Finger Printing Data

Gideon Samid
Department of Electrical Engineering and Computer Science
Case Western Reserve University, Cleveland, OH
BitMint, LLC
Gideon@BitMint.com

Keywords: identity-theft, information-assurance, database breach, data recovery

Regular Research Paper

Abstract: **By representing data in a unary way, the identity of the bits can be used as a printing pad to stain the data with the identity of its handlers. Passing data will identify its custodians, its pathway, and its bona fide. This technique will allow databases to recover from a massive breach as the thieves will be caught when trying to use this 'sticky data'. Heavily traveled data on networks will accumulate the 'fingerprints' of its holders, to allow for a forensic analysis of fraud attempts, or data abuse. Special applications for the financial industry, and for intellectual property management. Fingerprinting data may be used for new ways to balance between privacy concerns and public statistical interests. This technique might restore the identification power of the US Social Security Number, despite the fact that millions of them have been compromised. Another specific application regards credit card fraud. Once the credit card numbers are 'sticky' they are safe. The most prolific application though, may be in conjunction with digital money technology. The BitMint protocol, for example, establishes its superior security on 'sticky digital coins'. Advanced fingerprinting applications require high quality randomization. The price paid for the fingerprinting advantage is a larger data footprint -- more bits per content. Impacting both storage and transmission. This price is reasonable relative to the gained benefit.**

I. INTRODUCTION

Data normally is 'non-sticky', so its handlers don't leave a fingerprint on it. Data, normally, does not contain the information as to how many readers it had, and rarely who was its writer. This fact is so ruthlessly exploited in cyber crime. We all await the practical manifestation of quantum computing theories which promise to bring 'fingerprinting sensitivity' to data, but until then, and with much more simplicity, we propose here a conventional way to represent data so it is 'sticky' -- it bears the finger prints of its writer and readers.

The fundamental principle is simple and straightforward: all data can be expressed as integers, all integers can be represented as a series of bits where the count of bits reflects the data carried by the string. Accordingly all 2^n possible n-bits strings will carry the same value, n. The range of 2^n possible strings all representing the same value n, may be used as meta data associated with the prime data (n), and this meta data may be regarded as 'fingerprinting' the primary data, n.

Nomenclature. Fingerprinted data will be denoted with a right side underscore: data_. where:

$$data\_ = value\_identity$$

and write:

$$value = data\_v,\ identity = data\_i$$

Example: the value x=6, represented as 000001, will be also written as 6_1, and if represented as 011111 will be written as 6_31.

While value ranges from 0 to ∞, identity ranges from 0 to 2^{value} -1: $0 \le value \le \infty$; $0 \le identity \le 2^{value} - 1$

We shall use the term 'identity', 'shadow', 'fingerprint' interchangeably.

Data, d, not expressed in the fingerprinting mode will be regarded as 'naked data'. x=12 is naked, x=111000111000 is 'dressed data' or 'fingerprinted data', 'shadowed data', 'identified data'.

Let TM be a Turing machine mapping some input_v to a certain output_v. TM will be associated with a shadow Turing Machine, TM_ which will map the input_i to output_i. TM_ data range is 0 to 2^{output_v}, or say: output_i → output_i MOD 2^{output_v}.

For example: let TM be: c= a+b, and let TM_ be: c_i = (a_i + b_i) mod 2^{c_v}.

Numerically: let a = 4_6 , and b = 7_107, expressed as: a = 0110 and b=1101011 . We shall compute c_v = a_v + b_v = 4+7 = 11, and compute c_i = a_i + b_i = 6 + 107 = 108 MOD 2^{11}, = 00001101100. Or say:

$$0110 + 1101011 = 00001101100$$

We assume the algorithmic data in the Turing Machines to be naked. So for Turing Machine TM: b = a + 5, a and b

may be 'shadowed' but the constant '5' will be naked. Hence TM_ may be defined as b_i = (a_i +325+ δ)/2, where δ=1 for an even a_i, and δ=0 otherwise. Hence for a=110 (a_v=3, a_i=6), we write: b_v = a_v + 5 = 3 + 5 = 8, and b_i = 6 + 325+1 = 332 mod 2^8 = 76. So we write:

01001100 = TM(110)

Since fingerprinting applies to data, it will not affect algebraic signs, operational marks, or imaginary numbers notation. Say then that x = -4 will be written as -0000, -0001, -0010,.... -1111, and ***i***5, will be written as ***i***00000, ***i***00001,...***i***11111.

Irrational numbers cannot be 'fingerprinted' but they are never an output of a Turing Machine. Any rational approximation thereto will be written as a ratio of integers. Thus π may be approximated to 3.14, or 314/100. $2^{0.5}$ may be approximated to 141/100 We assume the algorithmic data in the Turing Machines to be naked. So for Turing Machine TM: b = $(a\text{-}7)^{0.5}$, we may define the associated Turing Machine TM_ : b_i = $(a\_i)^2$ mod 2^{b_v}.

Hence for a=11010 (a_v=5, a_i =26), we have b_v = $(a\_v - 7)^{0.5}$ = ***i****$2^{0.5}$. where ***i*** = $(-1)^{0.5}$. The square root of 2 is computed by a Turing Machine with some finite resolution: $2^{0.5}$ = g/h, where g and h are integers. TM_ will determine g_i, and h_i. Say g_i =654321 mod 2^{g_v}, and h_i = |a_i - 50| mod 2^{h_v}. For economy of display we use a low resolution: $2^{0.5}$ = 1.4 = 14/10. Namely g_v = 14, h_v= 10. We have then g_i = 654321 mod 2^{14} = 15345, and h_i = |26-50| = 24, and thus we write:

*i * 11101111110001/0000011000 = TM(11010)*

Resolution: It would appear that the shadow Turing Machines are limited by the MOD limitation, so that when the output of the corresponding Prime Turing Machine is a small integer value x → 0, then the scope of the shadow machine, limited to 2^x will severely limit its operation. In fact this resolution limitation is readily overcome. For value output x < 1, a corresponding fraction x = y/z will allow one to use sufficiently large values so that 2^y, and 2^z will satisfy any desired resolution. For integer output x such that 2^x is too small, the shadow Turing machine could use: x = y-z, and, like before use any desired size for y and z. We conclude then that the shadow Turing machines are as general in practice as a general Turing machine.

Multiplication: The simplest way to adjust resolution is by multiplication. Instead of reading the bit count as the represented data, one can carry the integer n by a bit string comprised of kn bits, k=1,2,.... where k is a user's choice allowing for any desired resolution.

Count-to-Value Formula: Mapping bit count (n) to value (v) may be carried out via some choice formula, f: v = f(n). For example: v = kn+b, which will expand on the multiplication option discussed above, and will add a base, b, to insure that the value of zero comes with a sufficient range of shadow values (2^b).

Complexity: A shadow Turing Machine may issue a constant output y_i, regardless of the input data. In this case the shadow Turing Machine (STM) will offer a fixed signature identifying the machine. It can compute its output y_i based on the value x_v of the input, or on the value x_i of the input, or on both parts of the x data.

Basic machine tracking: We consider a Turing Machine TM_1 and its shadow $TM_{1\_}$, and another Turing Machine TM_2, and its shadow Turing Machine $TM_{2\_}$.

cases:

Case I: $TM_1 = TM_2$; $TM_{1\_} = TM_{2\_}$

in which case, upon examination of the input and the output, it will be impossible to determine whether TM_1 or TM_2 processed the input.

Case II: $TM_1 = TM_2$; $TM_{1\_} \neq TM_{2\_}$

In this case, an examination of both the input and the output will expose whether TM_1, or TM_2 has processed the data. We have here a basic tracking procedure.

Case III: $TM_1 \neq TM_2$; $TM_{1\_} \neq TM_{2\_}$

In this case, an examination of both input and output will identify which machine processed the data. However, examination of only the output data might, or might not determine which machine processed the data because there may be one possible input that would fit with the hypothesis that TM_1 was working here, and another input corresponding to TM_2.

The general Tracking Case: Given a 'computing environment' comprised of t Turing Machines: TM_1, TM_2,.... TM_t, and their corresponding shadows: $TM_{1\_}$, $TM_{2\_}$, ... $TM_{t\_}$, given input x to this environment, and a corresponding output y. We shall define the notion of a 'computing sequence' as a data processing configuration leading from x to y. The configuration will identify input and output for all Turing Machines, any splits of data, and any combinations of data. A data item z may be split as input to two or more Turing Machines, and any Turing Machines may be operating on any number of input data items. We now ask what are the computing sequences that would satisfy this given set of parameters.

{fingerprinting solutions} to satisfy {TM_1, TM_2,.... TM_t, $TM_{1\_}$, $TM_{2\_}$, ... $TM_{t\_}$, x_v, x_i, y_v, y_i }

Every computing sequence that satisfies these terms will be regarded as a fingerprinting solution.

There may be no solutions to a set of parameters, one solution, or several.

The important observation here is that given a computation environment where there exists more than one computing sequence that would be compatible with a pair of input-output, as analyzed per the prime set of t Turing Machines (TM_1, TM_2,....TM_t) , with no knowledge of (or non existence) the corresponding t shadow Turing Machines,

then this equivocation can be eliminated via a proper set of Shadow Turing Machines that for sufficient amount of processed data all but one computing sequence will be eliminated. This is the fundamental tracking idea of the fingerprinting concept.

The Fundamental Theorem of Data Fingerprinting: Given a computing environment with t arbitrary Turing Machines, there exists a set to t corresponding Shadow Turing Machines that would eliminate any computing sequence equivocation which may arise, given the first set of t Turing Machines.

Proof: Consider two computing sequences, each taking a given input x_v to a given output y_v. The last Turing Machine in the first sequence is TM_1, and the last Turing Machine in the second sequence is TM_2. The first machine is activated with input x_1_v, and the second with input x_2_v. It may be that x_1_v = x_2_v, or that x_1_v ≠ x_2_v. But their output is the same: y_1_i = y_2_i. One will then set:

$$y_1\_i = TM_1\_(x_1\_v, y_1\_v) \neq y_2\_i = TM_2\_(x_2\_v, y_2\_v)$$

And thereby will eliminate this equivocation.

This procedure will continue over any two equivocated computing sequences. This may lead to a conflict where some Shadow Turing Machine i, which was adjusted once when it removed equivocation involving Turing Machine 1, has to change again to resolve an equivocation raised with respect to Turing Machine 2. Let the status of TM_i_ have originally been defined as function f_a, and to resolve the first conflict it changed to function f_b. But f_b is in conflict with another equivocation. This will only mean that TM_i_ will have to change to a function f_c which is $f_c \neq f_b$, and $f_c \neq f_a$. To insure that such a third function will be available, one has to insure that the resolution of the shadow functions is sufficiently large. We have seen that resolution can be adapted and increased at will. That means that no matter how many cases of double equivocation will be there, one will be able to construct a shadow Turing Machine that will eliminate all such equivocations.

This universal ability to eliminate any a two pathway equivocation can be applied step by step to eliminate any three-some, four-some or n-some equivocation, which proves the theorem.

Lemma: It is always possible to construct a set of Shadow Turing Machines that would reduce computing sequence equivocation to any desired degree. Proof: the proof of the fundamental theorem was constructed as elimination of equivocation pathways one at the time. One could simply stop such elimination when only some k > 1 computing sequences remain.

This is quite an intuitive conclusion, which is of theoretical import, but of very little practical significance. From a computer engineering point of view, the question is how easy, how simple, how unburdensome is it to eliminate computing sequence equivocation with a set of Shadow Turing Machines.

The straight forward use of this fingerprinting is deterministic, as will be illustrated ahead. Apart from it, fingerprinting may be applied via randomization and modulation.

A. Value-Identity Separation

Obviously a network data flow can be analyzed per the value of the flow items (x_v, y_v), ignoring the shadows. Less obvious is the reverse, where one is tracking the flow through the shadow only, without being aware of the value.

We have indicated the general case where the value of a bit string, y_v, is evaluated via some formula f with the bit count, b as argument: y_v = f(b). If f is unknown, then knowledge of b alone does not indicate the corresponding value. This implies that one could analyze a network data flow by checking value and identity (shadow) simultaneously, or each of them separately.

The significance of this separation is in the fact that very commonly the people focused on the value part of the data are different than the people focusing on the identity part of the data. The value people don't wish to be burdened by the identity info, and those charged with forensic tasks to track data may not need to be exposed to the contents (the value) of the data they are tracking.

II. RANDOMIZATION & MODULATION

The purpose of the shadow is to fingerprint data, not to carry specific data values. This important distinction may be readily exploited through randomization.

In a deterministic shadow environment the various computing machines will have to coordinate their shadow operation in order to insure the desired fingerprinting. This may be impractical in environments with a large number of computing machines. By contrast randomization allows for shadow operation without coordination.

Uncoordinated Shadow Machines: Let a computing environment be comprised of t Turing Machines TM_1, TM_2,.....TM_t. Let the corresponding shadow machines TM_1_, TM_2_,.....TM_t_ each be fully randomized. Namely given the primary value y_v_j j=1,2,...t, they will specify the identities of the $|y\_v_j|$ bits in a "purely randomized way" (or close enough to it) and keep a record of y_i_j.

Even if all the t y_v values are identical, for a sufficient bit size of the outputs, the chance for a collision can be set to be negligible. A collision here is a state where two Turing Machines will randomly select the same y_i so that it would not be clear which one of them processed the data. We have here a situation where *probability calculus enables a computing environment to work without pre-coordination.* Suppose that the bit count of all the y_v values is n=17. Let the computing environment be

comprised of t=1000 Turing Machines. The chance for a collision will then be:

$$Pr[shadow\ collision] = 1 - (1-2^{-n})^{n} = 1 - (1-2^{-17})^{1000} = 1\%$$

And that probability vanishes for n>17.

Alternatively the machines will use a standard mapping algorithm to create the base shadow for their output, and then randomly flip 50% (or close to it) of these bits. The same calculus applies, the chance for a collision can be made as small as desired.

Consider a reading situation involving t readers (t Turing Machines). Let an input x be distributed linearly among those readers, and the output is x_v = y_v. Using y_i one will be able to identify the exact sequence of readers of this information given that every reader flipped about 50% of the incoming bits. It is straight forward to compute the chance for any pathway equivocation, and reduce it as necessary by increasing the bit count. In particular consider the process of authentication. A network user offers his account number, PIN, or even password to prove her credentials.

A host of powerful applications is being opened by adding modulation on such randomization.

A. modulation

Consider a computing environment comprised of t readers, each applying a randomization strategy for shadow setting. The expected Hamming distance between any two arbitrary outputs y_i_k, y_i_j, is 0.5n, where n is the value of y_v_j (let's say, they are all the same). Alternatively stated, the probability for a Hamming distance of H much smaller than n/2 is small:

$$Pr_{collision}[H << n/2] \rightarrow 0 \text{ for for some t readers, for } n \rightarrow \infty$$

This fact implies that by flipping a sufficiently small number of bits in y_i, one will not harm her ability to track which reader read y_v recently. Such flipping is called modulation.

It implies that a y_i may carry around secondary messages in the form of modulation.

Modulation will allow one to authenticate a prover without having a copy of the authentication data. It offers a capability similar to more common zero-knowledge protocols. Only that it does not resort to the algorithmic complexity used in those protocols (and their vulnerabilities). It is based on simple combinatorics.

B. Superposition of randomization over determination

We have seen above that shadow randomization brings to bear specific advantages not present in a deterministic shadow formula. It bring about a much better resistance to hacking, and it opens the door for modulation. On the other hand a deterministic shadow sheds light on the inner working of the Turing Machine and allows for advanced forensic and tracking power of a given data flow. It is therefore of some advantage to combine the two varieties. One would associate a given Turing Machine with a deterministic shadow TM_, and then superimposed on it with a randomized operation, marked as TMρ_. We write:

$$y\_i = TM\rho\_(TM\_(x)) \text{ where } y\_v = TM(x\_v)$$

Accordingly every Turing Machine, TM, will be associated with two shadow machines: one deterministic TM_, and one randomized TMρ_

Superposition Illustration:Let a Turing Machine TM be defined as $y=x^2-64$, or say $y\_v = (x\_v)^2-64$.

Let the associated deterministic Turing Machine TM_ be defined as follows: (i) let $y^* = 11(x\_i)^2$ Let y'_i = {the y_v leftmost bits of y*, for $y\_v \leq y^*$, padding with zeros otherwise}.

Let the associated randomized Turing Machine, TMρ_ be defined as follows: a seed based randomization apparatus will generate a pseudo-random sequence, R. The generated bits will be taken y_v bits at a time, and associated by order to the bits in y*. This will build a series of y* bits, one after the other. Each bit in y* will be associated with the sum of the corresponding bits in the series of y* randomized bits. This process will stop when one of the bits in y* is associated with a greater sum than all others. The "winning bit" will flipped. This will be repeated q time.

For example, let x=9 written as 100111011, namely x_v=9, and x_i = 315.

TM: $y\_v = (x\_v)^2-64 = 17$.

TM_: $y^* = 11(x\_i)^2 = 11 * 315^2 = 1{,}091{,}475 = 10000101001111001001 1_{binary}$. And y'_i = 1000 0101 0011 1100 1 (the 17 rightmost bits in y*)

Now we need to superimpose the randomized flipping: activating the randomizer, one gets the following first batch of |y_v|=17 bits: 1100 0001 0101 1111 0. There is no clear winner. So the next batch of 17 random bits is invoked: 0011 0001 1100 1011 1. Adding the bits:

```
1100 0001 0101 1111 0
0011 0001 1100 1011 1
--------------------------
1111 0002 1201 2022 1
```

There are four bits scoring 2, no clear winner, so another batch is invoked:

```
1111 0002 1201 2022 1
0011 0011 0111 0100 1
- - - - - - - - - - - - - - -
1122 0013 1312 2111 1
```

There are 2 bits with a score of 3, so another batch is needed:

```
1122 0013 1312 2111 1
0111 0011 0011 1110 0
```

- - - - - - - - - - - - - - - -

1233 0024 1323 3221 1

This time we have a winner, bit 8, counting from the left has a score of 4, more than all others. So bit 8 in y'_i is flipped. If TMρ_ prescribed only one bit to flip then the final superimposed output is:

$$y\_i = 1000\ 0100\ 0011\ 1100\ 1$$

In summary: the illustrated node (Turing Machine) accepts: 100111011 as input, and generates: 1000 0100 0011 1100 1 as output.

III. FLIPGUARD: DATABASE PROTECTION

Databases holding private data of many users are a natural hacking target. Especially because users use the same private data in many databases. So a hacker can compromise the least protected database, and use the stolen data to fake credentials in many other databases. In the scope of so many databases today, there are bound to be some that are poorly protected and end up compromised.

By applying the fingerprinting technique, it is possible to distinguish between private users' data held by the user, and the same data held by the database. Such that if a database is compromised, and a hacker turns around to use the stolen data to falsely claim credentials then, not only would he not be admitted, but the database will readily realize that the submitted data marked with the database fingerprinting is evidence of the database being compromised. The latter is quite important because successful hackers hide their success for many months at times.

Here is how to carry out this fingerprinting protection of a database.

We consider a database serving a large number of users. The database holds private information for each user. Let X represent such private information of an arbitrary user of an arbitrary database. Let X be fingerprinted so that:

$$X_u\_v = X_b\_v$$
$$and$$
$$X_u\_i \neq X_b\_i$$

where X_b, and X_u are the values of X held by the database and the user respectively. The non-equality between X_u_i and X_b_i is due to modulation.

This arrangement will allow the database to recognize an access applicant purporting to be the user with the X credentials. The recognition will be due to a minor difference in the bit compositions of the two values, consistent with the applied modulation. However, if the source of the credentials (X) is a successful hacking of the database, then the database will find: X_u_i = X_b_i, (no modulation present), and will be alert to this fact.

Of course, if a hacker compromised the user he would be able to pose as the bona fide user, using the user's fingerprint: X_u_i, and be admitted. This FingerPrinting technique (code named FlipGuard) is designed solely to protect against a "wholesale" hacking risk, compromising the database. It provides no protection against "retail" hack, one user at a time.

This is the basic idea, which has to be well built to make it stick. We call it the randomization fingerprinting protection level 0. An ignorant database hacker, unaware of the fingerprinting will be readily caught. Albeit, it is unrealistic to assume that this technique can be applied in secret. One must assume that a hacker smart enough to break into a database will be smart enough to realize that fingerprinting is in force, and strive to break it too.

We shall therefore proceed with describing how to implement database protection against a the smartest hacker we can imagine. Before that we will describe hierarchical application of the database fingerprinting technique.

A. Hierarchical Applications

We consider a 'top database, B, and a secondary database B'. There are individuals who are logged as users both in B and in B'. A typical such user will use some private data X in both databases. For example: name, social security number, address, salary information, professional credentials, etc.

We assume that the top database, also called the issuer database, is practicing fingerprinting operation with its user, hence each user has its X data marked as X_v and X_i.

The issuer, B, can share its own version for each X (X&b_v, X_b_i) with the secondary database, B', namely:

$$X_b\_v = X_b'\_v$$
$$X_b\_i = X_b'\_i$$

(b, and b' indices indicate the issuer database and the secondary database respectively). And in that case the secondary database will function with the same protection as the issuer database. This solution can be extended to any number m of secondary databases B'_1, B'_2,....B'_m. The problem with this solution is that (i) if a compromise is detected, it is not clear which of the (m+1) databases was hacked, and (ii) the security of the most secure database is reduced to the security of the least secure database in the list.

An alternative strategy would be for the issuer database to pass on to the secondary database, a different shadow:

$$X_b\_v = X_b'\_v$$
$$X_u\_i \neq X_{b'}\_i \neq X_b\_i$$

And if there are several secondary databases, then each will be given a unique shadow. All the shadows will be randomized so that they would be able to admit a user while being immunized against a breach into their database. And should any database in the strategy become compromised, then upon any attempt to use the compromised X data, the system will spot it, and recognize which database was breached.

B. Advanced Finger Printing Protocol

We consider a smart attacker who knows everything about the defense strategy except the actual values of the protected data. Such an attacker is assumed to have compromised the database. The attacker would know that the database shadow data is different than the user's shadow data and if he would try to log in, using the compromised X values, as copied from the database, then the database will not only not admit him, but will be alerted to the fact that the database was compromised. The attacker would further know that the database does not have the exact user shadow. It only knows that the user's shadow is similar to the database shadow. So all that the hacker has to do is to randomly affect some small changes in the stolen shadow data, and forward the altered data to gain access, and pass as the bona fide owner of that X data.

If the changes induced by the attacker are such that the database would consider the difference between the attacker offered data, and the database respective data, as 'normal, or 'acceptable' then the decision would be to admit the hacker, and the protection would fail. Note: similarity between strings is measured through the Hamming distance between them.

To counter this eventuality one could opt for a countermeasure strategy based on "off line repository". The idea here is make an exact copy of the user shadow (X_u_i), and remove this copy from the active database, safekeeping it on an external system where it will have to be handled manually, locally, totally un-accessible to any online command. When a hacker forwards stolen X data, reasonably modified, then the hacker will be admitted, but the database will retain a copy of the X_i that was used to attain access, and every so often the database will take all the admitted users and compare their admission string to the one manually extracted from the off line repository. This comparison will readily reveal that the database was fooled by a hacker and would further disclose that a database was in effect compromised. That is because it is highly unlikely that the fraudster would have guessed a string of sufficient size n such that its Hamming distance from the copy held by the database would be so small.

This counter measure, designated as fingerprinting randomization level 1, will alert a database on a breach as often as the off-line repository is consulted, which may be too infrequent.

IV. Deterministic Applications

Let us now discuss some practical situations for which fingerprinting may be useful:

Layered Application: Data issued by a top source to a client may be used between the client and a secondary agent per the nominal value, and only with the top source per the nominal and shadow value. Thereby the data itself which is shared with several secondary agents may be stolen from one of them, but that would not be enough for the thief to defraud the top source because the thief would not have the shadow information.

A. Who Done It? Who Read It?

The simplest and most straightforward application of fingerprinting of data is to associate t value identical Turing Machines, each with a unique shadow Turing Machine. Namely set up a computing environment comprised of t Turing Machines such that:

$$TM_1 = TM_2 = ... TM_t$$
$$and$$
$$TM_{i\_} \neq TM_{j\_} for\ i \neq j\ i,j=1,2,...t$$

For every input x to this computing set, one of the t Turing Machines will compute a corresponding y=f(x), such that $y\_v_1 = y\_v_2 = ... y\_v_t$, but $y\_i_k \neq y\_v_j$ for $k \neq j$ for k,j=1,2,...t . This configuration will allow one who knows the computing set to determine which of the t Turing Machines processed the input.

In the case where each of the t primary Turing Machines are neutral, this will turn into 'who read it?' case. Namely if for any i=1,2,...t $y\_v_i = x\_v$, then this configuration will identify which Turing Machine read the input.

It will be easy to adjust the shadow Turing machines to handle the case where a given input x is read by some $r \leq t$ Turing Machines, the identity of which is readily ascertained. One simple way to accomplish this is to use any resolution extension discussed above to insure that y_v is comprised of at least tn bits: $y\_v \geq nt$, of some positive integer n, and define shadow Turing Machine, TM'_i as flipping bits in+1, to in+n. By examining y_i, one will readily determine which are the r Turing Machines that read the input data x.

Illustration let x=110011001100110011001100, Let x_v= (|x|-6)/2 = (24-6)/2 =9 , x_v =13421772. Let the computing environment be comprised of t=3 Turing Machines TM_1, TM_2, TM_3 which are all neutral, namely the corresponding outputs are: $y\_v_1 = y\_v_2 = y\_v_3 = x\_v = 9$. The corresponding three shadow Turing Machines will be: $TM_{1\_}$ = flip bits 1,2, $TM_{2\_}$: flip bits 3,4, and $TM_{3\_}$: flip bits 5,6.

Let's mark the 18 rightmost bits as R, so that we can write x=110011R. If the output will be 000011R. The table below lists all the possible combinations regarding who read the input. If the output is not one of these 8 options then it will indicate some error, or an unidentified reader.

y	TM-1	TM-2	TM-3
110011R			
000011R	x		
111111R		x	
110000R			x
001111R	x	x	

000000R	x		x
111100R		x	x
001100R	x	x	x

B. Sequencing

A computing environment with t Turing Machines, takes an input x, and have $r \leq t$ machines read it, and then output it. One could define shadow Turing machines such that the output will not only identify which machine was exposed to the input but also in which order.

One such configuration is as follows: set $x\_i = \{o\}_n$ (000...0). TM_i will count i bits in the the rightmost continuous string of zeros, and then flip the next i bits. The result (given that $n > t(t+1)$) is unique for each sequence.

Illustration: let x = '0000000000000000' (x_v=16). Let the reading sequence be TM_1, TM_2, TM_3. We will have then: y_i_1 = 0100000000000000, y_i_2 = 0100110000000000, y_i_3 = 0100110001110000 (the final output).

For a reading sequence TM_2, TM_3, TM_1 we have y_i_2 = 0011000000000000, y_i_3 = 0011000111000000 , y_i_1 = 0011000111010000 (the final output).

For a reading sequence TM_3, TM_1, TM_2 we have y_i_3 = 0001110000000000, y_i_1 = 0001110100000000 , y_i_2 = 0001110100110000 (the final output).

It is easy to see that every distinct sequence (complete or partial) will be mapped to a unique shadow value of the output, and therefore ascertained by it.

C. Reconstruction

Reconstruction is a more complicated case but related to sequencing. It applies to a computing environment where an output y may have been computed via a relatively large number of pathways, and it its occasionally needed to find the exact path, to particular Turing Machines that worked on the input to generate the output.

One practical situation is when an input x is processed through p rounds of distinct calculations, such that x is computed $x \rightarrow p_1$, and p_1 is computed to p_2, and in general $p_i \rightarrow p_{i+1}$. and finally $p_{g-1} \rightarrow y$. Each of the g calculations can be done via some q fitting Turing Machines. The selection among them is done based on some criteria, say, load. This configuration creates a pathway space comprised of q^g options. The corresponding Shadow Turing Machines will have to record at least q^g distinct values in order to determine from the output the exact computational history of the output. This of course is very important if one tries to chase a bug, or hunt for malware.

D. External Intervention Detection

If the shadow results do not fit any computational path within the computing environment then, apart from some coding error, the suspicion must be raised over the prospect of intrusion, and substitution of a proper Turing Machine with an improper one.

Of course, if a hacker knows the nominal algorithm of the Turing Machine as well as its shadow algorithm then he is left undetected. And that is a good reason to change the shadow algorithms often enough. This should not impact the value calculation and does not have to be evident to anyone except the system operators.

The output of a computing environment, once verified, can be erased, as the data is forwarded to the next computing environment. In other words, it may advisable to separate two consecutive computing environments so that one cannot take the end of the second and learn something about the first.

Exposure: Nominally the system administrator is supposed to have the full set of Turing Machines in his environment as well as the details of the Shadow Turing Machines. However one can deem an architecture where the individual Turing Machines keep the corresponding Shadow machines private. An examiner, holding the output of the computing environment will inquire the last Turing Machine about its Shadow operation, and reverse the output with this information. Then the administrator will inquire about the shadow machine of the previous Turing Machine, and further reverse shadow. This reversal may continue towards the input to the computing environment to verify that the computing path is bona fide. This architecture will allow individual Turing machines to change the shadow machine as often as they please.

V. SUMMARY

We presented a host of applications emerging from the simple idea that bit identities, rather than carry the primary data, will be used for tracking, security, and general forensics.

VI. REFERENCE

1. B. Edwards, "Hype and heavy tails: A closer look at data breaches" Journal of Cybersecurity, Volume 2, Issue 1, 1 December 2016, Pages 3–14,
2. Burger, "Long Term Implications of Data Breaches" Jr. of Information Privacy and Security Vol 13, 2017 issue 4
3. L. Cheng "Enterprise data breach: causes, challenges, prevention, and future directions"Wiley Online
4. F. J. Kongso "Best Practices to Minimize Data Security Breaches for Increased Business Performance." Walden University Scholar Works.