (vs. 34.19% for PassGAN).

Appendix B. Rule-Based Baseline Catalogue

The input-dependent rule-based baseline in Section 5.2 applies a fixed catalogue of about 180 hand-crafted transformations to each leaked password: case changes, digit and symbol suffixes (single digits, common multi-digit strings, two-digit years, and the range 00–99), short prefixes, single- and multi-character leet substitutions, and head/tail truncations. Table A2 lists the most productive rules, ordered by how often each one alone produces the exact target tweak in the training set; this training-frequency ordering is the variant reported at 36.72/40.72/41.26% in Section 5.2. For reference we also evaluate Hashcat's standard `best64` rule set (77 rules, version 6.2.6) [2] under identical scoring, which reaches 9.38/23.40/23.40%.

Table A2. Most productive rules in the hand-crafted catalogue, ordered by training-set firing count (the number of training pairs for which the rule alone maps the leaked password exactly onto its tweak).

Rank	Rule	Train Firings
1	Lowercase	5680
2	Delete first character	5189
3	Capitalise	3610
4	Delete last character	3381
5	Strip trailing digits	2790
6	Title-case	2761
7	Prepend "0"	2495
8	Delete first two characters	2150
9	Delete last two characters	1871
10	Swap case	1814
11	Delete last three characters	1592
12	Delete first three characters	1523
13	Append "1"	996
14	Uppercase	915
15	Capitalise + append "1"	389
16	Append "0"	200
17	Append "123"	197
18	Append "5"	144
19	Append "2"	140
20	Duplicate word	113

References

1. Nisenoff, A.; Golla, M.; Wei, M.; Hainline, J.; Szymanek, H.; Braun, A.; Hildebrandt, A.; Christensen, B.; Langenberg, D.; Ur, B. A Two-Decade Retrospective Analysis of a University's Vulnerability to Attacks Exploiting Reused Passwords. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 5127–5144.
2. Hashcat. Available online: https://hashcat.net/hashcat/ (accessed on 14 November 2026).
3. John the Ripper Password Cracker. Available online: https://www.openwall.com/john/ (accessed on 5 August 2026).
4. Xu, M.; Yu, J.; Zhang, X.; Wang, C.; Zhang, S.; Wu, H.; Han, W. Improving Real-world Password Guessing Attacks via Bi-directional Transformers. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 1001–1018.

5. Pal, B.; Daniel, T.; Chatterjee, R.; Ristenpart, T. Beyond Credential Stuffing: Password Similarity Models Using Neural Networks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2019; pp. 417–434. [CrossRef]
6. Wang, D.; Zou, Y.; Xiao, Y.A.; Ma, S.; Chen, X. Pass2Edit: A Multi-Step Generative Model for Guessing Edited Passwords. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 983–1000.
7. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67.
8. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; Volume 30, pp. 5998–6008.
9. Islam, S.; Elmekki, H.; Elsebai, A.; Bentahar, J.; Drawel, N.; Rjoub, G.; Pedrycz, W. A comprehensive survey on applications of transformers for deep learning tasks. *Expert Syst. Appl.* **2024**, *241*, 122666. [CrossRef]
10. Das, A.; Bonneau, J.; Caesar, M.; Borisov, N.; Wang, X. The Tangled Web of Password Reuse. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014. [CrossRef]
11. Xiu, K.; Wang, D. PointerGuess: Targeted Password Guessing Model Using Pointer Mechanism. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 5555–5572.
12. He, X.; Cheng, H.; Xie, J.; Wang, P.; Liang, K. PassTrans: An Improved Password Reuse Model Based on Transformer. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*; IEEE: New York, NY, USA, 2022; pp. 3044–3048. [CrossRef]
13. Wang, D.; Zhang, Z.; Wang, P.; Yan, J.; Huang, X. Targeted Online Password Guessing: An Underestimated Threat. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, 24–28 October 2016; pp. 1242–1254. [CrossRef]
14. Zou, Y.; An, M.; Wang, D. Password Guessing Using Large Language Models. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 7799–7818.
15. Yang, T.; Wang, D. RankGuess: Password Guessing Using Adversarial Ranking. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 12–15 May 2025*; IEEE Computer Society: Los Alamitos, CA, USA, 2025; pp. 645–663. [CrossRef]
16. Wang, D.; Zou, Y.; Zhang, Z.; Xiu, K. Password Guessing Using Random Forest. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 965–982.
17. Hitaj, B.; Gasti, P.; Ateniese, G.; Perez-Cruz, F. PassGAN: A Deep Learning Approach for Password Guessing. In *Proceedings of the Applied Cryptography and Network Security (ACNS 2019)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2019; Volume 11464, pp. 217–237. [CrossRef] [PubMed]
18. Xia, Z.; Yi, P.; Liu, Y.; Jiang, B.; Wang, W.; Zhu, T. GENPass: A Multi-Source Deep Learning Model for Password Guessing. *IEEE Trans. Multimed.* **2020**, *22*, 1323–1332. [CrossRef]
19. Pasquini, D.; Gangwal, A.; Ateniese, G.; Bernaschi, M.; Conti, M. Improving Password Guessing via Representation Learning. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2021; pp. 1382–1399. [CrossRef]
20. Xie, Z.; Shi, F.; Zhang, M.; Ma, H.; Wang, H.; Li, Z.; Zhang, Y. GuessFuse: Hybrid Password Guessing with Multi-View. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 4215–4230. [CrossRef]
21. Melicher, W.; Ur, B.; Segreti, S.M.; Komanduri, S.; Bauer, L.; Christin, N.; Cranor, L.F. Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 175–191.
22. Denning, D.E. A lattice model of secure information flow. *Commun. ACM* **1976**, *19*, 236–243. [CrossRef]
23. Goguen, J.A.; Meseguer, J. Security Policies and Security Models. In Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982; pp. 11–20. [CrossRef]
24. Focardi, R.; Gorrieri, R. A Classification of Security Properties for Process Algebras. *J. Comput. Secur.* **1995**, *3*, 5–33. [CrossRef]
25. Focardi, R.; Rossi, S. Information Flow Security in Dynamic Contexts. *J. Comput. Secur.* **2006**, *14*, 65–110. [CrossRef]
26. Smith, G. On the Foundations of Quantitative Information Flow. In *Proceedings of the Foundations of Software Science and Computational Structures (FoSSaCS 2009)*; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2009; Volume 5504, pp. 288–302. [CrossRef]
27. Kudo, T.; Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Brussels, Belgium, 31 October–4 November 2018; pp. 66–71.
28. Gastaldi, J.L.; Terilla, J.; Malagutti, L.; DuSell, B.; Vieira, T.; Cotterell, R. The Foundations of Tokenization: Statistical and Computational Concerns. In Proceedings of the International Conference on Learning Representations (ICLR), Singapore, 24–28 April 2025.

29. Fan, A.; Lewis, M.; Dauphin, Y. Hierarchical neural story generation. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Melbourne, Australia, 15–20 July 2018; pp. 889–898.
30. Wilson, E.B. Probable Inference, the Law of Succession, and Statistical Inference. *J. Am. Stat. Assoc.* **1927**, *22*, 209–212. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.