Published in International Journal of Information Security; Volume 8, Number 4; August 2009
The original publication is available at www.springerlink.com; DOI 10.1007/s10207-009-0084-3

Richard Shay · **Elisa Bertino**

A Comprehensive Simulation Tool for the Analysis of Password Policies

Abstract Modern organizations rely on passwords for preventing illicit access to valuable data and resources. A well designed password policy helps users create and manage more effective passwords. This paper offers a novel model and tool for understanding, creating, and testing password policies. We present a password policy simulation model which incorporates such factors as simulated users, accounts, and services. This model and its implementation enable administrators responsible for creating and managing password policies to test them before giving them to actual users. It also allows researchers to test how different password policy factors impact security, without the time and expense of actual human studies. We begin by presenting our password policy simulation model. We next discuss prior work and validate the model by showing how it is consistent with previous research conducted on human users. We then present and discuss experimental results derived using the model.

Keywords Password · Policy · Simulation · Management

1 Introduction

Password-based authentication mechanisms are important for information system security [2]. User authentication is frequently performed by asking a user for a name and password combination [12]. Passwords are often the sole barrier between a potential intruder and its target [3,11]. Organizations are especially dependent on passwords security. If the password of a single user becomes known to a malicious party, an entire system can end up being compromised [1]. In several cases, organizations have faced damage to their reputation and finances from information theft [6].

Richard Shay
Norwood, MA 02062
Tel.: (781) 769-4233
E-mail: rich@richshay.com

Elisa Bertino
Purdue University
Department of Computer Sciences
305 N. University Street
West Lafayette, IN 47907-2107
E-mail: bertino@cs.purdue.edu

Because passwords are so important, password policies are critical. Unless their policy says otherwise, users tend to create very simple passwords [1,5,4]. These simple passwords are especially vulnerable to attackers [13,12]. Password policies may force users to create more effective passwords. Password policies are regulations governing each step of the password lifecycle, from how they are created to when they are deleted [10]. A sound password policy can increase the overall security of an organization [6].

Although having an effective password policy is clearly important, gathering empirical evidence about what constitutes such a policy remains difficult. At present, studying the effectiveness of the facets of a password policy is often accomplished by conducting surveys and studies on actual human users. Examples of such studies are found in Section 4. However, performing experiments on human users is both time-consuming and costly, and may raise privacy concerns.

The primary contribution of this paper is a password policy simulation model (model). This model is a significant enhancement and extension of the model presented in our prior work [10]. The model considers technical factors and human factors essential to the creation of a successful password policy [8]. The model highlights the interaction between password policy and financial health, underscoring the central role policy plays in safeguarding the assets of an organization [6]. The model has been implemented in a simulation tool called the Password Policy Simulation Tool (PPST).

Our model in general and PPST in particular are powerful, novel resources for administrators and researchers. Our work is novel because it enables more precise and thorough study of password policies, without the complexity of using live human test subjects. It facilitates research on password policies and the creation of actual password policies.

The paper is organized as follows. Section 2 presents the high-level concepts underlying our password policy simulation model, and the data used in the model. Section 3 describes the algorithms employed by our model. We discuss previous research and validate the model in Section 4. Experiments conducted using PPST and their results are presented and discussed in Section 5.

2 Model Concepts

feq This section and the next describe the password policy simulation model. In this section, we cover the high-level concepts underlying the model and its variables. In Section 3 we present how those variables are used in the model's algorithms.

2.1 Users, Accounts, and Services

The model computer system is divided into three primary types of components: `Users`, `Accounts`, and `Services`. The model requires at least one of each of these components. A `User` represents a single person legitimately using the computer system. An `Account` represents one account with which a user logs into the system. A `Service` represents a resource utilized by a `User` through an `Account`, to perform work and generate income. For example, Plato is a `User`; Plato's username represents an `Account`; and his email represents a `Service`. Plato logs into his `Account` to access his email `Service`.

Each `User` must have at least one `Account` and may have more than one. Each `Account` belongs to exactly one `User`; `Users` may not share `Accounts`. Each `Account` has one or more `Services`. Different `Accounts` may share the same `Services`. A `Service` is not assigned directly to a `User`; they are connected only through an `Account`. Figure 4 illustrates this graphically.

No new `Users`, `Accounts`, or `Services` are created while the `Simulation` is running. Which `Users` have which `Accounts`, and which `Accounts` access which `Services`, remain static.

2.2 Accounts and Passwords

Each `Account` has exactly one password at any time. An `Account` may be configured such that its password expires after a set period of time. If its password expires, an `Account` creates a new password. A `User` may also request a new password as described in Subsection 2.4. Each `Account` is given a new password on the first day in the model. All passwords for a given `Account` have the same length and per-character entropy. Each new password is assumed to be entirely new.

2.3 Being Compromised

At any point in time, each `Account` and each `Service` is either `Compromised` or not `Compromised`. All `Accounts` and `Services` start as not `Compromised`. An `Account` is considered to be `Compromised` if and only if a malicious entity knows its password. A `Service` is considered to be `Compromised` if and only if there is a `Compromised` `Account` which accesses it. In this model, a `User` is never considered `Compromised`. A `Compromised` `Account` is no longer `Compromised` when it receives a new password. A `Compromised` `Service` is no longer `Compromised` when none of the `Accounts` accessing it are `Compromised`.

How `Accounts` and `Services` are `Compromised` is illustrated in Figure 1.

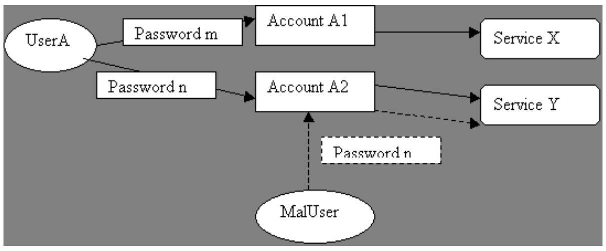

Fig. 1 A malicious user has obtained the password used in `Account` A2. This causes `Account` A2 to become `Compromised`, which in turn causes `Service` Y to become `Compromised`. This is indicated by the dashed lines.

2.4 Help Desk

The help desk is a resource which `Users` may call to create a new password for one of their `Accounts`. A `User` may call the help desk when the `User`, rightly or wrongly, suspects that an `Account` has become `Compromised`. Calling the help desk assigns a new password to the `Account` and makes it not compromised.

Note that the help desk is not called to create a new password when a password expires. It is only used to create new passwords at irregular intervals. In addition, there may be a cost associated with calling the help desk, as covered in Subsection 2.8.

2.5 The Memorization Cycle

The password lifecycle is shown in Figure 2. Each password belongs to an `Account`, which in turn belongs to a `User`. A password is at any time either written down or not written down. When a new password is created, its `User` attempts to memorize it. A password not successfully memorized is written down. Each day, the `User` attempts to memorize the password and thereby have it be no longer written. Subsection 3.1 explains how the probability of a successful memorization is determined. When a new password is created for an `Account`, its old password is discarded and the process of memorization begins again, with its `User` trying to memorize the new password.

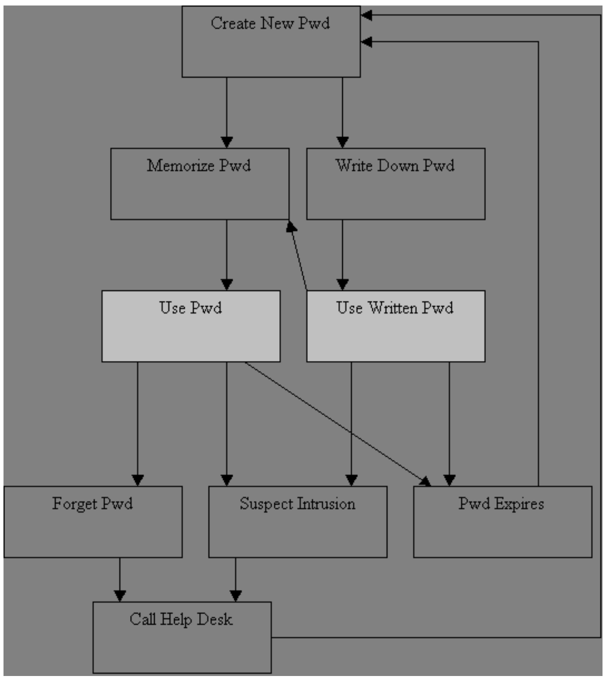

Fig. 2 The lifecycle of a password. A password may end a day only in one of the two highlighted states.

2.6 Password Inundation

As shown in Subsection 4.3, research has found that a `User` can become less successful at memorizing passwords if that `User` is forced to memorize too many passwords too frequently. In our model, this phenomenon is called password inundation. Password inundation in the model is based on the average number of new passwords a user receives per day. For example, if Plato has two `Accounts` which both must have their passwords changed every 30 days, then Plato has $\frac{1}{15}$ new passwords per day, on average. The calculation of average passwords per day includes only regularly scheduled password changes, not those resulting from a call to the help desk. The password assigned to each `Account` on the first day of the simulation is excluded from this number, as are those `Accounts` whose passwords do not expire.

2.7 Attacks and Vulnerabilities

There are two ways in which an `Account` may become `Compromised`. First, each `Account` is daily subject to a specified number of brute force attacks. The longer and more complex the password of the `Account`, the less likely it is to become `Compromised` in such an attack. Second, if the password of an `Account` is written down, it is daily subject to an additional probability of becoming `Compromised`.

2.8 Model Data

The input to the model consists of four components: the `Simulation`-wide variables; and one list each of `Users`, `Accounts`, and `Services`. The input variables for the entire simulation are shown in Table 1. Input variables defined for each `User` are listed in Table 2. Input variables for each `Account` are in Table 3. Table 4 contains the input variables for each `Service`.

As with input, output is divided into four parts. There are `Simulation`-wide output values. There are also specific output values for each `User`, `Account`, and `Service`. Output variables for the entire simulation are found in Table 5. Output variables for each user are found in Table 6. The output for each `Account` is listed in Table 7. The output variables for each `Service` are in Table 8.

2.9 Input Data Notes

`maxDays`sets the duration of the simulation; the simulation is run for the specified number of whole days. The simulation is given a list of the `Users`, `Accounts`, and `Services` to be in the simulation. Each `User` includes a list of `Accounts`; these are the `Accounts` associated with that `User`. Each `Account` includes a list of `Services`; these are the services associated with that `Account`.

The different `User` variables configure factors such as whether the `User` becomes more familiar with a given password over time and how likely the `User` is to believe falsely that he or she is compromised. This is also where the daily cost of the `User` is specified. Note that `User.memory` indicates how likely the `User` is to memorize a password of seven digits, and not necessarily an actual password which the `User` might have. This enables the memories of different `Users` to be entered in a uniform, comparable manner. The actual probability of the `User` memorizing a given password is calculated as a function of several factors, as is shown in detail in Subsection 3.1.

`User.incomeMultiplier` is a number which is multiplied by whatever income the `User` would generate. If this value is set to 1 for each `User` in an organization, then how much a `User` generates becomes a function of which `Services` that `User` accesses. However, to indicate that a given user is able to generate greater value from the same service, this may be set higher for that user. Likewise, for a malicious user who generates nothing whatsoever, this may be set to 0 to prevent any income from being generated by the user.

`Account.numAttacks` and `Account.compWrit` enable the administrator using the simulation to specify which sorts of threats present themselves in an organization. If an organization is under constant external attack on its passwords, but there is little danger of someone physically present maliciously using found information, then `Account.numAttacks` would be higher and `Account.compWrit` would be lower. The input variables of

Variable	Type	Description
`Name`	String	Name of the `Simulation`
`maxDays`	long	Number of days for the `Simulation` to run
`users`	List<`User`>	All `Users` in the `Simulation`
`accounts`	List<`Account`>	All `Accounts` in the `Simulation`
`services`	List<`Service`>	All `Services` in the `Simulation`

Table 1 Input variables for the whole model

Variable	Type	Description
`Name`	String	Name of the `User`
`dailyCost`	double	Daily cost of the `User`
`memory`	double	Base probability that the `User` remembers a seven-digit password after seeing it for the first time
`falsSuspect`	double	For each `Account` per day, if that `Account` is not `Compromised`, this is the probability that the `User` believes it is
`trueSuspect`	double	For each `Account` per day, if that `Account` is `Compromised`, this is the probability that the `User` believes it is
`incomeMultiplier`	double	All income of `User` is multiplied by this
`inundationMax`	double	Maximum probability that `User` fails to memorize a password due to password inundation
`inundationConst`	double	Multiplicative constant for password inundation; the higher the constant, the more the `User` is affected by inundation
`learningCurve`	double	Determines how well `User` learns a new password in time; 0 indicates the `User` doesn't become more familiar in time
`accounts`	List<`Account`>	List of the `Accounts` which belong to this `User`

Table 2 Input variables for each `User`

Variable	Type	Description
`Name`	String	Name of the `Account`
`numAttacks`	long	Daily brute-force attacks against this `Account`
`compWrit`	double	Daily probability that the `Account` is `Compromised`, if its password is written down
`passExpire`	long	The password expires after this many days; if this is less than 1, then the password never expires
`passLength`	long	Length of the password
`charEntropy`	double	Per-character entropy of the password
`helpDeskCost`	double	Cost per help desk call for this `Account`
`probForget`	double	Daily probability that memorized password is forgotten, if memory check fails; if zero, a memorized password is not forgotten
`services`	List<`Service`>	Services which are accessed by this `Account`

Table 3 Input variables for each `Account`

`Account` are also where the administrator specifies password requirement information, such as the length and complexity of passwords, and how often those passwords expire.

Note that the input variable `Account.ProbForget` is the probability that a `User` with a memorized password forgets that password, if and only if that `User` has already failed a daily memory check. In other words, each `User` with a memorized password must make a memory check each day against his or her password. If that `User` fails the check, then the `User` forgets his or her memorized password with probability equal to `Account.ProbForget`. If `Account.ProbForget` is zero, a memorized password is never forgotten.

Note that the input variables for a `Service` include both `Service.compDailyCost` and `Service.compIncomeMult`. These are both variables which indicate the effect of being `Compromised` on a `Service`, but they do so in different ways. `Service.compDailyCost` is an additional daily cost of a `Service` being compromised, while `Service.compIncomeMult` is a value by which all income of the `Service` is multiplied when `Compromised`. If being `Compromised` causes a given service to cease generating value, but does not entail any other expenses, then `Service.compDailyCost` may be low while `Service.compIncomeMult` may be a small fraction. Whereas, if a `Service` being compromised leads to costly public relations or compliance expenses for an organization, then `Service.compDailyCost` may be significantly higher.

3 Model Algorithm

This section explains the algorithms in the model. We first describe the formulas and subroutines used in model, and then show how they are combined into the single main algorithm.

Variable	Type	Description
`Name`	String	Name of the `Service`
`standardDailyCost`	double	Standard daily cost of the `Service`
`compDailyCost`	double	Additional daily cost of this `Service` being `Compromised`
`compIncomeMult`	double	When `Compromised`, all income generated is multiplied by this
`standardIncome`	double	Standard daily income of the `Service`

Table 4 Input variables for each `Service`

Variable	Type	Description
`balance`	double	Final balance
`balance_maximum`	double	Maximum possible balance, assuming nothing was `Compromised`
`in`	double	Actual Income
`in_maximum`	double	Maximum possible income, assuming nothing was `Compromised`
`out`	double	Actual cost
`out_minimum`	double	Minimum possible cost, assuming nothing was `Compromised`
`out_users`	double	Cost from `Users`
`out_accounts`	double	Total cost from `Accounts`
`out_accounts _falseSuspicion`	double	Cost from `Accounts` having false suspicion
`out_accounts _trueSuspicion`	double	Cost from `Accounts` having true suspicion
`out_accounts _forgottenPassword`	double	Cost from `Accounts` forgetting memorized passwords
`out_services`	double	Total cost from `Services`
`out_services _standard`	double	Cost from `Services`, standard operating costs
`out_services _compromised`	double	Cost from `Services`, resulting from being `Compromised`

Table 5 Output variables for the whole simulation

Variable	Type	Description
`balance`	double	Final balance of `User`
`totalIncome`	double	Total income from `User`
`totalCost`	double	Total cost of `User`
`num_maxDaily IncomeHash`	HashMap <`Service`, Double>	Maximum daily Income per `Service`
`num_incomeHash`	HashMap <`Service`, Double>	Total actual income per `Service`

Table 6 Output variables for each `User`

3.1 User Memory

Suppose that `User` U has `Account` A. There are multiple occasions when the memory of `User` is tested against the password of A. This occurs, for example, when A is assigned a new password, to determine whether that password is written down. It also occurs to check whether a `User` who has memorized a password later forgets that password. Each such memory check results either in a success or a failure. The derivation of the probability of failure of this memory check is described in detail here.

The factors considered in determining the success of the memory check are the memory ability of U, the complexity of the password, password inundation, and the age of the password. The algorithm uses `User` and `Account` input variables described in Subsection 2.8.

Let us define the following variables, which are either tracked by the `Simulation` or derived from administrator-defined values. None of these values are input directly by the administrator.

- A.**passEntropy** is the total entropy of the password of A
  A.**passEntropy** = A.**passLength** * A.**charEntropy**
- U.Δ is the average daily number of new passwords for U
  U.Δ = $\Sigma_{i=1}^{n}(U.Account_i.passExpire)^{-1}$
  where n is the number of `Accounts` of U with expiring passwords
  Note that U.Δ includes only scheduled password changes
- P_i is the probability that U fails the memory check due to inundation
  P_i = MAX(MIN(U.Δ* U.**inundationConst**, U.**inundationMax**), 0)

Variable	Type	Description
`num_newPassword`	long	Number of new passwords assigned
`num_writePassword`	long	Number of times password written down
`num_ unwritePassword`	long	Number of times a written password is memorized
`num_becomeComp`	long	Number of times the `Account` becomes `Compromised`
`num_helpDeskCall`	long	Number of help desk calls
`num_trueSuspicion`	long	Number of help desk calls based on true suspicion
`num_falseSuspicion`	long	Number of help desk calls based on false suspicion
`num_ forgottenPassword`	long	Number of help desk calls based on forgetting passwords
`num_ bruteforceAttacks`	long	Number of brute force attacks; they only occur if not `Compromised`
`num_ bruteforceSucceed`	long	Number of successful brute force attacks
`num_ writtenAttack Succeed`	long	Number of times password `Compromised` by being written
`num_daysEndComp`	long	Number of days ended `Compromised`
`num_daysEndUncomp`	long	Number of days ended not `Compromised`

Table 7 Output variables for each `Account`

Variable	Type	Description
`totalCost`	double	Total cost of this `Service`
`standardCost`	double	Cost from the standard operation of this `Service`
`compCost`	double	Cost due to being `Compromised`
`daysComp`	long	Number of days finished `Compromised`
`daysUncomp`	long	Number of days finished not `Compromised`

Table 8 Output variables for each `Service`

– age is the number of days for which the password of A has existed

Using the above variables, we can calculate the probability of U failing a memory check. Let the probability of U failing be $\mathscr{P}$. Let the probability of U failing the memory check, without regard to inundation or password age, be $\mathscr{P}_0$. Note that $\mathscr{P}_0$ and $\mathscr{P}$ refer to probability of failure to memorize, while U.**memory** is a probability of succeeding.

We make the following assumption:

$$\frac{P(fail\ memorize\ 7-digit\ number)}{Complexity\ of\ 7-digit\ number} = \frac{\mathscr{P}_0}{Complexity\ of\ password} \quad (1)$$

Therefore:

$$\frac{1-U.\textbf{memory}}{7*log_2 10} = \frac{\mathscr{P}_0}{A.\textbf{passEntropy}} \quad (2)$$

Therefore:

$$\mathscr{P}_0 = \frac{(1-U.\textbf{memory})*A.\textbf{passEntropy}}{7*log_2 10} \quad (3)$$

$\mathscr{P}_0$ and P_i are independent probabilities that U fails. The probability of U failing due to neither of them is:

$$(1-\mathscr{P}_0)*(1-P_i) \quad (4)$$

Therefore, the probability of U failing from at least one of them is:

$$1-(1-\mathscr{P}_0)*(1-P_i) \quad (5)$$

Finally, we allow U to be more likely to succeed in memory checks with increasing familiarity with a password. Therefore, we raise the entire probability of failure to a power which reflects the age of the password, and how quickly U learns:

$$1+U.\textbf{learningCurve}*age \quad (6)$$

This value is always at least one. Note that if U.**learningCurve** is zero, then the memory checks do not become more likely to succeed over time. Note also that since the value of age on the day a password is assigned is zero, U.**learningCurve** does not affect how likely U is to memorize a password on its day of creation.

This gives us:

$$\mathscr{P} = (1-(1-\mathscr{P}_0)*(1-P_i))^{1+U.\textbf{learningCurve}*age} \quad (7)$$

$\mathscr{P}$ is the probability that U fails in a given memory check. In Algorithm 1, A.**memCheck()** returns **false** with a probability of $\mathscr{P}$ and **true** otherwise.

3.2 Brute Force Attacks

The likelihood of a brute force attack succeeding against an `Account` is a function of the complexity of its password. A.**passLength** is the number of characters in the password of A, and A.**charEntropy** is the entropy of each character in the password of A. The total entropy of a string of text is the sum of the entropy of its characters. Therefore, for an account A, the total entropy of its password is the number of characters in its password A.**passLength** multiplied by the per-character entropy A.**charEntropy**:

$$A.\mathbf{passEntropy} = A.\mathbf{passLength} * A.\mathbf{charEntropy} \tag{8}$$

A.**passEntropy** represents the minimum number of bits which uniquely determine the password of A [9]. Put another way, the number of passwords possible under the policy of A, with fixed per-character entropy and length, is $2^{A.\mathbf{passEntropy}}$. Therefore the probability of a single brute force attack succeeding against A is defined as:

$$P(Brute\ Force\ Succeeds) = \frac{1}{2^{A.\mathbf{passEntropy}}} \tag{9}$$

A is subject to A.**numAttacks** brute force attacks each day. In Algorithm 1, the function A.**bruteForceAttacksSucceed()** returns `true` with a probability equal to that of at least one of the daily brute force attacks against A succeeding.

3.3 Cost and Income

3.3.1 Balance

The concept of a balance is key to the entire model. Balance is the net amount of profit made by the organization depicted in the simulation. The higher the balance, therefore, the more profit the organization has earned. Balance is net profit, the income minus cost. Using balance enables viewing the success of a given password policy in monetary terms. It also allows for two different password policies to be compared easily; if one policy leads to a higher balance than the other, the former is more likely to be a better password policy.

$$Balance = Income - Cost \tag{10}$$

In Algorithm 1, the function **calculateIncomeCost()** performs the cost and income calculations, as described below.

3.3.2 Income

All income is generated by a `User` U utilizing a `Service` S. U is considered to utilize S if and only if U has at least one `Account` which accesses S. U having more than one `Account` accessing S does not increase the income U generates utilizing S.

If S is not compromised, the daily income generated by U utilizing S is:

$$U.incomeMultiplier * S.standardIncome \tag{11}$$

If S is compromised, the daily income generated by U utilizing S is:

$$U.incomeMultiplier * S.compIncomeMult * S.standardIncome \tag{12}$$

3.3.3 Cost

The model includes both fixed daily costs and variable costs. Each `User` U has a fixed daily cost, U.**dailyCost**. Each `Account` A incurs a cost, A.**helpDeskCost**, whenever its `User` calls the help desk. Each `Service` S has a fixed daily cost, S.**standardDailyCost**. In addition, each day that S ends compromised has an additional cost S.**compDailyCost**.

3.4 The Daily Algorithm

The atomic unit of time for the model is the day. Each simulation is run for a specified number of whole days. The algorithm is found in Algorithm 1.

4 Assumptions, Related Work, and Validation

This section discusses the assumptions made in the model, prior related work, and the validation of our model. These three topics are included in a single section because they related to one another; the related work shows that our assumptions are reasonable, and this in turn helps to validate the model. We begin by contrasting our current model with that presented in our previous work on password policy simulation. We then discuss assumptions made in our model. Next, we validate the model by considering it and its assumptions in light of results found in studies on human users. Then, we discuss other relevant papers.

4.1 Our Prior Model

Our previous work, "Password Policy Simulation and Analysis," presents a model for simulating password policies [10]. Our new model makes significant improvements to that model.

The most salient improvement to our model is the introduction of a financial balance. In the previous model, damage to the simulated system is measured as a unitless integer called `harm`. The new model measures damage to the organization by tracking cost and income, and how these are both impacted by security lapses. This is more intuitive and meaningful than the previous `harm` value.

In our prior work, the computer system consists only of sets of `Users`. The new model divides the resources of a computer system into three parts – `Users`, `Accounts`, and `Services`. The old model is illustrated in Figure 3. The new model is shown in Figure 4.

Dividing the computer system into distinct users, accounts, and services allows new opportunities for research. A single sign-on system may be simulated by giving each `User` a single `Account` which accesses multiple `Services`. If multiple `Users` are sharing a `Service`, the `Account` of one such user being `Compromised` can impair the productivity of all `Users` wishing to access that `Service`. In this way, `Users` are more interdependent than before.

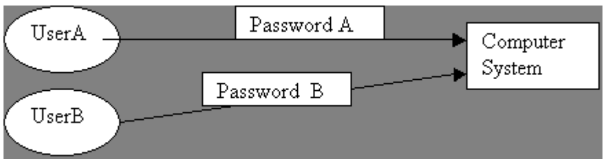

Fig. 3 The old model structure has no accounts or services.

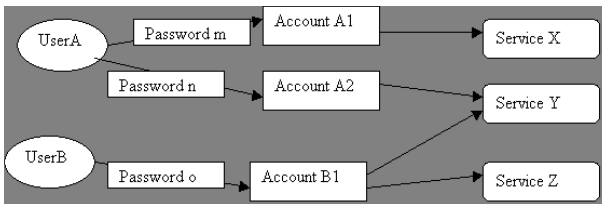

Fig. 4 Under the new model, `Users` may have any number of `Accounts` and thereby have access to one or more `Services`.

Moreover, different `Users` may be assigned access only to specific `Accounts`, and `Accounts` to `Services`. This gives the person using this simulation, the administrator, fine-grained control of how the `Simulation` components interact with one another. In addition, this control allows the separation of `Users` from their roles in the simulation.

4.2 Model Assumptions

While we have strived to create a model as rigorous as possible, any simulation model is forced to make assumptions. A model such as this requires translating observed trends into concrete rules and formulas. We have, however, been very careful about which assumptions to make; all assumptions described here are mentioned and shown to be supported by studies on human users in Subsection 4.3.

We begin by assuming that a password policy may have a direct impact on the financial health of an organization. Further, we assume that a successful attack against the information infrastructure of an organization can have cost beyond just the immediate loss of whatever income the effected services would have generated. We assume that if a single account which uses a service becomes compromised, then the service itself can become compromised.

We assume that the users of an information system are under constant attack on their accounts. Their passwords may be cracked by brute-force methods, and they may be compromised by human carelessness as well. We assume that users often write down their passwords, and further we assume that doing so can increase the likelihood that their passwords become compromised.

We assume that users opt for passwords as short and simple as allowed under their policy. We also assume that passwords become less vulnerable to brute force attacks as they become more complex. We assume that having more passwords can make a user less likely to recall a particular password, as can changing passwords more frequently.

4.3 Model Validation

The value of our model is predicated on the postulate that the simulation approximates genuine user behavior and its consequences. This, in turn, requires that the assumptions made in our model be reasonable. We have listed model assumptions above in Subsection 4.2. In this subsection, we consider published research involving real-world users. We show how each of the assumptions listed in Subsection 4.2 is validated and shown to be reasonable by studies on actual human users. This subsection thus validates our model.

Bishop et al. [1] conduct a study in which site administrators submit their password files to be cracked through an automatic cracking program. In an average system with 50 passwords, five to 15 are usually cracked in a single day; this bolsters our model's assumption that passwords are vulnerable to automatic attack. The paper by Bishop et al. also supports our model's assumption that a single compromised account for a given service leads the entire service to become compromised. Bishop et al. state that a single non-root account being compromised in a given system can be leveraged into any information on that system being compromised or destroyed. This study observes that users tend to opt for simple passwords. This is in agreement with our model assumption that users create passwords as simple as allowed by their policy. Bishop et al. also note that a written password is another source of being compromised, as is the case in our model.

Kuo et al. [3] conduct a survey in which users are asked to create two different passwords, one being under more restrictions than the other. For the standard passwords, with fewer restrictions, six percent are vulnerable to dictionary attacks, another five percent to a dictionary attack using permutations of words, and yet another eight percent to brute force attacks. This indicates that, as is the case in our model, passwords generated by humans are often subject to being cracked by automated, brute-force attacks. In addition, in our model, users can be set to create passwords of low entropy compared to the possible entropy for a password of a given length. This is in line with the authors' observation that humans seldom use random text for their passwords, but instead use characters with a frequency reflective of the character frequency in their language. Since Shannon [9] shows that English does not use characters with equal frequency, we know that this leads to lower total password entropy.

The idea that users use passwords as simple as allowed is further illustrated by research conducted by Leyden [4]. Leyden conducts a survey of office workers. Users tend to use simple, easily guessed passwords. "password" is the most common password, followed by the name of the user or a sports team. This bolsters our assumption that users select simple passwords when able to do so. Ninety percent of workers in Leyden's study are willing to divulge their passwords for a pen; as in our model, a system can become compromised through human as well as technical factors.

Polstra et al. [6] highlight findings from actual cases of computer security breaches. In our model, human errors can lead to a system becoming compromised; Polstra et al. point out that this is quite often the case for actual systems. Moreover, Polstra et al. provide data regarding the financial harm to corporations because of a successful attack. The harm arising from being compromised, they point out, is not limited to merely the loss of the compromised computer system. Rather, the negative attention and loss of good name brought about by a security breach can exceed the harm done to a company by actual data loss. This is reflected in our model. When a service is compromised, the harm done to the organization is not limited to losing some of the productivity of the service itself; there may be in addition a daily cost associated with being compromised. Further, Polstra et al. show that computer security is a business issue, not just a technical one. They demonstrate that a poor security policy can have financial consequences for an organization. This fact is reflected in our model, because security problems are translated into financial loss; thus the model depicts a security breach as a business loss.

Proctor et al. [5] perform experiments on password restrictions, in which users create passwords according to different criteria. In the first experiment users are divided into two groups, in which one is required to use a more strict password policy than the other. Using an automated cracking tool, 18 of the 24 less restricted passwords are cracked, while only eight of the 24 more restricted passwords are cracked. In the second experiment, users are required to use longer passwords, which results in fewer being cracked. It is found that users tend to make simple passwords unless forced to do otherwise, something reflected in our model. In addition, increasing the length of the password or increasing the restrictions placed on password generation reduces the likelihood that the password becomes cracked by an automated tool in the experiments. This matches our model in that increasing the complexity of the password, by increasing either its length or its per-character entropy, reduces its chance of being cracked.

A survey conducted by SafeNet [7] focuses on password usage in organizations. It finds that about half of employees write down their passwords. Over eighty percent of those employees also have three or more passwords. These passwords, in turn, are usually used to access multiple applications each. Almost half of the users who responded to the survey need to have their passwords reset at least once per year. Each of these findings matches our model. The model allows users to write down their passwords, to have multiple passwords, to tie a single password to multiple services, and to have the help desk reset a password should it be forgotten.

Vu et al. [12] perform experiments on the time and number of attempts required for users to create and recall passwords which satisfy various requirements. There are two salient experiments on users and passwords. In the first experiment, users are divided into two groups. The users in the first group create three accounts with different passwords and the users in the second group create five accounts each. The experiment studies the comparative recall of users in both groups. The main finding of the experiment is that users are significantly more successful with recall when they have three passwords rather than five; users in the five-account group are more likely to forget some of their passwords. In the second experiment, users are divided into two groups, and only one group is required to use a special character and a digit in passwords. This means that one group has more complex passwords than the other; in other words, one group has a higher required entropy for its passwords. Again, recall is compared for the groups. Users are less able to recall the more complex passwords; users forget the more complex passwords about twice as frequently. However, a cracking program is able to crack the passwords with fewer requirements 62 percent of the time, but only two percent of the time for the more complex passwords.

These two experiments by Vu et al. are in accord with our model. The first experiment shows that users are better able to recall passwords when they have fewer passwords to recall; in our model, password inundation can result in users in having greater difficulty recalling an individual password as there are more passwords to recall. Vu et al. note that difficulty in recalling passwords may arise from having too many passwords at once, and from too frequently changing a given password, as our model also represents. The second experiment indicates that a more complex password is more difficult for a user to recall, but is also more difficult to crack. That is also the case in our model; increasing the entropy of a password decreases the likelihood that a user recalls it, but also decreases the likelihood that it becomes cracked.

4.4 Other Relevant Works

Gehringer comments on the password-related difficulties faced by the modern user [2]. Lack of common password standards and policies can leave users struggling to recall different requirements for different systems, which can be confusing. Users can also be confounded by the number of passwords required to be memorized, making them more likely to write down their passwords.

Sasse et al. argue that security designers ought to identify and address root causes of poor user behavior [8]. They believe that a more user-centered design would help users behave in ways more consistent with good security practices. For example, complex and conflicting password policies can cause users to write down their passwords. The authors argue that while those creating security policies often view humans as enemies of security, policies created to help humans be good users are important to security. The focus on human factors is reflected in our model.

Summers et al. provide an analysis of passwords and their policies [11]. They recount many of the policy difficulties which our model takes into account. Users often select simple passwords when their policy permits, and this can lead to the password being easily cracked. On the other hand, passwords too difficult to remember can result in users writing them down. The authors also point out that overly frequent password changes can lead to problems.

5 Experiments

5.1 Experimental Methodology

This section describes four different experiments and their results, each performed using the simulation tool PPST. In the discussion of the methodology used in the experiments we use the following terms:

– A `run` is a single execution of PPST.
– A `runset` is a set of `runs`, each having the exact same input parameters.
– An `experiment` is conducted by creating a number of `runsets` which are similar to one another in input and examining how the differences in their inputs result in differing output.

For example, we may perform an experiment to assess how modification to password length impacts the final `balance` by creating a number of `runsets` which differ from one another only in password length. We may thus visualize an `experiment` by plotting a set of points to represent its runsets. Each point on the graph represents a `runset`. For each point, its $y-coordinate$ represents the mean final `balance` of the `runs` in that `runset`. Recall that, all else being equal, a higher final `balance` indicates better total security. Each point's $x-coordinate$ represents the value of the changing input variable for that `runset`, password length in the case of our example. In this way, the graphs indicate how changing the value of an input variable impacts security.

5.2 Password Entropy Experiment

This experiment assesses how changes to password complexity affect security. The input variables common to each `runset` are listed in Table 9. The value of `Account.charEntropy` changes between runsets and the other input values are constant. The results are shown in Figure 5. The $x-axis$ indicates the value of `Account.charEntropy`. This ranges from 1.0 to 5.9 increasing by increments of 0.1. According to the input in Table 9, users are very likely to succeed in memorizing for the low values of the range, and less likely to succeed for the higher values.

The results show that when passwords have low entropy, they are easily cracked. Passwords being cracked leads to a decreased `balance` because `accounts` and therefore `services` become compromised. However, when passwords become too complex, `users` have difficulty memorizing them. This leads to `users` writing down their passwords, which in turn leads to their `accounts` being compromised. Therefore, the best policy is to strike a balance between overly complex and overly simple passwords. Passwords ought to be complex enough to be difficult to crack, but not so complex that `users` are unable to recall them. That is why the `balance` is maximized around the middle of the graph, where `Account.charEntropy` is 2.7. To give an idea of what this might look like, consider that standard written English has a per-character entropy of around one [9]. A password composed randomly of uppercase and lowercase English letters, the digits 0 through 9, underscore, and space has a per-character entropy of six.

5.3 Password Inundation Experiment

This experiment analyzes how different frequencies in password change impact security. Each `run` in this experiment involves 64 `users` who are each given eight `accounts`. Each `account` connects to exactly one `service`, and no `service` is accessed by more than one `account`. Therefore, each `run` has $64*8 = 512$ `services`. Each `runset` contains 16 `runs`. The input values common to each `runset` are found in Table 9. The value of `Account.passExpire` is the same for each `account` in a given `runset` but varies across `runsets`. This value ranges from 5 to 300 increasing in increments of 5. This experiment is run for four years of simulated time, to highlight better the long-term impact of password inundation.

The results of this experiment are shown in Figure 6. These results indicate that when passwords are changed too frequently, the final balance greatly decreases. `Users` can become flooded by having to remember too many passwords too quickly. This results in their writing down those passwords rather than memorizing them, which in turn can lead to the passwords becoming compromised. The results also indicate that there is little difference between requiring passwords to be changed every 100 days and every 300 days. Although password changes can cause an account to no longer be compromised, this may be offset by the difficulty which `users` have with being inundated by new passwords.

5.4 Service Sharing and Attacks

This experiment compares multiple `users` sharing a single `service` with `users` each having their own copy of a `service`. In each `run` in this experiment, there are 32 `users`. This is a reasonable number of users for a small office environment. Each `runset` contains 256 `runs`. The input value that changes between `runs` is `Account.numAttacks`, the daily number of brute force attacks against the `accounts`. The input values that do not change between `runsets` are listed in Table 9. The value of `Account.numAttacks` ranges from 0 to 202, increasing in increments of one.

Unlike in the above two experiments, for each value of `Account.numAttacks`, there are two `runsets` created. They have the same input values, but differ in their composition. In one `runset`, all 32 users share a single `service`. In the other `runset`, each of the 32 users has an individual copy of the `service` to use.

The results of this experiment are shown in Figure 7. One line indicates the average final balance values for `runsets`

Variable	**Exp. 1**	**Exp. 2**	**Exp. 3**	**Exp. 4**
`Simulation.maxDays`	365	1460	365	365
`Number of Users`	100	64	32	var
`Users Share Services?`	No	No	var	var
`Runs per Run Set`	50	16	256	256
`Service.standardDailyCost`	2.0	10.0	10.0	10.0
`Service.compDailyCost`	2.0	5.0	5.0	5.0
`Service.compIncomeMult`	0.75	0.5	0.50	0.50
`Service.standardIncome`	20	30	30	30
`Account.numAttacks`	5	10	var	150
`Account.compWrit`	0.15	0.10	0.05	0.05
`Account.passExpire`	60	var	60	60
`Account.passLength`	6	6	8	8
`Account.charEntropy`	var	2.0	2.0	2.0
`Account.helpDeskCost`	4	5	2.0	2.0
`Account.probForget`	0.0	0.0	0.1	0.1
`User.dailyCost`	4	20	12	12
`User.memory`	0.75	0.50	0.75	0.75
`User.falsSuspect`	0.01	0.01	0.01	0.01
`User.trueSuspect`	0.05	0.05	0.05	0.05
`User.incomeMultiplier`	1.0	1.0	1.0	1.0
`User.inundationMax`	0.2	1.0	0.3	0.3
`User.inundationConst`	0.5	1.0	1.0	1.0
`User.learningCurve`	0.5	0.01	0.5	0.5

Table 9 Experiments' input variable values. The meanings of the variables are explained in Subsection 2.8.

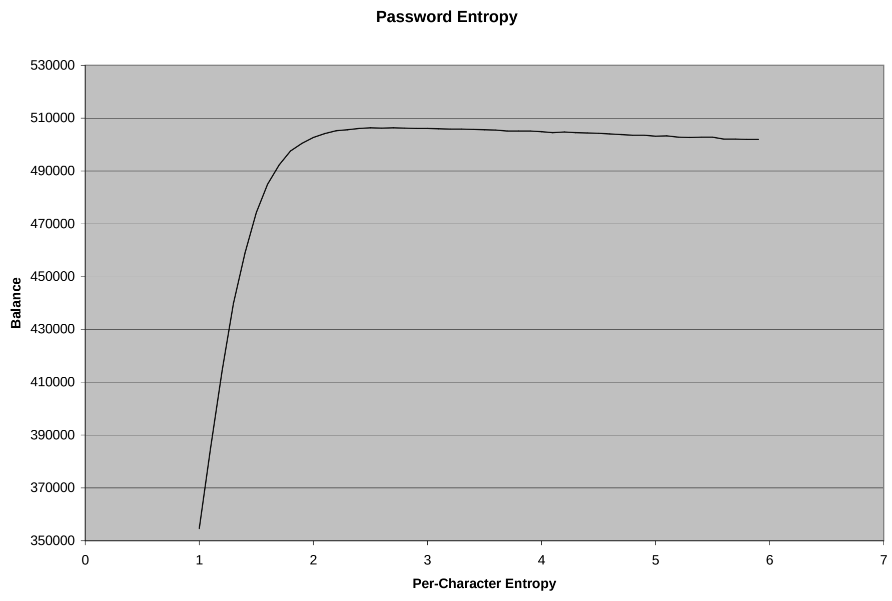

Fig. 5 Comparison of balance and password complexity

in which `users` share a single `service`. The other line indicates the balance for `runsets` in which `users` are each given their own `services`.

We see that when there are fewer attacks made against `accounts`, then it is better for `users` to share a single `service`. This reduces total cost because the daily cost of only a single `service` must be paid, instead of paying for

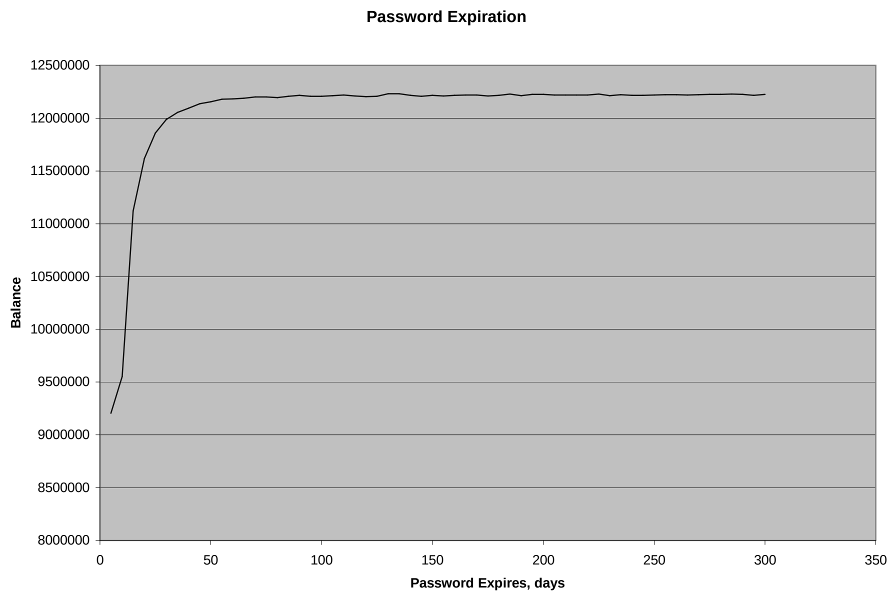

Fig. 6 Comparison of balance and password change frequency

one service per user each day. As show in Figure 7, these savings can be substantial. However, as the daily number of brute force attacks against `accounts` increases, using a single shared `service` becomes increasingly risky. This is because when `users` share one `service`, if any `account` using the one `service` becomes compromised, the entire `service` becomes compromised for all `users`. Whereas, if each `user` has an individual instance of the `service`, and that instance is compromised, the other `users` are not affected.

It is worth noting that the fixed parameters set forth in Table 9 have a significant impact on these results. If, for example, the passwords in this experiment were so complex that they were almost never cracked, there might be a higher final balance for `users` sharing a service for a higher value of `Account.numAttacks`

5.5 Service Sharing and Number of Users

Whereas the previous experiment explores the impact of `users` sharing `services` as the number of attacks increases, this experiment explores the impact of `users` sharing `services` as the number of `users` increases. As above, there are two groups of `runsets`, one with `users` sharing a single `service` and one with each `user` having an individual copy of the `service`. The only input value which changes between `runs` is the number of `users`, which ranges from 2 to 96 in increments of two. The input values that do not change between `runsets` are listed in Table 9. The number of daily attacks on each account, 150, is chosen because it is around where the two lines in the graph for the above experiment intersect.

For low numbers of `users`, there is a greater `balance` when those `users` all share a single `service`. This is because each `account` is individually unlikely to be compromised at any given time, and money is saved by supporting just one `service`. However, although increasing the number of `users` increases the `balance` in both cases, it increases more rapidly when each `user` has an individual `service`. When each `user` has an individual `service`, adding more `users` causes the `balance` to grow linearly. This is not the case when one `service` is shared by all `users`, because each added `user` increases the potential that the single `service` becomes compromised, thereby reducing the income of all `users`.

6 Conclusion

Current research indicates that while passwords are used to protect increasingly valuable assets, creating an optimal password policy for a given organization remains an open

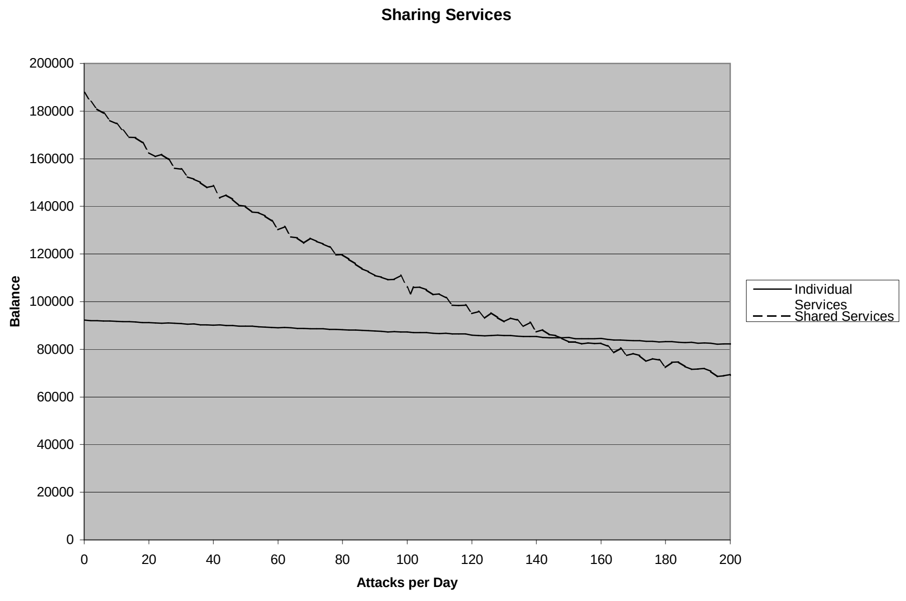

Fig. 7 Comparison of balance, attack frequency, and users sharing services

problem. In order to address this, we offer a password policy simulation tool, PPST.

PPST is based on a model providing a detailed representation of password policies and users, as well as organizational parameters such as costs and services. Using this tool we conduct several experiments.

While there are existent studies on human users and passwords, our work is novel in several ways. We offer what we believe to be the first comprehensive password policy simulation tool. While some studies present only heuristics or statistical information to policy-makers, we provide a practical utility to enable policies to be studied before implementation and deployment. No two organizations are exactly alike, and different security situations require different responses; therefore studies of one organization are only of limited utility to another organization. Using PPST, an organization can generate data tailored to its own requirements.

In addition, our simulation model presents a valuable new asset to security researchers. As illustrated in Section 5, PPST allows experiments to be conducted in which one or two variables are changed, and the rest are constant. This allows researchers to study the effects of different factors on password policy success in isolation. Doing this using actual human users would be difficult, expensive, and slow.

PPST lets researchers and administrators analyze different aspects of password policy design without needing to experiment on human users. This may lead to the creation of more solid password policies, and may facilitate further research.

References

1. Bishop, M., V.Klein, D.: Improving system security via proactive password checking. Computers and Security 14 **14**(3), 233–249 (1995)
2. Gehringer, E.F.: Choosing passwords: Security and human factors. Technology and Society, 2002. (ISTAS'02) p. 369373 (2002)
3. Kuo, C., Romanosky, S., Cranor, L.F.: Human selection of mnemonic phrase-based passwords. In: SOUPS '06: Proceedings of the second symposium on Usable privacy and security, pp. 67–78. ACM Press, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1143120.1143129
4. Leyden, J.: Office workers give away passwords for a cheap pen. The Register (2003). URL http://www.theregister.co.uk/-2003/04/18/office_workers_give_away_passwords/
5. Proctor, R.W., Lien, M.C., Vu, K.P.L., Schultz, E.E., Salvendy, G.: Improving computer security for authentication of users: Influence of proactive password restrictions. Behavior Research Methods, Instruments, & Computers **34**(2), 163–169 (2002)
6. Robert M. Polstra, I.: A case study on how to manage the theft of information. In: InfoSecCD '05: Proceedings of the 2nd annual conference on Information security curriculum development, pp. 135–138. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1107622.1107653
7. SafeNet: 2004 annual password survey results. SafeNet (2005). URL http://www.safenet-inc.com/Library/10/2004passwordsurveyresults.pdf

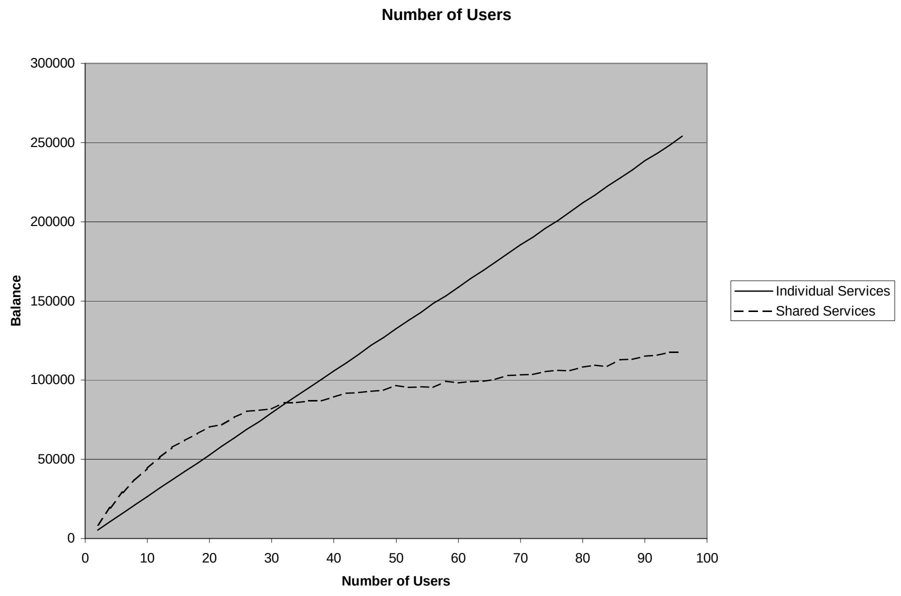

Fig. 8 Comparison of balance, number of users, and users sharing services

8. Sasse, M.A., Brostoff, S., Weirich, D.: Transforming the 'weakest link' — a human/computer interaction approach to usable and effective security. BT Technology Journal **19**(3), 122–131 (2001). DOI http://dx.doi.org/10.1023/A:1011902718709
9. Shannon, C.E.: Prediction and entropy of printed English. Bell Systems Technical Journal **30**, 50–64 (1951)
10. Shay, R., Bhargav-Spantzel, A., Bertino, E.: Password policy simulation and analysis. In: DIM '07: Proceedings of the 2007 ACM workshop on Digital identity management, pp. 1–10. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1314403.1314405
11. Summers, W.C., Bosworth, E.: Password policy: the good, the bad, and the ugly. In: WISICT '04: Proceedings of the Winter International Symposium on Information and Communication Technologies, pp. 1–6. Trinity College Dublin (2004)
12. Vu, K.P.L., Proctor, R.W., Bhargav-Spantzel, A., Tai, B.L.B., Cook, J.: Improving password security and memorability to protect personal and organizational information. International Journal of Human-Computer Studies (2007)
13. Yan, J.J.: A note on proactive password checking. In: NSPW '01: Proceedings of the 2001 workshop on New security paradigms, pp. 127–135. ACM Press, New York, NY, USA (2001). DOI http://doi.acm.org/10.1145/508171.508194

```
forall Account $A$ do
    $A.AssignNewPassword$;
end
for $DAY$ : 1 to Simulation.$maxDays$ do
    forall Account $A$ do
        if $A.passwordHasExpired$ then
            $A.AssignNewPassword$ ;
        end
        else if $A.isSuspectedCompromised$ then
            $A.CallHelpDesk$ ;
            $A.AssignNewPassword$ ;
        end
        else if $A.passwordIsWritten$ then
            if $A$.memCheck() then
                $A.passwordIsWritten \leftarrow$ false ;
            end
        end
        else if
        $!A.\mathbf{memCheck}()\mathbf{ANDrandom}[\mathbf{0}-\mathbf{1}] < \mathbf{A.probForget}$
        then
            $A.CallHelpDesk$ ;
            $A.AssignNewPassword$ ;
        end
    end
    forall Account $A$ do
        if $A.isWritten$ then
            with probability $A.compWrit$, $A.isCompromised \leftarrow$
            true ;
        end
        if $A$.bruteForceAttacksSucceed() then
            $A.isCompromised \leftarrow$ true ;
        end
    end
    forall Service $S$ do
        $S.isCompromised \leftarrow$ false ;
    end
    forall Account $A$ do
        if $A.isCompromised$ then
            forall Service $S$ : $A.services$ do
                $S.isCompromised \leftarrow$ true ;
            end
        end
    end
    calculateIncomeCost() ;
end
```

Algorithm 1: Model Algorithm