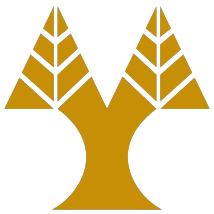
Thesis Dissertation

HONEYTRAPS: ENHANCING HONEYWORDS

Panayiotis Charitonos

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Honeytraps: Enhancing Honeywords

Panayiotis Charitonos

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgments

First and foremost, I would like to express my gratitude to Dr. Elias Athanasopoulos for his invaluable guidance throughout the course of my thesis. His insights and constructive feedback have been significant in forming the direction of my research. I am truly grateful for the opportunity to learn under his mentorship.

I would also like to sincerely thank Dr. Antreas Dionysiou for his expert advice and thoughtful input at crucial stages of this work. His depth of knowledge and willingness to share his time and experience played a significant role in the successful completion of this thesis.

Finally, I would also like to extend my heartfelt thanks to my friends and family for their support and encouragement throughout this journey.

Summary

The growing frequency of credential database breaches presents a major threat to web applications, enabling attackers to gain unauthorized access, perform identity theft. Honeywords are decoy passwords stored alongside real ones. These decoy passwords are designed to detect breaches by triggering an alarm when an attacker mistakenly submits a decoy, believing it to be the real password. However, existing honeyword generation techniques struggle to withstand modern threats, particularly in multi-server breach scenarios where users reuse passwords. Attackers can exploit intersections across honeyword sets to identify the true password. Moreover, attempts to fool the detection system by deliberately triggering false breach alarms is also an attack vector honeywords will face. In this thesis, we introduce HoneyTraps, a plug-and-play replacement for existing password storage schemes that probabilistically defends against intersection attacks, and false-positive manipulation. HoneyTraps replaces the traditional password hash with a structured list of credential tokens (CTs), consisting of one real token and multiple probabilistically selected decoys and traps. By modeling adversaries such as baseline-offline, false-positive (FP+), and hidden-offline attackers as constrained n-shot games, we formally capture their capabilities and demonstrate the optimality of their strategies. HoneyTraps leverages system-wide trap-to-honeytoken ratios to distinguish genuine breaches from false alarms, enabling confident detections. We implement and evaluate the HoneyTraps system using testbed simulations. Results show that HoneyTraps achieves high detection rates while producing zero false alarms across all attack scenarios. We also demonstrate its interoperability with legacy hash formats and significant storage optimizations via run-length encoding. Additionally, we discuss and evaluate the limitations of existing honeyword schemas. Our findings establish HoneyTraps as a practical, deployable framework that enhances the robustness of honeyword authentication systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Passwords have long been the cornerstone of user authentication, yet decades of research have revealed that they are fundamentally flawed and increasingly inadequate in the face of modern cybersecurity threats. Numerous studies have demonstrated that passwords are susceptible to a wide range of attacks—from brute force and phishing to more subtle issues like password reuse. Given these well-documented shortcomings, many experts argue that passwords should have been phased out in favor of more secure and user-friendly alternatives. For example, Bonneau et al. [4] provide a comprehensive evaluation of web authentication schemes and underscore the inherent vulnerabilities of password-based systems. Similarly, research by Florêncio and Herley [12] highlights the problematic nature of user-generated passwords and the risks associated with poor password habits. Yet, despite the availability of more robust authentication mechanisms, the widespread inertia of existing legacy systems and the challenges involved in deploying new technologies have perpetuated the reliance on passwords across digital platforms.

This persistent reliance on passwords directly contributes to a broader and escalating security risk. Credential database breaches continue to be a pervasive and escalating security concern. According to the 2022 Verizon Data Breach Investigations Report, credentials rank among the two most frequently compromised types of confidential data because of the significant advantage they offer attackers in disquising as legitimate users [31]. These breaches serve as the primary source of compromised passwords used in credential stuffing campaigns, which are responsible for most account takeovers [26, 30]. Adding to the challenge is the substantial delay—often estimated to be between six to eight months—between the occurrence of a data breach and its detection [25, Fig. 45]. This extended period of vulnerability provides attackers with ample time to crack the stolen passwords offline, subsequently selling them or using them directly to carry out further unauthorized activities.

A strategy to speedup the detection of credential database breaches, suggested by Juels and Rivest [20], involves storing decoy passwords—known as honeywords—alongside

the actual passwords in a site's credential database. Honeywords are carefully crafted to appear indistinguishable from a user's true password, creating ambiguity for potential attackers by making it difficult to discern which password is genuine from the list commonly referred to as the sweetlist. In a secure system, both the actual password and multiple honeywords are maintained for each user so that any attempt to log in using one of the decoys immediately triggers a security alert. This approach not only misleads intruders but also enables rapid detection of unauthorized access attempts, thereby strengthening the overall security posture of the authentication system.

A key aspect of ensuring the efficacy of this strategy lies in the design of the honeyword generation process itself. A critical requirement for robust honeyword generation techniques is that they must be irreversible [17]. In other words, even if an adversary gains full access to the honeyword generation process (HGT), they should not be able to reverse-engineer the procedure to retrieve the original password. Specifically, given a user's sweetlist, an attacker must not be able to undo the generation process—much like a cryptographic hash function, where it is computationally infeasible to determine the original input from its hash value. Additionally, the process must ensure that each time a password is processed—even if it is the same password—the resulting set of honeywords is completely different from previous instances. This property is essential because, if the algorithm were deterministic, an attacker could prompt the HGT with various passwords and identify the real one as the only input that consistently yields the same list of fake passwords.

An HGT that satisfies the non-reversibility property guarantees that the honeywords generated are entirely different with every invocation, even if the same password is submitted multiple times. While this randomness is crucial for security, it introduces a vulnerability: when a user reuses the same password across different servers, the variation in honeywords may cause the intersection of these sets to converge on a single common element—the user's actual password. This phenomenon, known as an intersection attack [20], highlights a significant risk, especially given the well-documented trend of password reuse across multiple accounts [8, 24, 32]. To mitigate such attacks, there is a clear need for a standalone framework that does not rely on server-to-server cooperation (which is often impractical) but instead guarantees that the intersection of credentials from different servers—when a user employs the same password—yields multiple elements. In this scenario, the larger the size of the intersection set, the better the security, as it makes it considerably harder for an attacker to pinpoint the true password.

A plausible solution is to generate honeywords deterministically so that, when a user reuses the same password on different systems, the resulting sweetlist remains identical. This approach increases the overlap across servers without requiring any inter-server communication. However, there are two major drawbacks. First, because the algorithm is

publicly available, it becomes possible to reverse the process by systematically querying the generation function with each candidate sweetword. The candidate that reproduces the full, expected sweetlist is likely to be the actual password. Second, this deterministic approach is vulnerable to online guessing attacks by adversaries who haven't breached the database but instead aim to trigger false breach alarms. An attacker could register with a specific password and, by replicating the publicly available HGT, deliberately use a honeyword in a login attempt to set off an alert. Since the HGT's security does not depend on keeping the algorithm secret, such an attack is feasible.

Huang et al. [17] reveal that contemporary honeyword generation schemes often struggle to balance two opposing errors: false negatives, where real intrusions slip by undetected, and false positives, where innocuous login attempts with decoy credentials trigger unwarranted breach alarms—especially when an attacker already knows a user's password from another service. Building on this insight, our work uncovers an additional design conflict: reducing the rate of false alarms while also defending against attacks that leverage password reuse pulls the system in two opposite directions. Specifically:

- **Suppressing false positives** demands a high degree of randomness in honeyword creation so that malicious users cannot deliberately trigger a false alarm by inserting a honeyword.
- **Resisting intersection attacks** requires deterministic outputs, ensuring that repeated use of the same password produces identical sweetlists across different servers.
- **The conflicting demand** arises because increased randomness undermines cross-server consistency—weakening defenses against password-reuse exploits—while greater determinism heightens the likelihood of false alarms.

Consequently, any attempt to optimize for one of these goals inevitably erodes the other, forcing the need for a careful balance between stochasticity and repeatability in HGTs.

Wang and Reiter [35] introduced Bernoulli Honeywords, the first technique to offer a precisely controllable false alarm rate. Their method treats honeyword selection as a Bernoulli process over the entire password space and records each candidate—with a fixed sampling probability—in a 128-bit Bloom filter (BF) using 20 independent hash functions, while always inserting the genuine password (sampling probability = 1). As Wang and Reiter note, explicitly generating and storing all sampled honeywords would be infeasible. Instead, they implicitly encode them by randomly setting extra bits in the BF: after the true password's bits are marked, they continue flipping random positions until a total of 30 bits are set. During authentication, any password whose hash does not activate exactly the same subset of 30 bits as the real password immediately triggers a breach alarm.

While Bernoulli Honeywords initially appears as an appealing solution, it ultimately falls short in several aspects. First, with the utilised BF structure ($b = 128$, $k = 20$) an adversary has a low probability of finding elements (passwords) that return a member-positive result, other than the user-chosen password and a single honeyword. An accurate estimate suggests that ≈ 2 stored elements correspond to 30 marked bits in the specific BF structure [22]. For any other string to appear as a honeyword, it must falsely yield a member-positive result (false positive), which occurs with a negligible probability of 4.019×10^{-12}. Additionally, Bernoulli Honeywords do not store arbitrary inputs but rather user-chosen passwords, which are often short in length and limited to a specific set of printable characters. This significantly lowers the chance of random inputs representing viable solutions. These facts deem dictionary attacks as an effective method for recovering the user-chosen password. Furthermore, the probability of finding a honeyword in an intersection attack decreases exponentially as the number of breached servers increases. This is because, for a honeyword to be a potential solution in an intersection attack, it must yield a member-positive result in *all* of the user's BFs, which are derived from different Bernoulli Honeyword servers.

In this paper, we propose HONEYTRAPS, an enhanced honeyword-based schema that satisfies both objectives: achieving an acceptably low false alarm rate while offering probabilistic guarantees against intersection and credential stuffing attacks. HONEYTRAPS uses a single cryptographic hash function (SHA3-256) with modulo reduction to introduce tunable and controllable collisions. This in turn allows us to configure the likelihood of common decoy credentials in multi-server breach scenarios. Specifically, to compute the valid Credential Token (CT) for a user-chosen password, the following deterministic operation is performed: SHA3-256(password) mod w, where w is a tunable integer parameter that determines the range of CTs. Since cryptographic hash functions are designed to uniformly cover their output range [6], constraining the output to a range of size w ensures uniform collisions per CT. The parameter w is inversely proportional to the number of collisions—i.e., as w increases, the number of (randomly generated or human-chosen) passwords mapping to the same CT decreases. Conceptually, HONEYTRAPS can be viewed as a flattened BF, which—compared to the BF used in Bernoulli Honeywords—has a much higher capacity for storing honeywords.

Similar to Bernoulli Honeywords, HONEYTRAPS produces honeytokens independently of the user's password, allowing it to be integrated into existing authentication systems that store password hashes with any standard cryptographic function—without requiring users to reset or re-enter their credentials. By contrast, all prior password-dependent HGTs [10, 16, 20, 23, 36] mandate a full re-entry of user passwords to generate honeywords, a clearly impractical requirement for real-world deployments. To demonstrate HONEYTRAPS's deployability, we have released a pip-installable Django package

including a custom authentication model with built-in HONEYTRAPS support, which can be seamlessly added to both new and existing Django projects without any user password changes.

Additionally, we introduce *traps*, which we define more concretely in later sections a mechanism to mitigate online adversaries—those who have not breached the credential database. These adversaries fall into two main categories. First, some attackers seek to provoke false breach alarms by deliberately submitting honeywords during login attempts. For example, an adversary who registers on the target site may craft passwords based on their understanding of the honeyword generation process, hoping to replicate one of the decoys. Second, credential-stuffing adversaries exploit credentials leaked from another service (site B) to infiltrate user accounts on site A [29]. Although, HONEYTRAPS can defend against both type of adversaries in this thesis we focus on the former type of adversary.

A key design challenge is determining the optimal number and placement range of Credential Tokens (CTs) for each user account. This involves balancing security and practicality: increasing the number of honeytokens strengthens defenses against offline adversaries but also heightens the likelihood of false breach alarms. Since breach responses are time-sensitive, costly, and disruptive [18], frequent false alarms may lead operators to ignore them [35]. To address this, we perform a comprehensive analysis of HoneyTraps' hyperparameters to find configurations that minimize false positives, ensure timely breach detection, and resist intersection attacks effectively.

Our *contributions* can be summarized as follows.

1. We design, implement, and evaluate HONEYTRAPS, the first standalone framework aimed at defending honeyword-based systems against attacks that exploit users' tendency to reuse passwords. Through a comprehensive security analysis, we demonstrate that HONEYTRAPS provides probabilistic guarantees against online adversaries (e.g. false-positive attacks) and offline adversaries (e.g., honeytoken distinguishing attacks), aligning with thresholds suggested in the literature. Thus, HONEYTRAPS enhances honeyword-based systems by streamlining their design and enabling practical, real-world deployment.
2. We are the first to identify a new trade-off in the design of standalone honeyword-based systems, adding to those previously observed in the literature [17]. Specifically, we show that improving security against online adversaries who know the user's real password and aim to trigger false alarms comes at the cost of reduced detection guarantees against intersection and credential stuffing attacks. This trade-off highlights the complexity of designing effective standalone honeyword-based solutions.
3. To support reproducibility, practical adoption, and further research on this topic, we

release the following: (i) a pip-installable Django package featuring a custom user authentication model with built-in HONEYTRAPS support, designed to integrate seamlessly into both new and existing Django projects—without requiring users to re-enter their passwords; and (ii) the complete source code for our experiments, along with a fully functional prototype of the HONEYTRAPS framework. (iii) The testbed used to model attackers to test the framework before deploying in a real-world setting.

Chapter 2

Background

2.1 Bloom Filters

A Bloom filter is a space-efficient, probabilistic data structure for representing a set $X = \{x_1, x_2, \ldots, x_n\}$ and supporting membership queries with no false negatives and a tunable false positive rate [3]. Formally, a Bloom filter is defined by:

- A bit array B of length m, initially $B[i] = 0$ for $i = 0, 1, \ldots, m-1$.
- A family of k independent hash functions $\{h_1, h_2, \ldots, h_k\}$, each mapping any element x to a uniform position in $\{0, \ldots, m-1\}$.

Insertion. To insert an element $x \in X$, compute each hash $h_i(x)$ and set

$$B\big[h_i(x)\big] \;=\; 1, \quad i = 1, \ldots, k.$$

Query. To test whether an element y is in X:

if $B\big[h_i(y)\big] = 1$ for all $i = 1, \ldots, k,$ then report "possibly in set"; else "definitely not in set."

Since any inserted element x will have all its corresponding bits set, membership queries never yield false negatives. However, non-member elements $y \notin X$ may have all k bits set by previous insertions, leading to *false positives*.

False positive probability. After inserting n elements, the probability that a particular bit remains zero is

$$\Pr[B[i] = 0] = \left(1 - \tfrac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}.$$

Hence the probability that a bit is one is

$$\Pr[B[i] = 1] = 1 - e^{-\frac{kn}{m}}.$$

A membership query for $y \notin X$ declares "possibly in set" only if all k hash positions are ones, giving a false positive rate

$$p_{\text{FP}} = \left(1 - e^{-\frac{kn}{m}}\right)^k.$$

This rate can be tuned by choosing m and k appropriately.

2.2 Honeywords

Password-based authentication systems maintain a sensitive password file, F, containing each user's password digest. A login attempt is accepted only if the hash of the provided password matches the stored digest in F for the corresponding user; otherwise, it is rejected. If F is compromised, it is realistic to assume that most (if not all) of the hashes can be reversed, revealing users' plaintext passwords. This is due to advanced password-guessing techniques and the availability of high-performance computing resources, such as GPUs and distributed clusters [10].

Honeywords are decoy passwords stored alongside each user's true password in a credential database. Introduced by Juels and Rivest [20], this mechanism transforms a stolen password file into a high-risk guessing game for attackers: any attempt to log in with one of the honeywords immediately triggers an alarm. The combined list of the real password and its decoys is often called the *sweetlist*. Honeywords rely on two core security properties. First, they must be *indistinguishable* from the real password without access to a secure "honeychecker." Second, they must be *irreversible*, meaning that even with full knowledge of the honeyword generation algorithm, it remains computationally infeasible to recover the original password [17]. These properties ensure that attackers cannot either identify the real password within the sweetlist or evade detection once they attempt to use a decoy.

This adversarial scenario can be formalized as an one-shot guessing game. Let the sweetlist for a user be

$$S = \{\, p,\, h_1,\, h_2,\, \ldots,\, h_k \,\},$$

where p is the real password and $h_1, \ldots, h_k$ are the k honeywords, so $|S| = k+1$. An adversary selecting an element $g \in S$ faces two outcomes:

$$\begin{cases} \text{success without alarm}, & g = p, \\ \text{alarm and failure}, & g \neq p. \end{cases}$$

If honeywords are truly indistinguishable from the real password, the adversary must guess uniformly at random, yielding

$$\Pr[\text{win}] = \Pr[g = p] = \frac{1}{k+1}, \quad \Pr[\text{lose}] = \Pr[g \neq p] = \frac{k}{k+1}.$$

The adversary's campaign may span multiple accounts, but the game terminates as soon as either all targeted accounts have been breached without triggering an alarm or the adversary is detected by setting off a breach alarm on any single attempt.

2.3 Honeychecker

The *honeychecker* is a minimal, always-trusted component that holds the secret index of each user's real password within their sweetlist. It never learns the passwords themselves—only the position i^* (with $1 \leq i^* \leq k+1$) corresponding to the genuine credential in

$$S = \{p, h_1, h_2, \ldots, h_k\}.$$

When a user submits a login attempt with guess such that $g \in S$, the authentication server computes its position i in S and sends i (not g) to the honeychecker. The honeychecker performs a the following check:

$$\begin{cases} \text{accept}, & \text{if } i = i^*, \\ \text{alarm}, & \text{if } i \neq i^*. \end{cases}$$

In the event that the submitted credential does not match any element of the sweetlist (i.e. $g \notin S$), the honeychecker is not invoked and the system rejects the login silently; no alarm is raised and access is not granted.

2.4 Honeyword Generation Requirements

A robust honeyword generation technique (HGT) must satisfy several key security properties to ensure that the addition of decoy credentials does not weaken the underlying authentication system. First and foremost, honeywords must be *indistinguishable* from the real password. In practice, this means that an attacker who obtains a user's sweetlist cannot identify the genuine credential by inspecting patterns in the decoys alone; without access to an external "honeychecker," each entry in the list should appear equally plausible [20].

Equally important is *non-reversibility*. Even if an adversary knows the exact algorithm used to produce honeywords, it must remain computationally infeasible to recover the original password from the sweetlist, just as cryptographic hash functions resist inversion [17]. This property prevents attackers from feeding candidate passwords into the HGT and matching outputs against the stored sweetlist to isolate the true credential.

Finally, something often overlook when it comes to HGT's is the practical considerations impose additional requirements. A generation method should allow *tunable false-alarm rates*, so that legitimate login attempts rarely trigger decoys, while still ensuring

prompt detection of real breaches [35]. At the same time, the algorithm must be efficient and easy to integrate into existing authentication infrastructures—ideally producing honeywords without requiring users to reset or re-enter their passwords.

2.5 Machine-Learning Support for Honeyword Generation

Motivation. Early honeyword schemes relied on ad-hoc "tweak" rules (e.g., flipping case or appending digits) that often produced unnatural decoys. A modern alternative is to *learn* the statistical structure of real passwords from data and then sample honeywords that match this distribution. The stronger the resemblance, the harder it is for an offline attacker to filter out the genuine secret.

FastText in a nutshell. *FastText* [19] is a lightweight neural language-model library released by Facebook AI Research that learns *character n*-gram embeddings rather than whole-word vectors. Given a password such as

"P@ssw0rd!",

FastText extracts all overlapping n-grams of length $n \in \{3,4,5,6\}$ (e.g. `<P@`, `P@s`, `ssw`, `sw0`, ...). Each n-gram receives a trainable vector $\mathbf{v}_g$; the final embedding of the entire password is the average

$$\mathbf{v}_{\text{P@ssw0rd!}} = \frac{1}{|G|} \sum_{g \in G} \mathbf{v}_g,$$

where G is the multiset of extracted n-grams. Because any previously unseen string can be decomposed into known n-grams, FastText embeds *out-of-vocabulary* passwords with no loss of fidelity—a crucial property for long-tail, policy-constrained secrets.

Generating honeywords with FastText. To fabricate decoys for a user's real password RP:

1. embed $\mathit{real password}(RP)$ to obtain its vector $\mathbf{v}_{RP}$;
2. query the k nearest neighbours of $\mathbf{v}_{RP}$ in the learned space;

Because the neighbourhood query returns strings that are statistically indistinguishable from real passwords, the resulting honeywords blend seamlessly into the site-specific distribution while requiring only a single embedding lookup at enrolment time.

Chapter 3

Architecture

3.1 Limitations of existing HGTs

3.1.1 HoneyGen

At first glance, training a machine-learning model to synthesise honeywords seems appealing, yet the approach has two underlying limitations:

(1) Model reconstruction. A generator M must be trained on a password corpus D. An adversary who aggregates large public leaks can approximate D and—by re-training under similar hyper-parameters—obtain a surrogate model $\widetilde{M}$. Because $\widetilde{M}$ reproduces a noticeable fraction of the honeywords that M would emit, the attacker's chance of guessing a decoy (and so raising a false breach alarm) rises even though the target site itself was never breached.

(2) Inconsistent sweetlists across sites. Training sets are inherently site-specific. Two services that deploy the *same* ML generator will still train on *different* corpora D_1 and D_2, producing distinct honeyword distributions for the same reused password. When an attacker intersects the two sweetlists, the *only* token common to both lists is overwhelmingly likely to be the genuine password, introducing intersection attacks rendering honeywords useless in multi-server breach scenarios.

HoneyGen introduced by Dionysiou *et al.* [10], instead of applying hand-coded "tweak" rules, HoneyGen learns a site-specific *FastText* model on the operator's own password database.

Three generation modes.

1. *Chaffing-by-tweaking.* Applies small random edits to the real password (case flips, symbol swaps, digit shifts). Fast and policy-aware, but its realism depends on the quality of the tweak rules.

2. *Chaffing-with-a-password-model.* Retrieves the k nearest neighbours of the real password in the FastText space; these neighbours already follow the empirical distribution of the site's users.

3. *Hybrid.* Combines the two: half of the honeywords come directly from the FastText neighbourhood; the other half are tweaked versions of those neighbours. This yields the best trade-off between realism (from the model) and diversity (from tweaking).

HoneyGen is therefore a useful proxy for evaluating the broader class of ML-based honeyword generators. To illustrate their inherent limitations we conducted an experiment with the `rockyou` dataset. First, we trained a FastText model M_1 on a subset D_1 and a second model M_2 on a *disjoint* subset D_2 ($D_1 \cap D_2 = \emptyset$). When we queried both models in *Hybrid* mode with the *same* input password, the resulting sweetlists exhibited zero overlap: a perfect demonstration of how corpus divergence defeats inter-site intersection. Conversely, querying M_1 (or M_2) twice produced *identical* sweetlists whenever the optional tweak stage was disabled, showing that an attacker who can reproduce the training corpus can pre-compute a subset of honeywords and inflate the false-positive rate. This demonstarates the underlying problem of creating intersection in sweetlists: while using the same or similar password corpus D to train different models allows for more intersections it also raises the false-postive occurances.

3.1.2 Bernoulli Honeywords

Wang and Reiter [35] proposed the first honeyword-based framework with a quantifiable and tunable false alarm rate, called Bernoulli Honeywords. Honeywords are implicitly selected using a Bernoulli process: each password (excluding the user-chosen one) is independently selected with a fixed probability p_h to be a honeyword. To efficiently select and store these without explicit enumeration, a BF of size $b = 128$ bits and $k = 20$ SHA-1 hash functions is configured per user. In particular, each user's real password is first hashed using SHA3-256 to produce a `digest`, and bits corresponding to `SHA-1(digest` $+ i$`) mod` b for $0 \leq i \leq 19$ are marked (with 1). Finally, an additional y bits are randomly marked to ensure that the total number of set bits is 30, as specified by the `numOnes` parameter in their official implementation. These extra bits are not marked by explicitly inserting honeywords but are randomly selected from unmarked bits. Essentially, this means that honeyword creation is out of control and that the stored honeywords are not generated to be visually similar to the real password or to any password at all. In fact, as we show below, the BF acts as a deterministic oracle—akin to a cryptographic hash function in the absence of honeywords—and can be efficiently reversed without diluting the attacker with multiple possible password options.

To make this clear, consider a very simple BF configuration (e.g., 2 bits, 1 hash function, and 1 inserted element—the real password). In this case, it is trivial to explicitly insert honeywords that map to the single marked bit. Reversing this BF is practically impossible, since any password has a 50% chance of being a potential solution (i.e., mapping to the marked bit) during a cracking attempt. In fact, this BF effectively contains half of all possible honeywords that could be computed with *any* algorithm. However, this is not the case with the BF used in Bernoulli Honeywords. Specifically, the configuration parameters suggested by Wang and Reiter [35] make the BF easy to reverse. As the BF grows (i.e., as the bitmap size and number of hash functions increase), finding inputs that map to specific, already marked bits becomes increasingly difficult—equivalent to the probability of a false positive. A false positive occurs only when the input maps to an exact subset of the bits set by the real password. This probability, which can be computed using Eq. (3.1) as suggested by Cao and Wang [7], is inversely proportional to the BF's size—that is, its bitmap length b, number of hash functions k, and number of inserted elements n. For the 2-bit BF discussed earlier, the false positive probability p_{fp} is 0.5.

$$p_{fp} = \left(1 - \left(1 - \frac{1}{b}\right)^{kn}\right)^{k} \tag{3.1}$$

Mitzenmacher and Upfal [22] provide a formula for estimating the expected number of marked bits, X, in a BF, given b, k, and n (Eq. 3.2). By solving for n, this formula can be adapted to estimate the number of explicitly inserted elements based on the total number of marked bits X. In Bernoulli Honeywords, only the real password is explicitly inserted into the BF, and an additional y bits are randomly marked. Thus, we can apply Eq. 3.2 to estimate the number of elements that could have been explicitly inserted. Using the parameters suggested by Wang and Reiter [35] (i.e., $b = 128$, $k = 20$, and $X = 30$), the formula yields an estimate of approximately 2 elements (1.7 to be precise). However, this estimate is not entirely accurate, as there is a possibility of more elements being 'stored' in the BF, particularly those that map to an exact subset of the 30 marked bits. These elements are now difficult to find. In fact, the probability of finding such an input corresponds to the false positive rate of the BF. For the configuration used by Wang and Reiter, this probability is approximately 4.019×10^{-12}, which is significantly lower than the 50% false positive rate seen in the earlier 2-bit BF example. As the BF grows, the probability of a false positive (i.e., encountering another honeyword) decreases, eventually approaching the determinism of a cryptographic digest. In other words, the BF evolves from a random oracle (e.g., 50% membership probability in a 2-bit BF) to a highly deterministic oracle that minimizes collisions—similar to a cryptographic hash function, where a

membership-positive result strongly indicates actual presence in the filter.

$$X = b\left[1 - \left(1 - \frac{1}{b}\right)^{kn}\right] \Rightarrow n = \frac{\ln\left[1 - \frac{X}{b}\right]}{kn \times \ln\left[1 - \frac{1}{b}\right]} \tag{3.2}$$

As a consequence, the BF structure used in Bernoulli Honeywords also fails to provide meaningful security guarantees against intersection attacks. The probability $P(I)$ of an input—other than the user-chosen password—yielding a member-positive result across all of a user's BFs on different servers is given by:

$$P(I) = \prod_{i=2}^{y} p_{fp} = \prod_{i=2}^{y} \left(1 - \left(1 - \frac{1}{b}\right)^{kn}\right)^{k}, \tag{3.3}$$

where y is the number of breached servers on which the user has accounts with the same password. As shown, this probability is significantly low and can be considered negligible. For example, performing an intersection attack on two of a user's BFs yields a single additional solution—other than the user-chosen password—with a probability of approximately $(4.019 \times 10^{-12})^2$.

To further demonstrate the inability of Bernoulli Honeywords to provide meaningful guarantees against intersection attacks, we conduct the following experiment. First, we randomly select 1,000 human-chosen passwords from the RockYou dataset [5]. For each password, we simulate a scenario in which a user registers on two different Bernoulli Honeywords-enabled servers using the same password. The construction of each user's BF follows the implementation by Wang and Reiter [35]. Next, we assume an intersection adversary who has compromised the credential databases (i.e., the BFs) from both servers. The adversary compiles a list of one billion human-chosen passwords leaked from various sites (e.g., RockYou, LinkedIn, Dropbox, Yahoo) and checks each password for membership in both BFs. A password is considered a potential solution for a user if both BFs respond affirmatively. The more such solutions per user, the stronger the protection against intersection attacks.

The experiment yielded *no* solutions beyond the user-chosen password pwd_t for any of the 1,000 target users—i.e., no false positives. Thus, with Bernoulli Honeywords, identifying a member-positive input other than pwd_t is challenging, and finding one that intersects across both BFs is exponentially harder. This implies two key observations: (1) credential stuffing adversaries face negligible risk of triggering honey-bits and getting detected; and (2) a password that tests positive across all BFs is likely the real password. Adapting Bernoulli Honeywords involves several inherent trade-offs [35], making it particularly non-trivial to modify the scheme for mitigating such attacks. Hence, we propose an alternative design that maps passwords to a configurable set of tokens (digits), offering both a low false alarm rate and effective resistance against intersection and credential stuffing attacks.

3.2 Threat Model

In this section, we outline the adversarial environments that HONEYTRAPS must withstand. We begin by characterizing *online adversaries*, who interact only through the normal authentication interface, and then describe *offline adversaries*, who have exfiltrated stored CTs but cannot tamper with the live system. For each class, we detail their objectives, capabilities, and the operational constraints under which HONEYTRAPS remains secure. We consider two distinct classes of adversaries, *online* and *offline* adversaries.

3.2.1 Online Adversaries

Online adversaries have not gained direct access to the site's credential storage. Their objectives are twofold: (1) to compromise user accounts by submitting valid CTs (credential-stuffing attacks), and (2) to provoke false breach alarms by intentionally guessing honeytokens (false-positive attacks). This characterization matches prior analyses of online threats in honeyword systems [17, 35].

3.2.2 Offline Adversaries

Offline adversaries operate under a passive breach scenario, as formalized in AMNESIA [34] and Lethe [9]. We assume they can exfiltrate all persisted CT values and invert the generation process to recover every associated password. However, they cannot:

- Alter server-side logic or inject malicious code.
- Break cryptographic protections for inter-server channels (e.g., obtain TLS private keys).
- Launch man-in-the-middle attacks against live communications.
- Predict future randomness used by the system.

Under these constraints, offline adversaries cannot interfere with live authentication operations but can analyze leaked CTs in bulk.

3.3 Security Requirements

We require HONEYTRAPS to meet two independent security goals: (1) withstand online guessing campaigns without excessive compromises or false alarms, and (2) prevent offline adversaries from distinguishing the real CT from honeytokens with probability above a specified bound.

3.3.1 Online Guessing Resistance

We model two classes of online attackers based on Florêncio et al. [13]:

- **Depth-first attackers** repeatedly target a small set of accounts, making up to $\lambda_d = 10^6$ guesses per account ($n_d = 10$ accounts).
- **Breadth-first attackers** spread guesses across many accounts, making up to $\lambda_b = 10^4$ guesses on each of $n_b = 10^3$ accounts.

Let $P_{\text{online}}(\lambda, n)$ denote the probability that an attacker, making up to λ guesses against each of n accounts, either (a) correctly hits a genuine CT or (b) triggers a false alarm by guessing a honeytoken. Following Wang and Reiter [35], we require:

$$P_{\text{online}}(\lambda_d, n_d) \leq 10^{-1}, \quad P_{\text{online}}(\lambda_b, n_b) \leq 10^{-1}. \tag{3.4}$$

That is, even if an adversary invests 10^6 guesses on 10 accounts or 10^4 guesses on each of 1000 accounts, the chance of any successful compromise or false alarm remains below 10%.

3.3.2 Offline Distinguishability

An offline adversary who obtains a user's sweetlist

$$H_i = \{h_{i,1}, \ldots, h_{i,k+1}\}$$

must not be able to identify the real CT c_i with probability exceeding the classic honeywords bound. Let

$$P_{\text{offline}}(H_i) = \Pr[\text{adversary picks } c_i \text{ from } H_i] \tag{3.5}$$

denote the success probability of an uninformed offline guess. We enforce:

$$P_{\text{offline}}(H_i) \leq \tfrac{1}{20},$$

i.e., the adversary's chance of selecting the true CT at random from the $(k+1)$-element sweetlist does not exceed 5%, matching Juels and Rivest's original security margin [20]. Furthermore, we assume each honeytoken

$$h_{i,j} \in H_i \setminus \{c_i\}$$

satisfies the indistinguishability property defined in Section 2.4 of Chapter 2.

3.4 HoneyTraps Overiew & Core Concepts

HONEYTRAPS assigns each user a Credential Token (CT) by hashing their password with SHA3-256 and reducing modulo w:

$$c \;=\; \text{SHA3-256}(\text{password}) \bmod w, \quad c \in \{0, 1, \ldots, w-1\}.$$

Here, w is a tunable parameter defining the CT space. Once the real CT c is computed, HONEYTRAPS adds a fixed number of decoys—called honeytokens—around it. We choose two nonnegative integers, r_{win} (right-side) and ℓ_{win} (left-side), so that

$$\text{total credential tokens} \;=\; \ell_{win} + 1 + r_{win}.$$

The amount of honeytokens placed above and below (ℓ_{win} and r_{win}) c, prevent an attacker from distinguishing c by position alone.

To select the honeytokens, HONEYTRAPS performs independent Bernoulli trials with probability p over CT values greater than c, picking right-side tokens until r_{win} are chosen. It repeats the process for values less than c to pick ℓ_{win} left-side tokens. All chosen honeytokens are stored explicitly.

Positions between the outermost honeytokens that were never selected become *traps*. A CT in this interval that is neither the true token nor a stored honeytoken. Since traps are not recorded in the database, any login using one signals an online attack—whether an adversary who knows the real password tries to force a false alarm, or a credential-stuffing attacker submits leaked CTs. See Figure 3.1 for a shematic example of HONEYTRAPS.

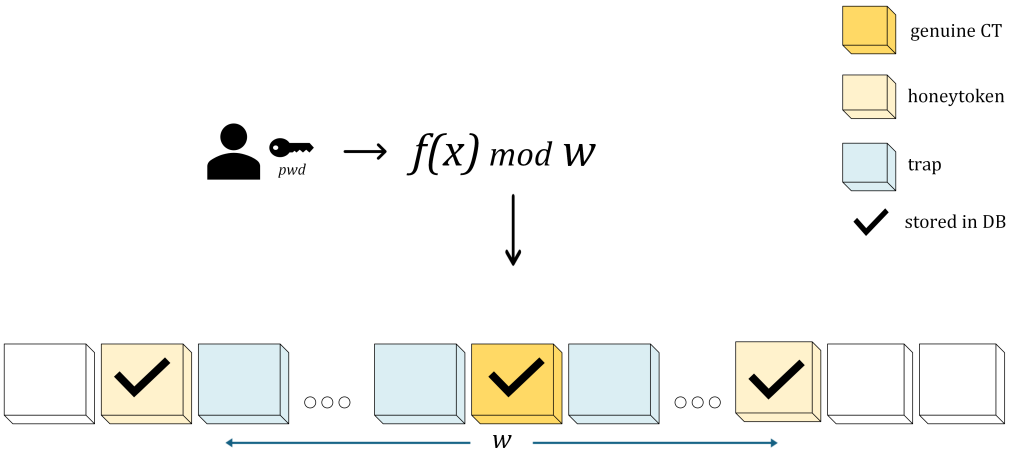


Figure 3.1: Illustration of HoneyTraps logic. The function $f(x)$ represents a cryptographically secure hash (e.g., SHA3-256), and w denotes the size of the token space.

3.5 HoneyTraps Formal Definition

In HONEYTRAPS, each user's true Credential Token (CT) is computed as

$$c \;=\; \text{SHA3-256(password)} \bmod w, \quad c \in \mathbb{Z}_w = \{0,1,\ldots,w-1\}.$$

For each user i, let

$$H_i \;\subseteq\; \mathbb{Z}_w = \{0,1,\ldots,w-1\}, \quad |H_i| = k+1,$$

denote the *sweetlist*, consisting of the genuine token c_i and k honeytokens. Equivalently, we may write

$$H_i = \{c_i\} \cup \{h_{i,1}, h_{i,2}, \ldots, h_{i,k}\}, \quad c_i \notin \{h_{i,1},\ldots,h_{i,k}\}.$$

For each user i, we first draw two nonnegative integers $\ell_{win,i}$ and $r_{win,i}$, representing the number of honeytokens to place to the left (values smaller than c_i) and to the right (values larger than c_i) of the true token c_i. Hence, the total size of the sweetlist H_i satisfies

$$|H_i| \;=\; \ell_{win,i} + 1 + r_{win,i}.$$

Define the candidate sets

$$\Delta_{r,i} = \{c_i+1,\, c_i+2,\, \ldots,\, w-1\}, \qquad \Delta_{\ell,i} = \{0,\, 1,\, \ldots,\, c_i-1\}$$

We perform independent Bernoulli trials $X_{x,i} \sim \text{Bernoulli}(p)$ on each element of $x \in \Delta_{r,i}$ in ascending order of x. Whenever a trial succeeds ($X_{x,i} = 1$), we include x in the right-side honeytoken set $H_{r,i}$. We scan $\Delta_{r,i}$ cyclically—wrapping back to c_i+1 after w—until $|H_{r,i}| = r_{win,i}$. Analogously, we scan $\Delta_{\ell,i}$ in descending order $(c_i-1, c_i-2, \ldots, 0)$ to collect ℓ_{win} left-side honeytokens $H_{\ell,i}$, wrapping back to c_i-1 if needed. Including the true token c_i for each user i yields the full *sweetlist*

$$H_i \;=\; \{c_i\} \cup H_{r,i} \cup H_{\ell,i}, \quad |H_i| \;=\; \ell_{win,i} + 1 + r_{win,i}$$

More specifically the *sweetlist* of each user i could be defined as follows

$$H_i \;=\; \{c_i\} \cup \{\delta \in \Delta_{\ell,i} : X_{x,i} = 1\} \cup \{\delta \in \Delta_{r,i} : X_{x,i} = 1\}, \quad |H_i| = k+1.$$

Let

$$H_i = \{h_{i,1}, h_{i,2}, \ldots, h_{i,k+1}\} \quad \text{and} \quad (h_{i,(1)}, h_{i,(2)}, \ldots, h_{i,(k+1)})$$

be the same elements arranged so that

$$h_{i,(1)} < h_{i,(2)} < \cdots < h_{i,(k+1)}.$$

We then define

$$h_{i,l} = h_{i,(1)}, \quad h_{i,r} = h_{i,(k+1)},$$

where $h_{i,l}$ is the smallest (left-most) CT and $h_{i,r}$ the largest (right-most).

Any CT within the window that is *not* in H_i is treated as a *trap*. We define the following *window* of interest

$$W \;=\; \{h_{i,l}, \ldots, h_{i,r}\} \;\subseteq\; \mathbb{Z}_w.$$

traps are then the positions in W_i that were *not* selected as honeytokens:

$$T_i \;=\; W_i \setminus H_i \;=\; \{x \in W_i : x \notin H_i\}$$

Unlike honeytokens, trap positions are *not* stored; rather, they are implicitly inferred for each user by $h_{i,l}$, $h_{i,r}$ and sample outcomes.

3.6 Honeytoken & Trap Functionalities

HONEYTRAPS utilizes both *traps* and *honeytokens* to distinguish an actual breach from other events that may occur in the lifetime of the system (e.g. a false-postive attacker trying to trigger a flase breach alarm, a user miss-typing their password or even credential stuffing adversaries that have leaked the sweetlist of a user that uses the same password on a different service that also employs HONEYTRAPS).

When a submitted password is hashed and reduced modulo w, it maps to a token $g \in w$. If $g \notin H_i$ and $g \notin W_i \setminus H_i$, the server simply rejects the login. If $g \in H_i$, the attempt corresponds to either a honeytoken or the valid credential token; in this case, the system executes the procedure in Chapter 3.6.2. Finally, if $g \in W_i \setminus H_i$, a trap has been activated and the system follows the steps outlined in Chapter 3.6.1. Algorithim 1 clearly outlines the two outcomes the system may follow.

3.6.1 Actions Upon Trap Activation

When a trap is activated for account i, one of two events has occurred:

1. A false-positive attacker (e.g., an adversary who registered a new account solely to trigger traps) has submitted a trap value while trying to hit a honeytoken, or

2. A credential-stuffing adversary is targeting that specific account.

The probability that a legitimate user accidentally selects a trap—say, through a typo—is effectively zero. Let T_i be the set of traps for user i and w the total tokens; then

$$\Pr[\text{legitimate typo triggers trap}] = \frac{|T_i|}{w} \approx \frac{1189 - 120}{11.3 \times 10^9},$$

Algorithm 1 Process Login Attempt for Account i

Require: submitted token g, sweetlist H_i, window W_i, true token index c_i

if $g \in H_i$ **then**
    $valid \leftarrow$ HONEYCHECKER$(i, \text{index}(g, H_i))$
    if $valid =$ **true then**
        Grant access
    else
        HANDLEHONEYTOKENACTIVATION(i, g)
    end if
else if $g \in W_i \setminus H_i$ **then**
    HANDLETRAPACTIVATION(i, g)
else
    Reject login; display "Wrong username and/or password."
end if

where the numerator statistics is explained in Chapter. 4.1.3 and the denomenator value is discussed in Chapter. 4.1.2. (Token space selection)

Upon *trap* activation, HONEYTRAPS enforces the following sequence:

1. **Login rejection.** Deny access and display a generic error (e.g., "Wrong username and/or password").

2. **Alarm suppression for the account.** Temporarily disable further breach alarms on this account, even if additional honeytokens or traps are later triggered.

3. **Password reset prompt.** Require the account owner to reset their password immediately; the account may be optionally locked until this reset completes. The user is informed that unusual activity was detected and a password change is necessary.

4. **Global trap counter increment.** Increment a system-wide counter of trap hits. This counter influences whether subsequent honeytoken activations should raise a full breach alarm (see Chapter 4.3).

5. **Re-enable alerts after verification.** If the account has just undergone a password reset, further trap activations nor honeytoken triggers for that acoount increment the counter until the account's legitimacy is confirmed via standard spam or Sybil-detection techniques [2, 14, 37]. Once verified, honeytoken-based breach alerts for that account are reactivated.

Algorithm 2 HandleTrapActivation

Require: account identifier i, submitted token g, sweetlist H_i, window W_i

Ensure: take appropriate actions if a trap is activated

```
REJECTLOGIN(i)
PROMPTPASSWORDRESET(i)
if ¬CANINCREMENTCOUNTERS(i) then                ▷ Incrementing suspended
    return
end if
SUSPENDALARMS(i)        ▷ Disable alarm capabilities and global trigger influence
trapCounter ← trapCounter + 1
```

3.6.2 Actions Upon Honeytoken Activation

When a honeytoken is activated for account i, one of two events has occurred:

1. A false-positive attacker (e.g., an adversary who registered a new account solely to trigger traps) has submitted a carefully crafted password to hit a honeytoken and has succeded, or

2. An adversary that has *leaked* the credential database has hit a honeytoken mistakenly thinking it has the valid credential token.

The probability that a legitimate user accidentally selects a honeytoken—say, through a typo—is effectively zero. Let H_i be the sweetlist for user i and w the total tokens; then

$$\Pr[\text{legitimate typo triggers trap}] = \frac{|H_i| - 1}{w} \approx \frac{120 - 1}{11.3 \times 10^9},$$

where the value of the numerator $|H_i|$ (i.e. size of the sweetlist) is derived in Chapter. 4.1.3 and the value of the denomenator w are derived in Chapter. 4.1.2

Upon *Honeytoken* activation, HONEYTRAPS enforces the following sequence:

1. **Login rejection.** Deny access and display a generic error (e.g., "Wrong username and/or password").

2. **Alarm suppression for the account.** Temporarily disable further breach alarms on this account, even if additional honeytokens or traps are later triggered.

3. **Password reset prompt.** Require the account owner to reset their password immediately; the account may be optionally locked until this reset completes. The user is informed that unusual activity was detected and a password change is necessary.

4. **Global trap counter increment.** Increment a system-wide counter of honeytoken hits. This counter influences whether subsequent honeytoken activations should raise a full breach alarm (see Chapter 4.3).

5. **Check for breach.** The system then evaluates Eq. 4.3 to determine whether a breach alarm should be raised. If the equation holds, a breach alarm is confidently confirmed.

6. **Re-enable alerts after verification.** If the account has just undergone a password reset, further trap activations nor honeytoken triggers from that specific account increment the counter until the account's legitimacy is confirmed via standard spam or Sybil-detection techniques [2, 14, 37]. Once verified, honeytoken-based breach alerts for that account are reactivated.

Algorithm 3 HandleHoneytokenActivation

Require: account identifier i, submitted token g, sweetlist H_i

Ensure: take appropriate actions if a honeytoken is activated

```
REJECTLOGIN(i)
PROMPTPASSWORDRESET(i)
if ¬CANINCREMENTCOUNTERS(i) then                ▷ Incrementing suspended
    return
end if
SUSPENDALARMS(i)          ▷ Disable alarm capabilities and global trigger influence
honeytokenCounter ← honeytokenCounter + 1
UPDATEGLOBALTHRESHOLD(void)
if globalThreshold < honeytokenCounter / trapCounter then
    RAISEBREACHALARM(void)
end if
```

Chapter 4

Implementation

4.1 Hyperparameter Tuning

4.1.1 Setting the Success Probability p

HONEYTRAPS assigns honeytokens by performing independent Bernoulli trials on each CT with success probability p. In this section, we derive the optimal p that simultaneously satisfies the online and offline security constraints from Chapter 3.3 and maximizes the expected overlap of honeytokens across different servers to resist intersection attacks.

Online adversaries who already know a user's true password—whether through phishing, a compromised third-party site, or even by being the account owner—can exploit the public CT mapping in HONEYTRAPS to provoke false breach alarms. Since the number of honeytokens is predefined and the Bernoulli process begins adjacent to the valid CT, the tokens subjected to the Bernoulli process first have a higher probability of being selected as honeytokens. An attacker with full knowledge of the hash-and-modulo algorithm can therefore craft a single targeted guess to maximize the chance of hitting a honeytoken and triggering an alarm.

We therefore define a more advanced online adversary, denoted FP^+, whose goal is to force a false breach alarm by taking advantage of HONEYTRAPS's randomized honeytoken placement. FP^+ examines the CT values immediately surrounding the genuine token and, knowing the Bernoulli selection probability, identifies the single candidate most likely to be a honeytoken. The adversary is constrained to one "best-case" guess: if this choice is not a honeytoken, it must be a trap, instantly triggering the defensive actions outlined in Chapter 3.6.1. FP^+ uses every piece of public information—the user's actual password and the honeytoken-generation algorithm—it represents the optimal false-alarm strategy for an attacker without access to the credential database; in Chapter 4.5.1 we prove why this is true while also provide a more foraml defition of this adversary's capabilities and limitations.

In the optimal false-positive attack, FP^+ succeeds with probability p; hence we must impose an upper bound on p in order to satisfy the online resistance requirement (Eq. 3.4). We therefore choose

$$p \leq 10^{-1}.$$

An attacker might attempt to boost their odds by registering multiple spam accounts. However, since each account independently triggers a honeytoken with probability p, the expected number of accounts needed to achieve one successful false alarm is

$$\frac{1}{p} \approx 10,$$

i.e. about 10 accounts on average. Such large-scale registrations and login patterns would be readily identified by existing anomaly- and spam-detection systems [1, 2, 14]. Furthermore, trap activations provide a backstop: if the combined count of honeytoken and trap triggers exceeds a system-wide threshold, a site-wide breach alert is raised (see Sec. 4.3). Together, these defenses make $p = 0.1$ a practical and effective upper bound. If we relax the per-user false-positive bound of $p = 10^{-1}$—which is applied solely for FP^+—and rely on system-wide monitoring, p can be raised further, substantially boosting inter-server intersection rates while still preserving breach-detection guarantees and a low false-positive rate (see Chapter 5.1.1 for a detailed discussion).

Moreover, p should be tuned to increase the intersection of CTs across servers where a user has reused the same password. To quantify how the Bernoulli parameter p affects the overlap of Credential Token (CT) sets across multiple servers (when a user reuses the same password), we performed a large-scale Monte Carlo simulation. Concretely, for a given triple (s, m, p), where s is the number of servers on which the user has accounts with the same password, m is the total number of CTs per account $|H_i|$, and p is the Bernoulli success probability for selecting any candidate CT as a honeytoken. We ran $N = 10^6$ independent trials. Furthermore, we set $s = 2$ and $m = 40$ to focus the investigation on how the value of p effects the intersection accross different servers. Each trial proceeds as follows:

1. **Generate the real CT.** We hash the fixed password (e.g. the string "password") with SHA-256, interpret the 256-bit digest as an integer, and reduce it modulo the token-space size $w = 11.3 \times 10^9$, yielding the true CT (c).

2. **Simulate each server's sweetlist.** For each of the s servers, we invoke the HONEYTRAPS honeytoken-generation procedure (with parameters m, p and w) to produce a sweetlist of size m that always contains c and samples from the $w - 1$ other CTs via a Bernoulli-scan as described in Chapter 3.4 and 3.5.

3. **Compute the intersection.** We take the set intersection of the $s = 2$ sweetlists and record its cardinality $\kappa \in \{1, \ldots, m\}$.

4. **Accumulate statistics.** For each $k = 1, \ldots, m$, we increment a counter if $\kappa \geq k$. Periodically and at the end, we normalize these counters by N to estimate

$$\Pr\left[\left|\bigcap_{j=1}^{2} H_j\right| \geq k\right],$$

where H_j is the sweetlist on server j.

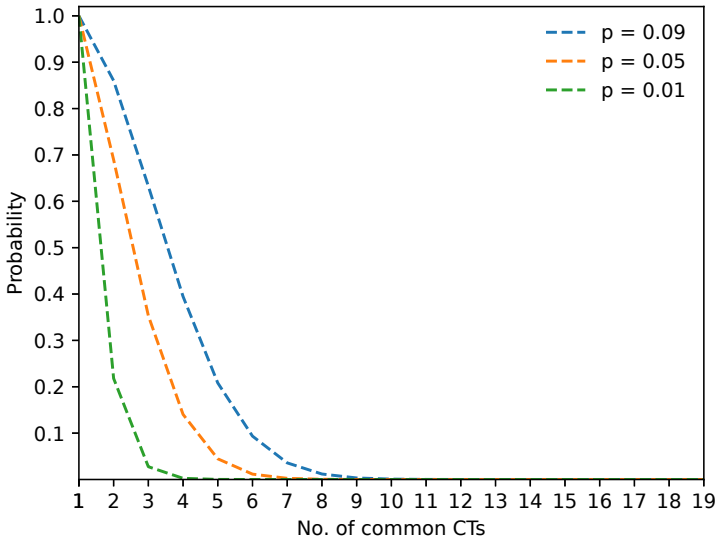


Figure 4.1: Probability of having at least n common CTs ($1 \leq n \leq 40$) in the intersection of two sweet-CT lists, where honeytokens are selected via a Bernoulli process with probability $p \in \{0.01, 0.05, 0.1\}$.

By sweeping p over the range $[0, 0.1]$ and varying s and m, we obtain curves of the form shown in Figure 4.1, which plot the probability of having at least k common CTs as a function of p. These results guide the choice of p that maximizes cross-server intersection (to resist intersection attacks) while still satisfying the security bounds from Section 3.3

As Figure 4.1 illustrates, increasing p raises the expected number of shared CTs between servers. Consequently, we adopt $p = 0.1$—the maximum value compatible with our online resistance requirement—to maximize cross-server intersection. Nevertheless, HONEYTRAPS remains fully configurable: system operators may choose a different p to satisfy alternative security or usability constraints. In Chapter 4.2, we present a decision framework to guide the selection of p and other parameters in accordance with an organization's specific threat model and operational priorities.

4.1.2 Setting the token space w

Rather than assuming independent draws with replacement, we model an online attacker's λ guesses as sampling without replacement from the finite pool of w possible CTs. This reasoning is because if the attempted CT is not the genuine credential token, the adversary would not retry the same CT twice. In this setting, the chance of hitting a particular subset of size k in λ draws follows the hypergeometric distribution. We apply two hypergeometric bounds to ensure both real-CT compromises and false-positive alarms remain below our threshold $\alpha = 10^{-1}$.

1. Genuine-CT compromise. There is exactly one genuine CT among the w tokens. If the depth-first attacker draws $\lambda_d = 10^6$ distinct CTs, the probability of *not* drawing the real CT is

$$\frac{\binom{w-1}{\lambda_d}\binom{1}{0}}{\binom{w}{\lambda_d}},$$

since we choose all λ_d guesses from the $w-1$ decoys. Therefore the probability of at least one successful guess is

$$P_{\text{breach}}(\lambda_d, 1) = 1 - \frac{\binom{w-1}{\lambda_d}}{\binom{w}{\lambda_d}} \leq \alpha$$

For n_d accounts under simultaneous attack, the probability that *none* is breached is

$$\left(\frac{\binom{w-1}{\lambda_d}}{\binom{w}{\lambda_d}}\right)^{n_d}$$

To ensure that not even one of the ten accounts is compromised with probability exceeding α, we require

$$P_{\text{breach}}(\lambda_d, n_d) = 1 - \left(\frac{\binom{w-1}{\lambda_d}}{\binom{w}{\lambda_d}}\right)^{n_d} \leq \alpha \tag{4.1}$$

2. False-positive alarm. We conservatively assume that the activation of a single honeytoken is sufficient to raise an alarm. In practice, however, HONEYTRAPS estimates a honeytoken-to-trap ratio to decide whether to raise an alarm (see Sec. 4.3 for details). There are $|H_i| - 1$ honeytokens in each user's sweetlist of size $|H_i|$. An attacker making $\lambda_d = 10^6$ guesses may trigger a false alarm by drawing at least one of these $|H_i| - 1$ decoys. The probability of avoiding all honeytokens is

$$\frac{\binom{w-(|H_i|-1)}{\lambda_d}\binom{|H_i|-1}{0}}{\binom{w}{\lambda_d}}$$

so the probability of at least one honeytoken hit for an account is

$$P_{\text{fp}}(\lambda_d, 1) = 1 - \frac{\binom{w-(|H_i|-1)}{\lambda_d}}{\binom{w}{\lambda_d}} \leq \alpha$$

For n_d accounts under simultaneous attack, the probability that *none* triggers a false alarm is

$$\left(\frac{\binom{w-(|H_i|-1)}{\lambda_d}}{\binom{w}{\lambda_d}}\right)^{n_d}$$

Thus the probability of at least one false-positive breach across n_d accounts is

$$P_{\text{fp}}(\lambda_d, n_d) = 1 - \left(\frac{\binom{w-(|H_i|-1)}{\lambda_d}}{\binom{w}{\lambda_d}}\right)^{n_d} \leq \alpha \tag{4.2}$$

Lower-Bound Token-Space w. We choose the smallest integer w_b such that the combined probability of either a successful breach or a false-positive alarm across n_d accounts in a depth-first attack remains below α:

$$P_{\text{breach}}(\lambda_d, n_d) + P_{\text{fp}}(\lambda_d, n_d) \leq \alpha.$$

This ensures that, with probability at least $1-\alpha$, an attacker cannot trigger either event on any of the n_d targeted accounts.

We repeat the same hypergeometric analysis for breadth-first attackers, who make $\lambda_b = 10^4$ guesses on each of $n_b = 10^3$ accounts. Having computed the minimal w_d for depth-first attacks and w_b for breadth-first attacks, we set

$$w = \max\big(w_d, w_b\big),$$

ensuring that HONEYTRAPS meets the security requirement against both types of online adversaries. The smallest possible tokenspace is selected to increase the collisions of strings. We choose the minimal token-space space w that satisfies our security requirements in order to maximize collisions among password strings. Unlike conventional cryptographic hash functions—where collisions are exceedingly rare—HONEYTRAPS deliberately induces collisions so that multiple, human-chosen passwords map to the same honeytoken. This design ensures that each CT corresponds to a diverse set of plausible passwords, making it significantly more difficult for an adversary to pinpoint the true credential token.

Therefore, HONEYTRAPS must navigate a critical trade-off: on one hand, it must limit the probability that an online adversary either successfully breaches an account or triggers a false alarm to the bounds established in prior work [13,35]; on the other hand, it must maximize the number of collisions—i.e. the number of distinct, plausible passwords

mapping to each CT—to impede offline attackers from isolating the true credential. This approach departs from the original honeywords scheme, where each decoy is a single decoy password, and from Bernoulli Honeywords [35], which deliberately keep the false-positive rate extremely low.

4.1.3 Setting the number of honeytokens

By combining Eqs. (4.2) and (4.1), we compute the smallest token-space size w that satisfies the online and offline security constraints from Section 3.3 for any given number of CTs per user. Choosing the CT count entails balancing greater cross-server overlap against increased collisions with human-chosen passwords and additional storage. To explore this trade-off, we simulated sweetlists containing 40, 80, and 120 CTs—each with its own w tuned to our security bounds—and measured the probability that two independent servers share at least one CT. We carried out a second large-scale Monte-Carlo experiment, this time fixing the number of servers and the Bernoulli parameter ($s = 2$ and $p = 0.1$) so as to isolate the influence of the sweet-list size m on cross-server intersection. Following the same protocol described in Chapter 4.1.1, we executed $N = 10^6$ independent trials for each candidate value of m. As Figure 4.2 illustrates, the likelihood of common tokens rises steadily with the number of CTs per user. To balance the aforementioned trade-offs, we set $|H_i| = 120$ and by extension using the reasoning in Chapter 4.1.2 we set $w = 11.3 \times 10^9$.

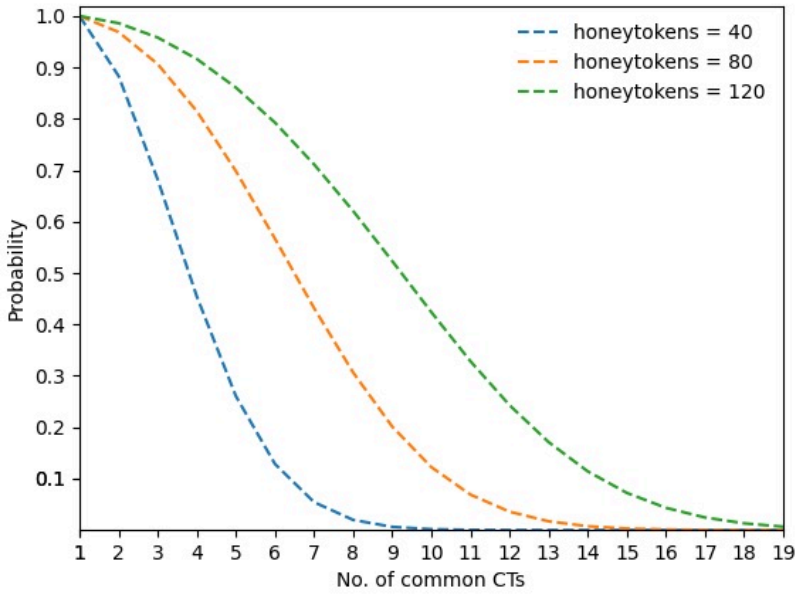


Figure 4.2: Probability of having at least n common CTs ($1 \leq n \leq 120$) in the intersection of two sweet-CT lists, each containing $x \in \{40, 80, 120\}$ CTs per account.

Notably, $|H_i| = 120$ is only a recommendation, as this parameter is tunable within the HONEYTRAPS framework based on the defined security requirements. If more collisions are needed and less storage is available, the number of CTs can be adjusted to fit *any* use case. Consequently, when setting $w = 11.3 \times 10^9$ and $|H_i| = 120$, experimental results show that, on average, 1,189 tokens lie between the leftmost and rightmost honeytokens. This observation aligns with theoretical expectations, as the probability of selecting a honeytoken is approximately $\frac{120}{1189} \approx 0.1$, which matches the value of p.

4.2 Rules of Thumb for Selecting Hyperparameters

The following step-by-step procedure helps system operators choose HONEYTRAPS's core parameters—success probability p, number of CTs per user $|\mathrm{H_i}|$, and token-space size w—in accordance with their unique security requirements:

1. **Define Security Requirements:** Establish security goals and constraints, as outlined in Sec. 3.3, for online/offline attackers. While this paper follows Florêncio et al. [13] guidelines, Eqs. 3.4 & 3.5 can be adjusted to accommodate specific use cases or security requirements.

2. **Set Success Probability (p):** Adjust p to balance the trade-off between maximizing intersections and minimizing false alarms. Increase p for more intersections, or decrease it to reduce false alarms.

3. **Set Number of Honeytokens Per User ($|H_i|$):** Optimize $|H_i|$ to balance intersections and storage overhead, while increasing collisions per CT. Decrease $|H_i|$ to increase collisions and reduce storage, or increase it for more intersections and higher security guarantees against intersection/credential stuffing attacks. Note that changing $|H_i|$ requires adjusting the range w.

4. **Configure CT Value Range (w):** Set w using Eqs. 4.2 and 4.1 to meet security requirements. The value of w depends on the security goals from Step (1) and the number of tokens per user ($|H_i|$).

These rules of thumb provide a practical framework for customizing HONEYTRAPS's hyperparameters to match diverse threat models and resource constraints.

4.3 Detecting FP^+ via System Threshold

An online adversary who has not breached the credential database but aims to trigger a false alarm can adopt two specific strategies. The first involves submitting random strings

in an attempt to hit a honeytoken, without considering the position of the valid CT. This naive approach has a negligible success probability:

$$\frac{|H_i| - 1}{w} = \frac{119}{11.3 \times 10^9},$$

A more sophisticated strategy leverages the position of the valid CT to exploit HONEYTRAPS's probabilistic design. This adversary, referred to as FP^+, knows the real password and, by extension, the position of the valid CT. Thus, FP^+ can submit strings that map to tokens near the valid CT, hoping that some were selected as honeytokens(Chapter 3.4).

Even with this *optimal* strategy, FP^+ can be easily identified due to the ratio of trap activations compared to honeytokens. To illustrate this, we simulated an FP^+ attack by creating 1 million accounts and recording the total number of trap and honeytoken triggers at each step. Each account was limited to a single login attempt, since triggering a trap exempts that account from raising a breach alarm in subsequent attempts (Chapter 3.6.1). We repeat this experiment ten times and report the average no. of trap and honeytoken activations in Fig. 4.3. A more formal definition of FP^+'s optimality, capabilities and attack stardegy is defined in Chapter 4.5.1.

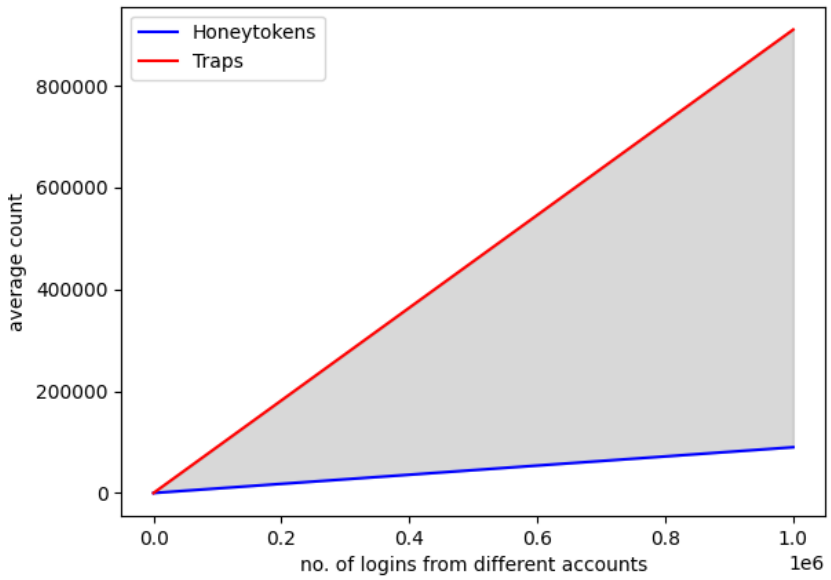


Figure 4.3: Ratio gap between trap & honeytoken triggers during an FP^+ attack.

As shown, the ratio of honeytoken to trap triggers differs significantly. Importantly, an attack by FP^+ *cannot* deviate from this pattern, as the attacker relies solely on the probability that tokens adjacent to the valid CT were selected as honeytokens. Any deviation from this ratio would allow the operator to confidently conclude that the credential database has been breached. Conversely, if the honeytoken-to-trap trigger ratio adheres

to the expected pattern, the system disregards the possibility of a breach. Specifically, the system-wide threshold for triggering a breach alarm is defined as:

$$\frac{\text{Number of Honeytoken Activations}}{\text{Number of Trap Activations}} > p \cdot \alpha \tag{4.3}$$

Here, p represents the probability with which a CT is selected as a honeytoken during the registration phase as descirbed in Chapter. 3.5, while α is a damping factor introduced to mitigate the cold start problem, as discussed in Chapter. 5.1.

One might argue that an offline adversary who has breached the credential database could attempt to evade detection by masquerading as an FP^+, deliberately triggering traps to avoid suspicion. However, this strategy is highly impractical: approximately 90% of affected accounts would be notified to reset their passwords (see Chapter 3.6.1), leaving the attacker with access to only about 10%—and that is assuming they successfully distinguish the valid CT from the honeytokens on the second attempt. For each subsequent attempt, the attacker would sacrifice an additional 90% of the remaining accounts per round to stay undetected. Consider a service with 10 million registered users. If the attacker triggers a honeytoken for 10% (1M) of accounts, they must forfeit access to the remaining 90% (9M) to avoid detection. If they fail to correctly guess the valid CT on the second attempt, another 90% (900K) of the remaining accounts would be lost. With each iteration, the pool of unnotified accounts shrinks significantly, rendering the breach effort increasingly unproductive. Simply put, to remain undetected, the attacker would not only cut off an arm, but nearly sacrifice the entire body.

Additionally, false-positive adversaries typically control only a fixed percentage of accounts—for example, 20% of Twitter accounts were reported as spam or fake [27]. Thus, if a significant number of trap activations occur (e.g., 9M in the previous example during the first round), it is reasonable to conclude that the credential database has been compromised and that an offline attacker is attempting to obscure their actions by deliberately triggering traps. We formalize this threat as the *hidden-offline* attacker—describing their capabilities and tactics—in Chapter 4.5.3.

4.4 Django-Honeytraps

In this section we describe the key implementation details of *HoneyTraps*, our Django-based honeyword authentication framework. We cover the custom user model, the honeytoken statistics, the honeychecker service, the migration strategy (including legacy-user support), and finally packaging for reuse.

4.4.1 Custom User Model

We extend Django's built-in `AbstractUser` to replace the standard password field with a JSON field of 120 integers (i.e. the users sweetlist). We additionally include a HoneytokenStats class to keep track of significant events that are used globaly within the system to detected a breach. Moreover we override the set_password and check_password to implement the custom logic of HONEYTRAPS. The model lives in `honeytraps/models.py`:

```
1   from django.db import models
2   from django.conf import settings
3   from django.contrib.auth.models import AbstractUser
4
5   w            = getattr(settings, "HONEYTRAPS_W", int(11.3 * pow(10, 9)))
6   total_cts    = getattr(settings, "HONEYTRAPS_TOTAL_CTS", int(11.3 * pow(10, 9)))
7   window_size  = getattr(settings, "HONEYTRAPS_WINDOW_SIZE", 120)
8   success_prob = getattr(settings, "HONEYTRAPS_SUCCESS_PROB", 0.09)
9   ignore       = getattr(settings, "HONEYTRAPS_IGNORE", 10)
10
11  DECAY_FACTOR = getattr(settings, "HONEYTRAPS_DECAY_FACTOR", 0.99)
12  p = getattr(settings, "HONEYTRAPS_P", success_prob)
13
14  class HoneytokenStats(models.Model):
15      """
16      Stats of system used to determine breach
17      """
18      global_count_ht     = models.PositiveIntegerField(default=0)
19      global_count_traps  = models.PositiveIntegerField(default=ignore)
20      alpha               = models.FloatField(default=round(1 / success_prob, 2))
21
22  class CustomUser(AbstractUser):
23      """
24      Custom user model that replaces the traditional password field with a field
25      with 120 stored integers
26      """
27      password            = None
28      password_tokens     = models.JSONField(default=list, blank=True)
29      valid_account       = models.BooleanField(default=True)
30      honeyword_attempts  = models.PositiveIntegerField(default=0)
31
32      def set_password(self, raw_password):
33          ...
34      def check_password(self, raw_password):
35          ...
```

Listing 4.1: CustomUser and HoneytokenStats models

4.4.2 Honeychecker Service

HONEYTRAPS does not require the always-trusted honeychecker component to operate as it can be seamlessly paired with AMNESIA [34] and Lethe [9]. However in this prototype we utilize this always trused component for demonstaration. The honeychecker is a separate HTTP service (in `honeytraps/honeychecker_server.py`) with two endpoints:

- `/honeychecker/store` - stores the valid token index for each user
- `/honeychecker/check` - validates a presented token index

It can be launched via the management command:

```
$ python manage.py run_honeytraps
```

The honeychecker is invoked only when a CT maps to one of the elements in the user's sweetlist. It then checks if the submitted token is in fact the valid credential. In production, the service is typically run behind an HTTPS reverse proxy (e.g., Nginx).

4.4.3 Migration Strategy

Initial Schema The first migration (`0001_initial.py`) creates the tables for `CustomUser` and `HoneytokenStats`. It depends on Django's latest auth migration to ensure compatibility.

Legacy-User Data Migration For projects upgrading from the default `auth.User`, we include a data migration (`0004_migrate_existing_users.py`) that: (a) Checks if any `auth.User` records exist. If none, aborts immediately (no action on new projects). (b) For each legacy user, copies `username`, `email`, etc. into the new `CustomUser` table. (c) Converts the old password hash to an integer via:

$$\text{hash\_int} = \texttt{int}(\text{hex\_digest}, 16), \quad \text{valid\_token} = \text{hash\_int} \bmod w.$$

Generates 120 tokens with `calc_lists_of_tokens` and populates `password_tokens`.

```
def migrate_existing_users(apps, schema_editor):
    OldUser = apps.get_model('auth', 'User')
    NewUser = apps.get_model('honeytraps', 'CustomUser')

    if OldUser.objects.count() == 0:
        return  # No legacy data  skip

    from django.conf import settings
    w           = settings.HONEYTRAPS_W
    window_size = settings.HONEYTRAPS_WINDOW_SIZE
    total_cts   = settings.HONEYTRAPS_TOTAL_CTS
    prob        = settings.HONEYTRAPS_SUCCESS_PROB

    from honeytraps.utils import calc_lists_of_tokens

    for old in OldUser.objects.all():
        new = NewUser(
            id=old.id, username=old.username, email=old.email,
            is_active=old.is_active, date_joined=old.date_joined
        )
        # derive valid_token from old.password...
        hash_int  = hash_to_int(old.password)
```

```
23          valid      = hash_int % w
24          tokens     = calc_lists_of_tokens(valid, window_size, total_cts, prob)[0]
25          new.password_tokens = tokens
26          new.save()
```

Listing 4.2: Key part of data migration

4.4.4 Packaging and Distribution

To simplify deployment and encourage reuse, we publish `django-honeytraps` on PyPI. The project layout includes:

- `setup.py`: Define package metadata (name, version, author), dependencies, and build requirements.
- `MANIFEST.in`: Ensures that database migrations, templates, and utility scripts are included in source distributions.

Installation: Install the HONEYTRAPS pip module:

```
1 $ pip install django-honeytraps
```

Configuration: Add the app and custom user model to your Django settings:

```
1 INSTALLED_APPS = [
2     # ... other apps ...
3     'honeytraps',
4 ]
5
6 AUTH_USER_MODEL = 'honeytraps.CustomUser'
```

Initialization: Run the migrations to create the necessary tables:

```
1 $ python manage.py migrate
```

This process yields a plug-and-play honeyword framework ready for immediate use in any Django project.

4.5 Modeling Attackers

In this section, we formally define the adversaries that a system deploying HONEYTRAPS may encounter. We model each attacker as an n-shot game to reflect the strategic nature of their behavior. This approach helps us reason about their capabilities and constraints, and also enables formal analysis of attack optimality—such as proving the effectiveness of FP^+. Modeling adversaries in this way is common in attacker-defender literature [11, 21, 28], as it provides a principled framework for analyzing decision-making

under uncertainty. After presenting each attacker model, we evaluate their effectiveness against a simulated HONEYTRAPS-enabled system in Chapter 5.1.

4.5.1 FP^+

We now formalize the capabilities, objectives, and constraints of the FP^+ adversary, who seeks to trigger a false breach alarm in the HONEYTRAPS framework.

Knowledge. FP^+ has full knowledge of the HONEYTRAPS parameters—namely, each user's sweetlist size $|H_i|$, the token-space size w, and the honeytoken selection probability p. We assume these hyperparameters are publicly documented or otherwise leaked. Moreover, FP^+ controls n user accounts and thus knows each account's password (e.g. obtained via phishing or self-registration for fake accounts).

Goal. The adversary's objective is to induce a system-wide breach alarm. Concretely, they must cause enough honeytoken activations—across the n accounts they control—to exceed the system's global alarm threshold defined in Chapter 4.3.

Strategy. Because HONEYTRAPS applies its Bernoulli selection first to the CT values immediately adjacent to the true credential c, these neighbors have the highest probability of being chosen as honeytokens. With that in mind FP^+ computes submits that single guess g in an attempt to trigger a false breach alarm.

Limitations. Each account allows only a single effective guess: if the submitted CT is not in the sweetlist, it must lie in the trap region $W_i \setminus H_i$, triggering the trap-activation routine (Chapter 3.6.1) disabling any further attempts on that account. Likewise, even a honeytoken hit consumes the one allowed guess (Chapter 3.6.2). Thus, FP^+ can make at most one informed guess per account.

Two-Party n-Shot Game for FP^+

To better understand the attack stradegy of this adversary we define a game between the **System** and the **Challenger**, who acts as the FP^+ adversary:

System Setup: The System initializes public parameters (w, m, p) and damping factor α; where w is the token-space, m is the number of hoenytokens and p is the probability used for selecting a honeytokens as described in Chapter 3.5. For each of the n accounts controlled by the Challenger, it computes

$$c_i = \text{SHA3-256}(pw_i) \bmod w, \quad H_i = \{c_i\} \cup \{h_{i,1}, \ldots, h_{i,m-1}\} \subseteq \mathbb{Z}_w.$$

It keeps each H_i and c_i secret and initializes counters $HT = TR = 0$.

Challenger's Guess Phase: The Challenger, knowing $\{pw_i\}$ and (w, m, p, α) but not $\{h_{i,j}\}$, submits one guess $g_i \in \mathbb{Z}_w$, $g_i \neq c_i$, for each account i, over time or all at once. Specifically, we define the neighbor-offset set $S = \{\pm 1, \pm 2\}$, the adversary samples uniformly $s_i \sim \text{Uniform}(S)$ and selects a string such that it maps to $g_i = (c_i + s_i)$. No intermediate feedback is given between each submision.

System Resolution: Upon each submision $(g_1, \ldots, g_n)$, for each i the System computes

$$X_i = \begin{cases} 1, & g_i \in H_i \setminus \{c_i\} \quad \text{(honeytoken hit)}, \\ 0, & \text{otherwise}, \end{cases} \qquad Y_i = 1 - X_i \quad \text{(trap activation)}.$$

and updates $HT \mathrel{+}= X_i, \quad TR + Y_i$. In case $X_i = 0$ a damping factor,namely α, is updated lowering the threshold. The rational behind α is established during the evaluation of the system in Chapter 5.1. After each evaluation, it raises a breach alarm if and only if

$$\frac{HT}{TR} > p \cdot \alpha.$$

It returns a single bit $\mathsf{Win} = 1$ if an alarm was raised, else $\mathsf{Win} = 0$.

Challenger's Success: The Challenger "wins" exactly when $\mathsf{Win} = 1$; They "lose" if *none* of the n guesses return $\mathsf{Win} = 1$.

Optimality of FP^+'s Attack Strategy

Under the constraints that FP^+ may submit exactly one guess g_i per account i and receives no intermediate feedback, the adversary's success probability on account i is

$$\Pr\big[g_i \in H_i \setminus \{c_i\}\big] = P_{i,j},$$

where $g_i = n_{i,j}$ is the jth neighbor of c_i in the Bernoulli-scan order. Recall from Chapter 3.5 that honeytokens are chosen by performing independent Bernoulli trials on the nearest CT values: first $c_i + 1$, then $c_i - 1$, then $c_i + 2$, and so on, but the process halts as soon as r_{win} honeytokens have been selected to the right of c_i (similarly l_{win} to the left). Because at least r_{win} trials will always be conducted on right-side neighbors, the inclusion probability is the same for the first r_{win} positions:

$$P_{i,1} = P_{i,2} = \cdots = P_{i,r_{win}}.$$

However, the $(r_{win} + 1)$-th neighbor is only tested if fewer than r_{win} successes occurred among the first r_{win} trials, making its inclusion probability strictly lower:

$$P_{i,1} > P_{i,r_{win}+1} > P_{i,r_{win}+2} > \cdots$$

An analogous argument applies to the left-side neighbors and ℓ_{win}. Consequently, the single-guess strategy

$$g_i^* \;=\; \arg\max_{n_{i,j}} P_{i,j} \;=\; n_{i,1} \;=\; c_i + 1$$

(or symmetrically $c_i - 1$) maximizes the per-account honeytoken-hit probability. No alternative choice $g_i \neq g_i^*$ can yield a higher $\Pr[g_i \in H_i]$, proving that FP^+ is optimal under a one-guess constraint. If the adversary has control over n accounts they will by extension perform n optimal one-guess attempts.

This maximizes the adversary's overall chance of triggering the system-wide breach alarm. Hence, the FP^+ strategy of guessing the immediate neighbor of the true credential token on each account is provably optimal in the n-shot game.

4.5.2 Baseline-Offline Adversary

Knowledge. The offline adversary has leaked the credential database. Therefore, has obtained sweetlist H_i of each $user_i$. Moreover, they have full knowledge of the HONEYTRAPS parameters—namely, each user's sweetlist size $|H_i|$, the token-space size w, and the honeytoken selection probability p.

Goal. Their objective is to identify the genuine credential token c_i for each user without ever triggering a breach alarm.

Strategy. They must submit a string that maps to one of the credential in H_i for each user. Each element in H_i maps to multiple strings; unlike cryptographic hash functions. Hence, the attacker faces multiple probable passwords and resorts to selecting one of the elements by sampling uniformly at random from H_i.

Limitations. Each account allows only a single effective guess: if the submitted CT is not the true credential token of the user c_i then it is a decoy causing the system to perform the hoenytoken-activation routine (Chapter 3.6.2) notifying the user to change their password. Thus, the baseline-offline adversary can make at most one informed guess per account. The adversary could potentialy gain access before the user resets their password, however the access granted would be short-lived. The user can even lock their account once notified allowing them to even more swiftly block the malicious actor.

Two-Party n-Shot Game for the Baseline-Offline Adversary

System Setup: Public parameters (w, m, p) are fixed. For each of the n accounts the System chooses

$$c_i = \text{SHA3-256}(pw_i) \bmod w, \qquad H_i = \{c_i\} \cup \{h_{i,1}, \ldots, h_{i,m-1}\} \subseteq \mathbb{Z}_w,$$

and discloses the entire set H_i to the Challenger while keeping the index of c_i secret. Additionally initializes counters $HT = TR = 0$.

Challenger's Guess Phase: Knowing $\{H_i\}_{i=1}^{n}$ but *not* the locations of the c_i, the Challenger submits a single guess

$$g_i \in H_i$$

for each account i, with feedback between guesses.

System Resolution: For every account the System computes

$$X_i = \begin{cases} 1, & g_i = c_i \quad \text{(correct, no alarm)}, \\ 0, & g_i \neq c_i \quad \text{(honeytoken, notify user)}. \end{cases}$$

and updates $HT \mathrel{+}= X_i$. In case $X_i = 0$ a damping factor,namely α, is updated lowering the threshold. The rational behind α is established during the evaluation of the system in Chapter 5.1. After each evaluation, it raises a breach alarm if and only if

$$\frac{HT}{TR} > p \cdot \alpha.$$

It returns a single bit after each guess $\mathsf{Win} = 0$ if an alarm was raised, else $\mathsf{Win} = 1$.

Challenger's Success: The Challenger "wins" if the system does not return $\mathsf{Win} = 0$ after a target of n accounts have been breached; They "lose" if *any* of the n guesses return $\mathsf{Win} = 0$.

Challenger's Incentive for Selecting a Large n

The offline attacker is not merely interested in *staying undetected*; their real payoff comes from the number of accounts they successfully compromise. With less attempts it is less likely that the adversary triggers enough honeytokens to surpass the threshold and raise a breach alarm. However with fewer attempts the Challenger also receives fewer Win = 1 outcomes and by extension less accounts are potentialy breached. Therefore there exists an incentive for selecting a large n. Even though remaining undetected is key the payoff of exfiltrating the credential database lies within the aspect of compromising as many accounts as possible.

4.5.3 Hidden-Offline adversary

As noted in Chapter 4.3, the use of a system-wide alarm threshold opens the door to a subtler adversary. Such an attacker *has already exfiltrated the credential database* yet tries to masquerade as a false-positive actor. Concretely, they breach accounts by guessing real

tokens but, whenever they accidentally hit a honeytoken, they deliberately trigger traps on other accounts to keep the honeytoken-to-trap ratio below the alarm threshold, thereby hiding their activity.

Although we argue that this "hidden-offline" strategy squanders most of the attacker's effort, we nevertheless formalise the model and evaluate it against HONEYTRAPS to confirm that the defence remains effective.

Knowledge. The hidden offline adversary has leaked the credential database. Therefore, has obtained sweetlist H_i of each $user_i$. Moreover, they have full knowledge of the HONEYTRAPS parameters—namely, each user's sweetlist size $|H_i|$, the token-space size w, and the honeytoken selection probability p.

Goal. Their objective is to identify the genuine credential token c_i for each user without ever triggering a breach alarm.

Strategy. For every user i the adversary must supply a string whose credential token lies in the leaked set H_i. Because each token in H_i has many pre-images (unlike a cryptographic hash, which is designed to be pre-image resistant), the attacker has several plausible strings to choose from. Lacking any distinguisher, they pick a token uniformly at random from H_i and craft a password that maps to it.

If a submitted token turns out to be a honeytoken, the attacker deliberately "sacrifices" other controlled accounts by submitting strings that fall into their trap regions $W_j \setminus H_j$. These extra trap activations enlarge the global denominator, keeping the honeytoken-to-trap ratio below the alarm threshold and allowing the adversary to continue probing the remaining accounts.

Limitations. Each account allows only a single effective guess (honeytoken or trap): hitting a trap or hoenytoken causes the system to to perform the trap and hoenytoken-activation routines in Chapters 3.6.1 and 3.6.2 respectively.

Two-Party n-Shot Game for the Hidden-Offline Adversary

We model the interaction between the **System** (which enforces HONEYTRAPS) and the **Challenger** (the hidden-offline attacker) as an n-shot game.

System Setup: Public parameters (w, m, p, α) are fixed. For each of the n leaked accounts, the System already stores

$$c_i = \text{SHA3-256}(pw_i) \bmod w, \qquad H_i = \{c_i\} \cup \{h_{i,1}, \ldots, h_{i,m-1}\} \subseteq \mathbb{Z}_w.$$

It reveals the entire set H_i of each $user_i$ to the Challenger, but keeps the index of c_i secret. Global counters are initialised to $HT = 0$ (honeytoken hits) and $TR = 0$ (trap hits).

Challenger's Move: Let $A = \{1, \ldots, n\}$ be the set of all compromised accounts, each equally valuable to the Challenger. While $A \neq \emptyset$:

1. Randomly select and remove one index $i \in A$. Submit a token $g_i \in H_i$ (chosen uniformly) for that account.
2. If the System replies $X_i = 1$ (login succeeds), proceed to the next iteration.
3. If the System replies $X_i = 0$ (honeytoken hit), choose j additional indices uniformly from the remaining A, submit trap tokens for each, and remove them. The attacker selects j to be *only* as large as necessary for the System to be able to absorb one additional honeytoken activation without crossing the breach-alarm threshold, thereby avoiding the unnecessary "sacrifice" of extra accounts. This keeps $HT/TR \leq p \cdot \alpha$ so that the attacker can continue probing the rest of the accounts without triggering an alarm.

System Resolution: For every account the System evaluates

$$X_i = \begin{cases} 1, & g_i = c_i \quad \text{(correct login)}, \\ 0, & g_i \neq c_i \quad \text{(honeytoken hit)}, \end{cases} \qquad Y_i = \begin{cases} 1, & g_i \in W_i \setminus H_i \quad \text{(trap)}, \\ 0, & \text{otherwise.} \end{cases}$$

It updates the global counters $HT \mathrel{+}= X_i, \quad TR + Y_i$. In case $X_i = 0$ a damping factor,namely α, is updated lowering the threshold. The rational behind α is established during the evaluation of the system in Chapter 5.1. After each evaluation, it raises a breach alarm if and only if

$$\frac{HT}{TR} > p \cdot \alpha.$$

It returns a single bit $\mathsf{Win} = 1$ if an alarm was raised, else $\mathsf{Win} = 0$.

Challenger's Success: The Challenger always remains undetected since they always make sure that the System returns by "sacrificing" accounts $\mathsf{Win} = 0$. However, the Challenger considers the campaign successful only if this undetected run also produces a *non-trivial payoff*—namely, a sufficiently large number of breached accounts. If too few accounts are compromised (because many must be "sacrificed" as traps to stay below the threshold), the attacker regards the effort as a loss despite remaining unnoticed.

4.5.4 Enhancing Offline Avdversaries with Intersection Capabilities

Often users reuse the same password on different sites [8, 24, 32]. This enables offline adversaries to use the information gained from different breaches to narrow down their search. *All* honeyword schemas suffer from intersection attacks however HONEYTRAPS enhances the probability of matching honeywords to appear on different servers where a user reuses the same password; besides the true CT.

Intersection attacks can be used to enhance both hidden and baseline offline attackers. In both cases the n-shot game remains the same, however by intersecting the two or more sweetlists of each user were they reuse the same password they have less option to choose from allwoing them to make a more educated guess. Specifically we modify the challenger's guess phase to include the following in both hidden and basline games to reflect this additional information.

Challenger's Guess Phase: Suppose the adversary has recovered $k \geq 2$ independent databases that each contain the same n user accounts. For every user i the Challenger first computes the *intersection set*

$$I_i = \bigcap_{s=1}^{k} H_i^{(s)}, \qquad 1 \leq |I_i| \leq |H_i^{(s)}|,$$

which is guaranteed to contain the genuine token c_i. Having no additional distinguisher, the Challenger selects one token

$$g_i \xleftarrow{\$} I_i$$

and submits it, receiving the System's immediate feedback $X_i \in \{0,1\}$ before proceeding to the next account.

Chapter 5

Evaluation

5.1 Security Analysis

To analyse HONEYTRAPS under conditions that resemble production use, we constructed a comprehensive testbed in which the system is attacked by the different types of adversaries mentioned in Chapter 4.5, each instantiated multiple times based on a predefined probability. This setup allows us to assess whether HONEYTRAPS can reliably distinguish actual database breaches from false-positive events under conditions that reflect real-world usage. Furthermore, we introduced typo events where a user mistakenly inputs a wrong password to test whether benign anomalies could inadvertently trigger false breach alarms.

In the testbed, the system progresses in *epochs*; one epoch corresponds to a single day, so 1 095 epochs model a continuous three-year deployment. During each epoch we draw *one* security event at random from the following pool, all of which were implemented as descirbed in Chapter 4.5:

1. **Baseline-offline attacker** — already in possession of the target site's credential database, the adversary tries to single out the genuine CT within each sweetlist (§4.5.2).

2. FP^{+} **attacker** — knowing the users' passwords, the adversary submits neighbour tokens in order to force a false breach alarm (§4.5.1).

3. **Hidden-offline attacker** — also holding the leaked database, but now blending true-token guesses with deliberate trap activations to stay below the global threshold while compromising accounts (§4.5.3).

4. **Intersection-enhanced offline attacker** — possesses a second HONEYTRAPS database from another service and, assuming worst-case password reuse, intersects the two

Table 5.1: Accounts compromised after a three-year test-bed (1,095 epochs). All scenarios yielded **zero** false breach alarms.

Adversary	**Scenario**	**Compromised (%)**
Adv. 1	Baseline offline	0.0067
Adv. 1*	Offline + intersection (2 sweetlists)	0.18
Adv. 2	Hidden offline	0.095
Adv. 2*	Hidden offline + intersection (2 sweetlists)	2.109

sweetlists to shrink the candidate set before launching either the baseline or hidden-offline attack on the target site.

By replaying $1,095$ such epochs we observe how HONEYTRAPS copes with a statistically representative mix of real breaches, false-positive attacks, and password-reuse exploits over a multi-year horizon.

During the initial epochs of the simulation we observed a pronounced *cold-start* phenomenon. With only a handful of trap and honeytoken activations recorded, the empirical ratio in Eq. (4.3) was highly volatile and—despite its analytically low probability—occasionally exceeded the alarm threshold, producing false positives.

This behaviour is purely statistical. The global test in Eq. (4.3) assumes that the underlying trap and honeytoken frequencies are estimated from a sufficiently large sample. When the observation set is still small, sampling noise dominates, allowing the measured ratio to deviate sharply from its expected value. As the deployment accumulates more events, these frequencies converge, the ratio stabilises, and the false-alarm rate falls back to the theoretical bound.

To address this issue, we introduced a damping factor, α, to dynamically adjust the breach alarm threshold ($p \cdot \alpha$, Eq. 4.3). Each time a honeytoken is triggered, the value of α is updated, lowering the threshold for raising a breach alarm. Initially, α is set such that the threshold is close to 1, but it decreases rapidly after a few honeytoken activations. This adjustment helps reduce the likelihood of false breach alarms during the early stages of system operation. The damping factor is updated according to:

$$\alpha \leftarrow \max\left(d \cdot \alpha, \alpha_{\min}\right), \tag{5.1}$$

where d is a decay factor that controls the rate at which α is reduced. The lower bound $\alpha_{\min}$ is defined to ensure that $p \cdot \alpha$ does not fall below $p + C$, where C is a small constant used to tolerate minor deviations due to noise (e.g., a typo inadvertently triggering a honeytoken).

Table 5.1 shows that HONEYTRAPS remained effective under every attack we simulated. The baseline-offline adversary (Adv. 1) was detected after compromising only

0.0067 % of accounts. When we enhanced the same attacker with an intersection of two leaked sweetlists per user (Adv. 1*), the breach rate rose to just 0.18 %; by comparison, a conventional honeyword scheme that lacks traps and a global threshold would leave *all* reused-password accounts exposed. Throughout the three-year run the FP^+ adversary failed to trigger even a single false breach alarm, confirming that HONEYTRAPS's threshold mechanism does not penalise benign traffic. The hidden-offline attacker (Adv. 2) managed to log in to only 0.095 % of targets while sacrificing the remaining 99.905 % of its accounts as deliberate trap hits to stay below the alarm threshold. Even when this attacker combined its trap tactic with an intersection strategy across two sweetlists—our worst-case assumption that all users reused the same password—the breach rate increased only to 2.109 %, well short of a catastrophic compromise.

Since HONEYTRAPS produced *zero* false positives in every scenario, administrators can treat any breach alarm as a genuine incident—an essential property, given the high operational cost of incident response and the tendency to ignore alerts when they occur too often [18, 35]. Among the parameter settings mandated by Chapter 3.3, the combination $p = 0.10$, decay factor $d = 0.99$, and minimum damping constant $C = 0.02$ yielded the best trade-off: no spurious alarms and timely detection of every simulated credential-database breach.

5.1.1 Relaxing the Per-User Flase-Positive Bound

Chapter 4.1.1 adopted a conservative per-user bound $p = 0.1$ for the FP^+ adversary, in line with the requirement of Eq. (3.4). Under HONEYTRAPS, however, a honeytoken hit does not immediately raise an alarm; an alert is generated only when the *system-wide* honeytoken-to-trap ratio in Eq. (4.3) exceeds the threshold. Because this ratio is evaluated globally, the constraint on any *single* user can be relaxed without loosening overall security.

The threshold in Eq. (4.3), derived from empirical FP^+ behaviour (Fig. 4.3) and validated in our test-bed (Section 5.1), produced *zero* false breach alarms throughout the three-year simulation. This observation suggests that HONEYTRAPS can safely tolerate a higher per-user false-positive probability under the same system-level monitoring.

Raising p enlarges the expected intersection of credential tokens across servers where users reuse the same password. Figure 5.1 plots the probability of observing at least n common CTs ($1 \leq n \leq 120$) across $x \in \{2,3,4,5\}$ breached servers with $p = 0.3$. While, Figure 5.2 plots the probability of observing at least n common CTs ($1 \leq n \leq 120$) across $x \in \{2,3,4,5\}$ breached servers with $p = 0.1$. For two servers the chance of finding at least 5, 10, and 20 common CTs is 0.98, 0.93, and 0.71, respectively—improvements of 0.12, 0.51, and 0.71 over the same experiment with $p = 0.1$.

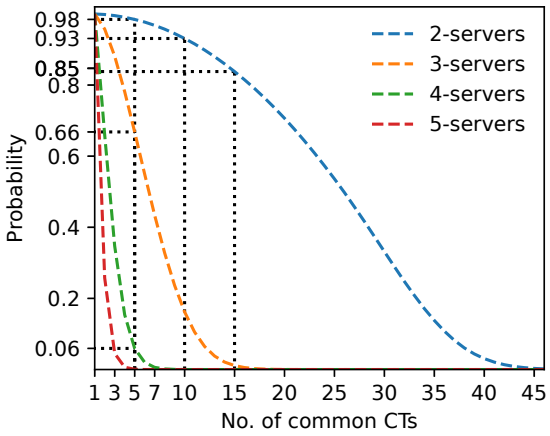


Figure 5.1: Intersection for $p = 0.3$

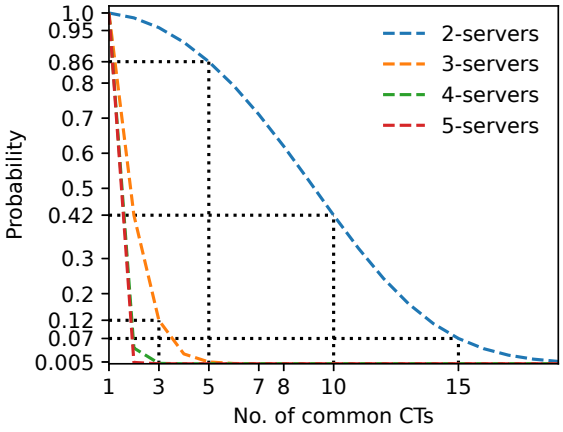


Figure 5.2: Intersection for $p = 0.1$

Figure 5.3: Probability of having at least n common CTs ($1 \leq n \leq 120$) in the intersection of sweet-CT lists from x different servers ($2 \leq x \leq 5$), for $p = 0.3$ and $p = 0.1$

However, a higher p gives an offline attacker more room to mimic an FP^+ by triggering traps, yet the economic trade-off is still crippling. With $p = 0.3$ an attacker must forfeit roughly 70 % of accounts on every round to keep the ratio below the threshold, retaining only 30 % after the first round and 9 % after the second. On a platform with ten million users, the first honeytoken wave forces password resets for seven million accounts—hardly an undetected breach.

Testbed results at $p = 0.3$. Re-running the full three-year test-bed with the relaxed parameter $p = 0.3$ (and $d = 0.9999$, $C = 0.13$) showed that HONEYTRAPS remains resilient. As Table 5.2 shows, the baseline offline adversary (Adv. 1) compromised only 0.0139% of accounts before the system reacted, while augmenting that attacker with a two-sweet-list intersection (Adv. 1*) raised the breach rate to just 0.044%. The hidden-offline adversary (Adv. 2) succeeded on 0.77% of accounts, and even when it combined its trap-masking tactic with intersection (Adv. 2*) the breach rate rose to only 3.82%. Crucially, every scenario again produced *zero* false breach alarms, demonstrating that the higher p markedly improves cross-server intersection without sacrificing the reliability of HONEYTRAPS's global alarm mechanism.

Thus increasing p to 0.3 substantially improves intersection- based detection of password reuse and credential-stuffing attacks while maintaining a zero false-positive rate and limiting attacker pay-off.

Table 5.2: Accounts compromised after a three-year test-bed (1,095 epochs). All scenarios yielded **zero** false breach alarms.

Adversary	**Scenario**	**Compromised (%)**
Adv. 1	Baseline offline	0.0139
Adv. 1*	Offline + intersection (2 sweetlists)	0.044
Adv. 2	Hidden offline	0.77
Adv. 2*	Hidden offline + intersection (2 sweetlists)	3.818

5.2 Storage Requirements

In HONEYTRAPS, we need to store 120 integers (CTs), each requiring 5 bytes since they are in the range $[0, 11.3 \times 10^9)$, resulting in a total of 600 bytes per user. In general, storage costs increase linearly with the number of users and are minimal compared to the data users host on services like Facebook and Instagram.

Bernoulli Honeywords require a 128-bit vector (BF), which equates to 16 bytes per user. Even for 10 million users, Bernoulli Honeywords and HONEYTRAPS require only 160MB and 6GB, respectively, which is negligibly low given today's storage costs. In fact, even the classic honeywords schema (20 digests/user) requires 12.8GB for the same number of users.

However, HONEYTRAPS's storage footprint can be significantly reduced using Run-Length Encoding (RLE). Specifically, we represent the selected honeytokens for each user as a bit-stream of size 11.3×10^9. Honeytokens are marked with 1, while the rest are marked with 0. Given that each user has 120 honeytokens $\approx 99.99999\%$ of the bits marked with 0, the majority of which are successive. RLE is chosen due to its efficiency with datasets containing long sequences of repeated values, as it compresses these sequences into a value and count. The use of RLE enables a logarithmic reduction in the size of a bitstream, with the compression efficiency increasing as the length of the bitstream grows. To illustrate, consider a bitstream composed entirely of zeros, with a total length equal to the size of the token space, (i.e 11.3×10^9). Applying RLE to this sequence reduces the representation to a single value corresponding to the length of the run. The number of bits required to encode this value is determined by $\lceil \log_2(11.3 \times 10^9) \rceil$, thus 34 bits are required to store the length of the sequence. Now, consider a case where the sequence of zeros is interrupted by a single 1 at the midpoint, effectively splitting the sequence into two halves. Following the same method it is determined that each half requires 33 bits (66 bits in total). If the sequence is interrupted at any other point the required bits needed to store the length of both halves will be less. This example highlights how introducing breaks midpoint in the sequence reduces compression efficiency because

of the logarithmic nature of RLE. Let us now estimate the compression efficiency of RLE in the worst-case scenario. First, we consider the size of the window containing all honeytokens (and traps), which is 1189. In the worst case, this window is placed at the midpoint of the sequence of zeros. This configuration divides the sequence into three distinct segments: X, Y, and Z. Specifically X and Z represent the sequences of zeros to the left and right of the honeytoken window, respectively. Y represents the honeytoken window itself. The bits required to represent of both X and Z are $\lceil \log_2((11.3 \times 10^9 - 1189)/2) \rceil$ = 33 each. To estimate Y we again consider the worst case scenario were all 120 ones are spread evenly across the 1189 possible positions. Hence, $\lceil \log_2((1189 - 120)/120) \rceil$ = 4 are needed for each sequence of zeros. The positions of the ones will be inferred from the run length of zeros and thus will not be exclusively stored. In total, approximatley 542 bits (33 + 33 + 4 x 119) are requried for each user which equates to $\approx$ 68 bytes per user. Thus, for 10 million users 680MB are needed.

5.3 Efficiency

The computational cost of HONEYTRAPS is modest. Both registration and authentication require only a *single* evaluation of SHA3-256 and one modulo operation per password, whereas Bernoulli Honeywords invoke twenty independent hash functions for each event; HONEYTRAPS therefore incurs roughly 1/20th of the hashing overhead.

Compression with run-length encoding adds no runtime penalty. Because each run encodes only its length, the index of every credential token (and every trap position) can be reconstructed on-the-fly—no decompression step is needed during login or registration.

5.4 Deployment Effort

A major benefit of HONEYTRAPS is its drop-in compatibility with existing authentication infrastructures. Because honeytokens are generated *independently* of the user's plaintext password, the scheme never needs the original secret during enrolment or migration. Operators can therefore retrofit HONEYTRAPS without asking users to re-enter credentials.

Migrating an existing password file is uncomplicated. Production systems already keep password digests produced by a cryptographically strong KDF—typically PBKDF2, bcrypt, SHA-2, or SHA-3. For each stored digest the operator simply reduces the digest *as an integer* modulo w to derive the account's genuine credential token c. Once c is known, the normal HONEYTRAPS procedure— randomly positioning honeytokens and traps around c (Chapter 3.4)—is applied.

No user re-enrolment is necessary. During authentication the server continues to hash

the submitted password with the *same* legacy KDF already in place; it then performs the same modulo-w reduction to check whether the resulting token matches the account's stored c or one of its honeytokens. Thus the only code change is the extra modulo operation, leaving the original hash/KDF pipeline—and therefore the site's security assumptions—intact.

This one-pass transformation makes HONEYTRAPS deployable even in production systems with millions of legacy accounts, avoiding the massive user re-enrolment that password-dependent honeyword generators require [16, 20, 23, 36].

Chapter 6

Discussion

6.1 Limitations & Future Work

Personally Identifiable Information (PII). Password guessing, where no honeywords are involved, can be augmented using leaked personally identifiable information (PII) such as birthdates, phone numbers, or addresses. These PII-enriched passwords can be hashed and compared against stored digests, leveraging the deterministic nature of cryptographic hash functions as one-to-one oracles. This issue is similarly present in *Bernoulli Honeywords*, where the probability of a collision (false positive) is negligible and even if a collision occurs, the result is unlikely to resemble a plausible password.

In contrast, HONEYTRAPS behaves as a many-to-one oracle, permitting multiple human-chosen inputs to map to the same credential tokens (CTs). This property makes it harder for an adversary to distinguish the correct input using PII. In classic honeyword systems [20], the attacker can evaluate each potential password against a fixed set of sweetwords, allowing PII to help distinguish the real password from decoys. However, in HONEYTRAPS, collisions in the CT space result in a vastly expanded honeyword space. This reduces the utility of PII in targeted guessing attacks, as many distinct strings may map to the same CT. Furthermore, while HONEYTRAPS does not guarantee that collisions will map to strings containing a user's personally identifiable information (PII), the scheme can be extended to better mitigate PII-enhanced attacks. One possible approach is to generate multiple clusters of honeytokens, each incorporating at least one slightly modified version of the real password enriched with PII. This diversification would make it more difficult for adversaries to identify the true credential based solely on PII cues. In contrast, *Bernoulli Honeywords* is not easily extendable in this manner. Due to the nature of the Bloom filter (BF) employed, explicitly storing additional PII-derived strings would rapidly saturate the filter, as discussed in Chapter 3.1.2, thereby suffering from false-alarms.

Strong Passwords. In traditional single-password authentication, an account's susceptibility to online guessing attacks is directly tied to the strength of the user's chosen password. In contrast, HONEYTRAPS equalizes this vulnerability: all accounts, regardless of password complexity, are equally susceptible to online guessing. Specifically, an attacker only needs to guess a string that maps to the same credential token (CT) as the user's strong password to breach the account.

Modern honeyword schemes, including *Bernoulli Honeywords*, operate under the assumption that user passwords are weak and thus easily guessable [20]. As a result, all passwords—strong or weak—are incorporated into the schema. To protect weak passwords effectively, some strong passwords may be inadvertently rendered vulnerable, reducing their security in practice. This generalization is an inherent limitation across current honeyword-based designs. On the other hand, slightly weakening password strength may be a worthwhile trade-off if it enables faster detection of database breaches. An interesting research direction is to explore honeyword schemes that apply only to weak passwords. This is challenging since signalling which passwords are weak or strong during a data breach is also problematic.

Chapter 7

Related Work

Intersection attacks

Guo et al. [15] introduced *Superword*, a honeyword-based mechanism designed to mitigate intersection attacks. Rather than storing sweetword hashes on the main server, Superword stores an index that references a corresponding record maintained by a separate honeychecker, which holds the actual password hash. However, this design introduces a critical weakness: because the honeychecker stores both real and decoy credentials, compromising it would render Superword insecure. By contrast, HONEYTRAPS integrates directly with systems like Amnesia and Lethe, avoiding any dependence on a permanently trusted external component.

Password Reuse Mitigation.

Addressing password reuse is a fundamentally different challenge from the focus of this work. Here, we aim to defend against intersection (and credential stuffing attacks) under the realistic assumption that users commonly reuse passwords across different websites—we do not attempt to prevent or interfere with that behavior.

Wang et al. [33] proposed a system in which websites collaboratively discourage password reuse. In their approach, when a user selects a password on one site (the requester), that site can query other participating sites (the responders) to check whether the same or a similar password has already been registered elsewhere. If so, the requester may reject the password and prompt the user to choose a different one. While effective in theory, this model raises serious usability concerns, as it disrupts the very convenience that motivates password reuse in the first place.

Chapter 8

Conclusion

HoneyTraps offers a practical, deployable framework that strengthens the detection of credential breaches while addressing the limitations of traditional honeyword systems. By probabilistically distributing decoys and incorporating trap tokens, HoneyTraps resists intersection (and credential stuffing attacks), even when adversaries possess knowledge of HONEYTRAPS's functionality. Its compatibility with legacy systems, low storage overhead, and strong false-positive resilience make it a robust defense for modern authentication infrastructures.

Bibliography

[1] M. Aldwairi and L. Tawalbeh. Security techniques for intelligent spam sensing and anomaly detection in online social platforms. In *IJECE*, volume 10, page 275, 2020.

[2] Z. Alom, B. Carminati, and E. Ferrari. Detecting spam accounts on twitter. In *ASONAM*, pages 1191–1198, 2018.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[4] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567, 2012.

[5] W. J. Burns. Rockyou password leak, Jan 2019. https://www.kaggle.com/wjburns/common-password-list-rockyoutxt.

[6] J. Canny. Secure Hash Algorithms. `https://people.eecs.berkeley.edu/~jfc/cs174/lecs/lec22/lec22.pdf`. [Accessed 05-07-2024].

[7] P. Cao and Z. Wang. Bloom filters - the math. `https://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html`. Accessed: 2024-06-25.

[8] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. In *NDSS*, pages 23–26, 2014.

[9] A. Dionysiou and E. Athanasopoulos. Lethe: Practical data breach detection with zero persistent secret state. In *EuroS&P*, pages 223–235, 2022.

[10] A. Dionysiou, V. Vassiliades, and E. Athanasopoulos. Honeygen: generating honeywords using representation learning. In *AsiaCCS*, pages 265–279, 2021.

[11] C. T. Do, N. H. Tran, C. Hong, C. A. Kamhoua, K. A. Kwiat, E. Blasch, S. Ren, N. Pissinou, and S. S. Iyengar. Game theory for cyber security and privacy. *ACM Comput. Surv.*, 50(2), May 2017.

[12] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 657–666, New York, NY, USA, 2007. Association for Computing Machinery.

[13] D. Florêncio, C. Herley, and P. C. Van Oorschot. An administrator's guide to internet password research. In *LISA*, pages 44–61, 2014.

[14] S. Gheewala and R. Patel. Machine learning based twitter spam account detection: a review. In *ICCMC*, pages 79–84, 2018.

[15] Y. Guo, Z. Zhang, and Y. Guo. Superword: A honeyword system for achieving higher security goals. *Computers & Security*, 103:101689, 2021.

[16] X. He, H. Cheng, J. Xie, P. Wang, and K. Liang. Passtrans: An improved password reuse model based on transformer. In *ICASSP*, pages 3044–3048. IEEE, 2022.

[17] Z. Huang, L. Bauer, and M. K. Reiter. The impact of exposed passwords on honeyword efficacy. In *USENIX Security*, page To appear., 2024.

[18] IBM. Cost of a data breach report 2023. https://www.ibm.com/reports/data-breach.

[19] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification, 2016.

[20] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In *CCS*, pages 145–160, 2013.

[21] M. Mavronicolas, V. Papadopoulou, A. Philippou, and P. Spirakis. A network game with attackers and a defender. *Algorithmica*, 51:315–341, 2008.

[22] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[23] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *SP*, pages 417–434, 2019.

[24] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *CCS*, pages 295–310, 2017.

[25] I. Security. Cost of a data breach report 2022. `https://www.ibm.com/security/data-breach`, 2022.

[26] S. Security. 2018 credential spill report. https://info.shapesecurity.com/rs/935-ZAM-778/images/Shape_Credential_Spill_Report_2018.pdf, 2018.

[27] S. Shead. Elon Musk says Twitter deal 'cannot move forward' until he has clarity on fake account numbers — cnbc.com. https://www.cnbc.com/2022/05/17/elon-musk-says-twitter-deal-cannot-move-forward-until-he-has-clarity-on-bot-html. [Accessed 01-01-2025].

[28] S. Shiva, S. Roy, and D. Dasgupta. Game theory for cyber security. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, New York, NY, USA, 2010. Association for Computing Machinery.

[29] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware? Understanding the risks of stolen credentials. In *CCS*, pages 1421–1434, 2017.

[30] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1421–1434, New York, NY, USA, 2017. Association for Computing Machinery.

[31] A. Vigderman. Password manager annual report 2022, Jan. 2023. Available at https://www.security.org/digital-safety/password-manager-annual-report/.

[32] C. Wang, S. T. Jan, H. Hu, D. Bossart, and G. Wang. The next domino to fall: Empirical analysis of user passwords across online services. In *CODASPY*, pages 196–203, 2018.

[33] K. C. Wang and M. K. Reiter. How to end password reuse on the web. In *NDSS*, 2019.

[34] K. C. Wang and M. K. Reiter. Using amnesia to detect credential database breaches. In *USENIX Security*, pages 839–855, 2021.

[35] K. C. Wang and M. K. Reiter. Bernoulli honeywords. In *NDSS*, 2024.

[36] F. Yu and M. V. Martin. Honey, i chunked the passwords: Generating semantic honeywords resistant to targeted attacks using pre-trained language models. In *DIMVA*, page 89–108, 2023.

[37] D. Yuan, Y. Miao, N. Z. Gong, Z. Yang, Q. Li, D. Song, Q. Wang, and X. Liang. Detecting fake accounts in online social networks at the time of registrations. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1423–1438, 2019.