The Tiled Bitmap Forensic Analysis Algorithm

Kyriacos E. Pavlou and Richard T. Snodgrass, *Senior Member, IEEE*

***Abstract*— Tampering of a database can be detected through the use of cryptographically-strong hash functions. Subsequently-applied *forensic analysis* algorithms can help determine when, what, and perhaps ultimately who and why. This paper presents a novel forensic analysis algorithm, the *Tiled Bitmap Algorithm*, which is more efficient than prior forensic analysis algorithms. It introduces the notion of a *candidate set* (all possible locations of detected tampering(s)) and provides a complete characterization of the candidate set and its cardinality. An optimal algorithm for computing the candidate set is also presented. Finally, the implementation of the Tiled Bitmap Algorithm is discussed, along with a comparison to other forensic algorithms in terms of space/time complexity and cost. An example of candidate set generation and proofs of the theorems and lemmata and of algorithm correctness can be found in the appendix.**

***Index Terms*— Database Management, Security, integrity, and protection, Temporal databases.**

I. Motivation

Widespread news coverage of collusion between auditors and the companies they audit (e.g., Enron, WorldCom) helped accelerate recent passage of federal laws (e.g., Health Insurance Portability and Accountability Act: HIPAA [12], Sarbanes-Oxley Act [13]) that mandate better controls on electronic data. *Compliant records* are those required by myriad laws and regulations to follow certain "processes by which they are created, stored, accessed, maintained, and retained" [2].

We previously proposed an innovative approach in which cryptographically strong one-way hash functions allow the detection of a *corruption event* (CE), which is any event that corrupts the data and compromises the database. The corruption event could be due to an adversary, including an auditor or an employee or even an unknown bug in the software (be it the DBMS or the file system or somewhere in the operating system), or a hardware failure, either in the processor or on the disk [10]. Tamper detection is accomplished by hashing data manipulated by transactions and periodically *validating* the audit log database to detect when it has been altered. Validation involves sending the hash value computed over all the database to an external *notarization service*, which will indicate whether that value matches one previously computed. Should tampering have occurred, the two hash values will not match.

K. Pavlou and R. Snodgrass are with the University of Arizona.

At this point, all that is known is that at some time in the past, data somewhere in the database has been tampered. *Forensic analysis* is needed to ascertain *when* the tampering occurred, and *what* data was altered. Knowing the "when" and "what" can give indirect clues to the CIO and CSO that would perhaps allow them to ultimately determine who the adversary is and why the corruption was done. The identification of the adversary is not explicitly dealt with.

Validation provides a single bit of information: has the database been tampered with? To provide more information about when and what, we hash the data of various sequences of transactions during validation. The database transactions are hashed in commit order creating a *hash chain*. Then, during forensic analysis of a subsequent validation that detected tampering, those chains can be rehashed to provide a sequence of truth values (success or failure), which can be used to narrow down "what."

We have elsewhere [7] proposed the Monochromatic, RGB, and Polychromatic forensic analysis algorithms. These algorithms differ in the amount of work necessary during normal processing (computing additional hash chains during periodic validation) and the precision of the when and what estimates produced by forensic analysis. Here we introduce a more efficient algorithm, the Tiled Bitmap Algorithm.

We first present the threat model, then the Tiled Bitmap Algorithm by way of an example. This algorithm requires what we term as the *candidate set*. We then consider the more general problem of characterizing the candidate set, which can be utilized to produce two approaches for computing

that set. This is followed by an evaluation of the implemented algorithm. We end with a discussion of previous work and a summary.

II. Parties Involved and Threat Model

In this section we introduce the parties involved and the underlying threat model.

The parties involved are:

- The DBMS
- An external digital notarization service. This is a company which can digitally notarize documents and then validate their correctness.
- The validator. This is a DBMS application which periodically contacts the digital notarization service.
- The forensic analyzer. This is a DBMS application responsible for executing the chosen forensic analysis algorithm.

Few assumptions are made about the threat model. The system is assumed to be secure until an adversary gets access, at which point he has access to everything: the DBMS, the operating system, the hardware, and the data in the database. We still assume that the notarization and validation services remain in a trusted computing base. This can be done by making them geographically and perhaps organizationally separate from the DBMS and the database [5], thereby effecting correct tamper detection even when the tampering is done by highly-motivated insiders. (A recent FBI study indicates almost half of attacks were by insiders [1].) To prevent spoofing between the DBMS and the validator, it is possible to use a combination of Trusted Platform Modules (TPMs), mutual authentication, and a secure communication channel. The specifics of this scheme are beyond the scope of this paper.

The basic mechanism described in the next section provides correct tamper detection. If an adversary modifies even a single byte of the data or its timestamp, the independent validator will detect a mismatch with the notarized document, thereby detecting the tampering. The adversary could simply re-execute the transactions, making whatever changes he wanted, and then replace the original database with his altered one. However, the notarized documents would not match in time. Avoiding tamper detection comes down to inverting the cryptographically-strong one-way hash function. An extensive presentation of the approach, performance limitations, tamper detection, threat model and other forensic analysis algorithms can be found elsewhere [8], [10].

III. An Example

Consider a database recording when privacy release authorizations were signed by a patient (in the US all patients are now required by HIPAA [12] to sign such authorizations). For ease of discussion we'll use a granularity of an hour. Dr. Dan inadvertently revealed confidential health information to an insurance company on hour 30, shortly *before* patient Pam actually signed the authorization (on hour 31). Dr. Dan later realized his mistake, which is an offense under HIPAA and can have significant legal implications. So on hour 51 Dr. Dan colludes with his friend the database administrator to alter the database to back-date that authorization from hour 31 to hour 28. The database now implies that authorization had been received on hour 28, just before the confidential information was transferred on hour 30: everything looks fine.

In order to ensure HIPAA compliance, the health care company that Dr. Dan works for uses a database management system incorporating tamper detection and forensic analysis. Each transaction is hashed when it is committed and linked to the previous transaction. Every 16 hours the system runs the validator, which rehashes all the transactions and compares the value with the previously-notarized and stored hash value. The time interval between two successive validations is termed the *validation interval*, or I_V (see Table I). The validator also computes partial chains that will later be useful in forensic analysis. Specifically, it computes the five hash chains shown in Figure 1, hash chain c_0 through hash chain c_4, over the previous 16 hours, storing five hash values in a secure database available only to an external digital notarization service. Each 16-hour collection of partial hash chains is termed a *tile*.

When the validator runs at hour 64, it detects the tampering. The forensic analysis algorithm springs into action. It first reports to the compliance service that the database was tampered sometime within the last sixteen hours, between hour 49 and hour 64. That helps bound the "when" of the tampering. The algorithm then recomputes some of the partial hash chains on the tampered data and sends the new hash values to the notarization service, which responds

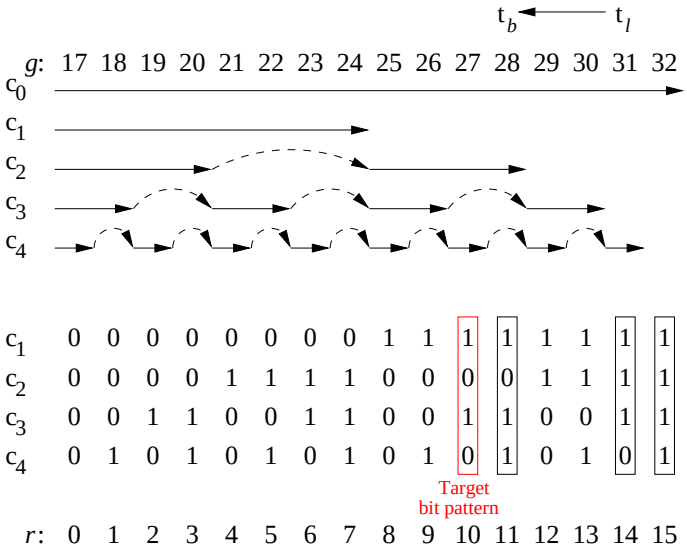

Fig. 1. The hash chains of a single (second) tile. t_l is the actual time of authorization, while t_b is the backdated authorization time. The rectangles mark the elements of the candidate set.

with "success" (the old and new values match) or "failure," for each hash chain queried.

Specifically, the algorithm linearly scans all the tiles in the database to identify in which tile(s) the tampering occurred. The "success" and "failure" response of the c_0 chain of each tile will, in this case, narrow down the tampering to the tile covering hours 17 to 32. Note that each tile that includes a tampering can be independently analyzed, and a corruption across tiles, say changing a timestamp from hour 31 (in the second tile) to hour 7 (in the first tile) can be analyzed by examining each tile independently. In Dr. Dan's case, the validation of the c_0 chain will report a "failure," for only the 17–32 tile. This tells the algorithm that the "what" of the tampering was data stored between hours 17 and 32, a sixteen-hour period. However, we would like to narrow down the tampering to a much finer granularity: that of a single hour, or at least down to a few hours. (It turns out that depending on when the corruption occurred, sometimes we can do very well and sometimes less well.)

In Figure 1, Dr. Dan's back-dating of the authorization from hour 31 to hour 28 is shown at the top, as a left-pointing arrow. The first hash chain, c_0, is also shown at the top.

The algorithm now recomputes the other four partial hash chains for this tile, c_1 through c_4. Four partial hash chains are used to get down to an hour granularity, given that each tile is 16 hours, which is the validation interval I_V.

The finest spatial granularity of the corrupted data would be an explicit attribute of a tuple or a particular timestamp attribute. However, this proves to be costly and hence we define R_s, the finest granularity chosen to express the uncertainty of the spatial bounds of a corruption event. R_s is called the *spatial detection resolution*.

The database administrator specifies both I_V and R_s, in this case, 16 hours and 1 hour, respectively. An R_s of 1 hour implies four other chains, excluding c_0, are needed in a tile (since $\lg(I_V) = \log_2(16) = 4$). If we wanted a finer granularity of, say, 15 minutes ($1/4$ hour), we would need an additional two chains (i.e., $\lg(16 \div \frac{1}{4}) = \lg(64) = 6$).

Hash chain c_1 covers the first eight hours of the tile. Hash chain c_2 covers the first four hours, then skips four hours, then covers hours 8 through 11. Similarly, c_3 covers four two-hour periods, with embedded skips, and c_4 covers every other hour. (Hash chain linking is discussed in more detail elsewhere [10].)

Changing the timestamp on an authorization is equivalent to removing that authorization from all hash chains that cover the original time and adding that authorization to all hash chains that cover the inserted time. Examination of Figure 1 will explain why hour 28, in which the authorization was added by Dr. Dan, appears in hash chains c_0 and c_2. Hour 31, from which the authorization was removed by Dr. Dan, appears in hash chains c_0 and c_4. Hence, c_1 and c_3 report "success" and c_0, c_2, and c_4 report "fail."

We can assemble the success and failure results for the four hash chains c_1 through c_4 into a 4-bit binary number, with failure denoted with "0" and success with "1". The number that results from this particular back-dating from hour 31 to hour 28 is 1010. We term this value the *target binary number*, or *target*. The target is the input to the forensic analysis. Our task is to take this binary number, the target, and figure out what could have happened.

The truth values shown at the bottom of the figure indicate the target string that would result had the corruption event tampered with data stored at the indicated hour. For example, changing the data of a tuple that was originally stored in the first hour of this interval would have rendered all of the chains as failure, resulting in a value of 0000.

Recall that our corruption event occurred at hour 51, changing a timestamp from 31 to 28,

TABLE I
SUMMARY OF NOTATION USED.

Symbol	Name	Definition
CE	Corruption event	An event that compromises the database
I_V	Validation interval	The time between two successive validations
R_s	Spatial detection resolution	Finest granularity of a CE's spatial bounds uncertainty
τ_{FVF}	Time of first validation failure	Time instant at which the CE is first detected

the hash chains provide a target of 1010. What could such a target indicate? It could indicate the corruption of data during a single hour, or any combination of timestamp and/or data during two or more separate hours such that the resulting target after validation is equal to 1010. For example, one possibility is that only the data in hour 27 ($r = 10$) was modified. Another is that the timestamp was moved from 31 ($r = 14 = 1110_2$) to 28 ($r = 11 = 1011_2$), again yielding a target 1010. This is in fact what happened. A different possibility is that the data in hours 28 and 31 were corrupted independently. A fourth possibility is that the time was moved from 28 to 31. Other possibilities are a change from hour 27 to 31, a change from hour 27 to hour 32, a change from hour 27 to hour 28, or a change in the other direction. All these possibilities and many others result in a target of 1010. Precisely because this list of possibilities can get quite long, we introduce in the next section the notion of candidate set which retains comprehensiveness but is a lot simpler.

There are two special cases worth discussing separately. If target bit pattern is 1111 then it is not the case that no corruption has happened. To begin with, we are *certain* that there is corruption in this tile because c_0 reported "failure." The only thing that pattern 1111 implies is that no corruption has occurred in the time granules covered by hash chains c_1, c_2, c_3, c_4. The only granule *not* covered by these last four hash chains is the last granule (15) so we can conclude (by eliminating all other possibilities) that the corruption must have been located in time granule 15. This is the only case we are certain that there are no false positives.

On the other hand, if the target bit pattern is 0000 then a corruption event can be anywhere in the tile's 16 time granules. Even though this does not affect correctness, this is the worst case scenario where we could potentially have the maximum number of false positives.

```
// input: τ_FVF is the time of first validation failure
//        I_V is the validation interval
//        k is used for the creation of C_{t,k}
//        R_s is the spatial detection resolution
// output: C_set, an array of binary numbers
function Tiled_Bitmap(τ_FVF, I_V, k, R_s)
1:  t ← 0        // the target
2:  C_set ← C_temp ← ∅
3:  τ ← 1
4:  while τ < τ_FVF do
5:    if ¬ val_check(c_0(τ)) then
6:      n ← lg(I_V/R_s)
7:      for i ← n to 1
8:        t ← t + 2^{n−i}·val_check(c_i(τ))
9:      C_temp ← candidateSet(t, n, k)
10:     for each r ∈ C_temp
11:       g ← renumber(r, τ, R_s)
12:       C_set ← C_set ∪ {g}
13:   τ ← τ + I_V
14: return C_set
```

Fig. 2. The Tiled Bitmap Algorithm.

IV. THE TILED BITMAP ALGORITHM

We formally define the problem as follows.
Problem Definition: The task is to compute from a single target all the possible corruption events, which we term the candidate set.

For the example in the previous section, the candidate set would comprise the hours $\{27, 28, 31, 32\}$. We now present an algorithm to do this.

In the algorithm shown in Figure 2, I_V is the number of hours between validations (in the example, $I_V = 16$). We use a helper function called val_check. This function takes a hash chain as a parameter and returns the boolean result of the validation of that chain.

The partial hash chains within a tile are denoted by $c_0(\tau), c_1(\tau), \ldots, c_{\lg(I_V)}(\tau)$, with $c_i(\tau)$ denoting the i^{th} hash chain of the tile which starts at time instant τ. On line 4 the algorithm iterates through the different tiles and checks if the longest partial chain $c_0(\tau)$ evaluates to FALSE. If not, it moves on to the next tile. If the chain evaluates to FALSE (line 5), the algorithm iterates through the rest of the partial chains in the tile (line 7) and "concatenates" the result of each validation to form the target

number (line 8). Then the candidateSet function is called (line 9) to compute all the candidate set elements from the target number according to the user-specified parameter k discussed in the next section. On lines 10–12 the candidate granules are renumbered to reflect their global position. The function renumber( ) on line 11 uses R_s to find the global position of r, computing g as a single granule, or group of granules if $R_s > 1$. Once the C_{set} is reported the CSO can exactly pinpoint the corrupted tuples and can thus weed out the false positives. In order to achieve this he must compare the data stored in the backup tapes with the data contained in the granules.

We now state the running time of the Tiled Bitmap Algorithm. Let D be the number of granules (hours) before the first validation failure (for the example, $D = 64$). The "while" loop on line 4 takes $\lceil D/I_V \rceil$ in the worst case. In reality, because of the "if" statement on line 5 the body of the loop gets executed only if corruption is initially detected by using $c_0(\tau)$. Hence, the actual number of times the loop is executed is $\Theta(F)$ where F is the number of times the validation of a $c_0(\tau)$ chain fails. The "for" loop on line 7 takes $\lg(I_V/R_s)$ while the candidateSet function takes $\Omega(\lg(I_V/R_s) + 2^{z(t)})$, where $z(t)$ is the number of zeros in the target binary number t (see Section V). The loop on line 10 takes $\Theta(2^{z(t)})$. Hence the runtime of this algorithm is

$$\begin{aligned}\Omega(F \cdot (\lg(I_V/R_s) + (\lg(I_V/R_s) + 2^{z(t)}) + 2^{z(t)})) \\ = \Omega(F \cdot (\lg(I_V/R_s) + 2^{z(t)})) \\ = O((D/I_V) \cdot (\lg(I_V/R_s) + 2^{z(t)})) \\ = O((D \cdot \lg(I_V/R_s))/I_V + D)\end{aligned}$$

given that in the worst case F takes the value (D/I_V), which is the total number of tiles, and $2^{z(t)}$ takes the value I_V, which the total number of granules in a tile.

There is one important aspect left unaddressed in the above algorithm: the candidateSet function. But before we can present this latter algorithm, we must formally characterize the candidate set.

V. Characterizing the Candidate Set

In the forensic analysis context, the parameter k passed to the algorithm represents the *actual* number of granules corrupted. In the example shown in Figure 1, $k = 2$. However, we usually have no knowledge of the value of k. What we have is only the *target* from which we have to find the possible bit patterns (each generated by the validation of the chains in the tile assuming that corruption *occurred by itself*) that when bitwise *AND*ed produce the *target*. The reason for requiring that the different bit patterns produce the *target* when *AND*ed is because this is effectively what happens when the corruptions occur *simultaneously* within a tile. This arises from the mechanics of forensic analysis. Specifically, each corruption event renders some of the chains as "failing." A chain will succeed in the end only if it succeeds in every one of the corruption events. So in the example, chains c_1 and c_3 succeed, but chains c_2 and c_4 fail. The set of all such bit patterns which could produce the *target* when *AND*ed is termed the *candidate set*. In order to be more rigorous in our analysis we proceed to give a formal characterization of the candidate set.

We define the length l of a binary number b, denoted by $|b| = l$, as the number of its digits. From this point forward we consider l to be fixed. candidateSet essentially "sums up" the pre-images of all the binary numbers of length l, $\mathbb{B} = \{b : |b| = l\}$, under a family of bitwise *AND* functions whose domain is a finite Cartesian product.

$$AND_k : \mathbb{B}^k \longrightarrow \mathbb{B}$$

$$AND_k((b_1, b_2, \ldots, b_k)) = b_1 \wedge b_2 \wedge \ldots \wedge b_k$$

Observe that the maximum number k of sets participating in the Cartesian product is 2^l (i.e., every granule in the tile is corrupted), since if k is allowed to take a value beyond that, it will force a repetition of one of the binary numbers. For forensic analysis purposes this implies that the same granule has been corrupted more than once. This is not informative or useful in any way since repeated *AND*ing operations with the *same* binary number leave the result invariant (the operation is *idempotent*). This is also compatible with forensic analysis since we only care if a granule is corrupted or not—if we wanted to know more we would need to increase the resolution by choosing a smaller granule size (i.e., smaller R_s). In other words, repetition is not allowed and hence for a given k-tuple all its components are distinct. Also note that the value of k uniquely identifies a specific AND_k function in the above family.

We formally define the set of all binary numbers which appear as components in at least one of the pre-images (i.e., k-tuples) of a specific target binary number t the *candidate set*:

$$C_{t,k} = \{b \in \mathbb{B} : \exists\, b_1, b_2, \ldots, b_{k-1} \in \mathbb{B} \text{ s.t. } \mathit{AND}_k((b, b_1, \ldots, b_{k-1})) = t\} .$$

The $\wedge$ operation is commutative: the order of the operands does not matter, and that is why this can be defined more simply as a set of booleans rather than as a set of k-tuples of booleans. The word "candidate" was used to name this set because in forensic analysis, its elements correspond bijectively to the granules (in the example, the hours indicated in Figure 1), which are candidates where corruption may potentially have occurred. In Dr. Dan's case, the candidate set would be the hours 27, 28, 31, and 32 that is, $r = 10$, $r = 11$, $r = 14$, and $r = 15$.

Observe that, it is not the case that $|C_{t,k}| = k$, i.e., k is *not* the cardinality of the candidate set. The cardinality in the example is 4: in this case the algorithm can narrow down the possibilities only to four granules, two actual ones ($k = 2$) and two false positives. The candidate set will comprise *all possible* binary numbers that could produce the target bit pattern, and not just the granules corrupted in a specific case. Hence, the candidate set will always include the actual k granules that were corrupted together with other potentially corrupted granules. This ensures correctness but allows for the existence of false positives.

For convenience we can express these sets in decimal, though our algorithms read and write in binary. For example: $C_{1010,1} = \{1010\} = \{10\}$, $C_{1010,2} = \{1010, 1011, 1110, 1111\} = \{10, 11, 14, 15\}$. 1001 is not in $C_{1010,2}$ because 1001 cannot be in the pre-image of 1010. Note that even though two binary target strings may have the same numerical value, if their length is different then their candidate sets will be different. For example, the candidate set $C_{000,2}$ is different from $C_{0000,2}$.

Let $z(t)$ be the number of zeros in the binary number t, e.g., $z(1010) = 2$. By definition $1 \leq k \leq 2^l$ and $0 \leq z \leq l$. The behavior of $C_{t,k}$ is interesting: as k increases the candidate set for a fixed t remains invariant and equal to the candidate set for $k = 2$, until some threshold value $2^{z(t)}$ after which it becomes empty. Simply put, $C_{t,k}$ obeys an all-or-none law.

Lemma 1:
$C_{t,k} = C_{t,2}$ if $l \geq z(t) > 0$ and $2 \leq k \leq 2^{z(t)}$. In other words, the candidate set remains invariant given that the stated conditions are met.

Proof: Given in Appendix A. □

A complete characterization of the candidate sets is given below.

Theorem 1:

$$C_{t,k} = \begin{cases} \{t\} & , k = 1 \quad (1) \\ \emptyset & , z(t) = 0 \wedge k > 1 \quad (2) \\ C_{t,2} \neq \emptyset & , l \geq z(t) > 0 \wedge 2 \leq k \leq 2^{z(t)} \quad (3) \\ \emptyset & , l \geq z(t) > 0 \wedge k > 2^{z(t)} \quad (4) \end{cases}$$

Proof: Given in Appendix B. □

Theorem 1 is the reason for which we decided to make k user-configurable in the Tiled Bitmap pseudocode. If the CSO by some other means has any indication for the value of k, i.e., the actual number of corruptions occurred, then he can pass that information to the algorithm. If the algorithm returns an empty candidate set then the CSO can deduce that his initial knowledge/guess for the value of k was incorrect. If the CSO has no *a priori* knowledge about the value of k, as is usually the case, then according to Theorem 1 the CSO need only give k the default value of 2 and not worry that any other choice for k would compromise the forensic analysis results.

Corollary 1:

$$|C_{t,k}| = \begin{cases} 1 & , \quad k = 1 \\ 0 & , \quad z(t) = 0 \wedge k > 1 \\ 2^{z(t)} & , \quad l \geq z(t) > 0 \wedge 2 \leq k \leq 2^{z(t)} \\ 0 & , \quad l \geq z(t) > 0 \wedge k > 2^{z(t)} \end{cases}$$

Proof: This follows directly from Theorem 1. □

For example, with our target bit pattern of $t = 1010$, we have $z(t) = 2$ and the *candidate set* is $C_{1010,2} = \{10, 11, 14, 15\}$ with $|C_{1010,2}| = 2^2 = 4$.

We now turn to ways in which the candidate set may be computed. We first give an algorithm that is optimal in time, except for a very few cases. Following some further observations on candidate sets, we show how, given a candidate set, one can calculate other candidate sets with a smaller l in constant time.

VI. Computing the Candidate Set

Figures 3, 4, 5, and 6 present an optimal algorithm for computing the candidate set given the target string t and k, and again assuming a fixed l. Recall that candidateSet is used in the Tiled Bitmap

```
// input:  t is a binary target number
//         l is the length of t
//         k is a function index for AND_k
// output: C_{t,k} is an array of binary numbers
//         (also created is an array of zeros, Z)
1: function candidateSet(unsigned int t, int l, int k)
2:     C_{t,k} ← new array()
3:     z ← 0
4:     Z ← new array()
5:     for i ← l − 1 to 0
6:         if t & (1 << i) = 0 then z ← z + 1
7:         Z[l − i − 1] ← z
8:     if k < 1 ∨ k > 2^l then report NOT_DEFINED
9:     else if k = 1 then C_{t,k} ← {t}
10:    else if (z = 0 ∧ k > 1) ∨ (l ≥ z > 0 ∧ k > 2^z)
11:        then C_{t,k} ← ∅
12:    else if (l ≥ z > 0) ∧ (2 ≤ k ≤ 2^z) then
13:        rightmost ← createRightmost(t, l)
14:        C_{t,k} ← generate(t, rightmost[l], l, C_{t,k})
15:        C_{t,k} ← funkySort(z, C_{t,k})
16:    return C_{t,k}
```

Fig. 3. The candidateSet function.

```
// input:  t is the target bit number
//         l is the length of the bit representation of t
// output: the populated rightmost array
1: function createRightmost(unsigned int t, int l)
2:     int i, j, flag
3:     j ← −1
4:     flag ← FALSE
5:     rightmost ← new array()
6:     for i ← l − 1 to −1
7:         if flag then j ← l − i − 2
8:         if t & (1 << i) = 0 then flag ← TRUE
9:         else flag ← FALSE
10:        rightmost[l − i − 1] ← j
11:    return rightmost
```

Fig. 4. The createRightmost function.

Algorithm. It generates the elements (bit patterns) in the candidate set from the target pattern t preserving bit positions with 1s and creating combinations of patterns of 1s and 0s using the remaining positions having 0s. Finally, it sorts the patterns in ascending order of their numeric values by using an interesting linear-time sort. (An example of the candidate set generation for our target of $t = 1010$ can be found in Appendix G.) All arrays and strings use zero-based indexing. All parameters are passed by value.

Let us now briefly examine this algorithm. We

```
// input:  t is the modified bit pattern at each stage
//           of the recursion
//         p is the position of one of the zeros in t
//         l is the length of the bit representation of t
//         C_{t,k} array in which candidate granules
//           are accumulated
// output: C_{t,k} contains candidate granules (unsorted)
1: function generate(unsigned int t, int p, int l,
                     array C_{t,k})
2:     if p = −1 then C_{t,k}.append(t)
3:     else
4:         C_{t,k} ← generate(t, rightmost[p], l, C_{t,k})
5:         C_{t,k} ← generate(t + (1 << (l − p − 1)),
                     rightmost[p], l, C_{t,k})
6:     return C_{t,k}
```

Fig. 5. The generate function.

```
// input:  z is the number of zeros in t
//         C_{t,k} is the result of generate( )
// output: C_{t,k} sorted in ascending order
1: function funkySort(int z, array C_{t,k})
2:     sorted ← new array()
3:     indices ← new array()
4:     indices[0] ← 0
5:     int i, offset, power
6:     offset ← 0
7:     power ← 1 << z
8:     for i ← 1 to (1 << z) − 1
9:         if (i & (i − 1)) = 0 then
10:            power ← power >> 1
11:            offset ← 0
12:        indices[i] ← indices[offset] + power
13:        offset ← offset + 1
14:    for i ← 0 to (1 << z) − 1
15:        sorted[i] ← C_{t,k}.get(indices[i])
16:    return sorted
```

Fig. 6. The funkySort function.

first start by looking at the candidateSet function (Figure 3) and discuss each different function as we encounter it.

The use of the *Z* array on lines 4 and 7 will be explained later in the discussion following Theorem 2. Lines 8–12 follow the result of Theorem 1. Then on line 13 the createRightmost helper function is called (Figure 4) to preprocess the target binary number t and to fill the *rightmost* array in order to answer the "rightmost zero" query in constant time. More specifically, *rightmost*$[p]$ is the index (bit position) of the rightmost zero to the left of index p

non-inclusive. Within this function i iterates over t from left to right (high-order to low-order bits). The flag is required because we must remember what we saw in the previous iteration: if *flag* = TRUE we saw a 0, otherwise we saw a 1. This runs in $\Theta(l)$.

On line 14 (Figure 3) the generate function (Figure 5) is called. This is a recursive function which creates the candidate set elements. Given a position p, which is a specific index in the zero-based enumeration (left to right) of the binary number t, it finds the index of the rightmost zero to the left of p using the *rightmost* array. It first recurses on that index maintaining the same binary number (line 4) and then sets the digit at position p to 1 and recurses on the same index *rightmost*$[p]$ but with this new number (line 5). We can consider the input target string t as capturing all the $2^{z(t)}$ numbers that must be generated during the recursion, so we can consider the input size to be $n = 2^{z(t)}$. Also, at each recursive call the position of the zero processed is never revisited so the input size at each call is essentially halved. Moreover, the amount of work done at each stage of the recursion is constant hence the formula that captures this recursion is $T(n) = 2T(\frac{n}{2}) + \Theta(1)$. The solution of this formula is $\Theta(n)$ so the running time of the generate function is $\Theta(2^{z(t)})$. However, a side-effect of this recursive creation of the candidate set elements is that the elements are not generated in numeric order.

On line 15 of candidateSet (Figure 3) we call the sorting function. Even though the elements are not sorted there does exist a pattern in the order in which they are created. This funkySort function (Figure 6) creates the sequence of indices which when used to index into the $C_{t,k}$ array will result in the ordering of the candidate set elements. This is achieved by performing a single pass over the *indices* array and creating each new index by manipulating appropriately previous ones (lines 8–13) within the funkySort function.

For example, with a binary target of $t = 10000$, i.e., 16 in decimal, after the generate function finishes the candidate set will be $C_{t,k} = \{16, 24, 20, 28, 18, 26, 22, 30, 17, 25, 21, 29, 19, 27, 23, 31\}$ in this order. Examining closely the set we see that in order to create the *sorted* array we must recursively visit the first element of each subsequent half of $C_{t,k}$. Line 12 creates this sequence of indices: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15. More specifically, by starting from 0 the 8 can be created by $0 + 2^{z(t)-1}$ where $z(t) = 4$. Then, 4 and 12 can be obtained by adding 2^2 to each of 0 and 8. Then 2, 10, 6 and 14 are obtained by adding 2^1 to 0, 8, 4, 12 respectively. Finally, the last 8 numbers are obtained by adding 2^0 to the first 8 numbers. This explains why at elements appearing at indices which are powers of 2 in the *indices* array, the *offset* is reset to zero and the *power* is halved. On lines 14–15 (Figure 6) we use the sequence of indices we created and the actual sorting happens. This pass over the *indices* array runs in $\Theta(2^{z(t)})$.

TABLE II
CANDIDATE SETS FOR TARGETS $|t| = 4$ WITH $k = 2$

Binary Number t	$\lvert C_{t,2}\rvert$	$C_{t,2}$
0000	16	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
0001	8	$\{1, 3, 5, 7, 9, 11, 13, 15\}$
0010	8	$\{2, 3, 6, 7, 10, 11, 14, 15\}$
0011	4	$\{3, 7, 11, 15\}$
0100	8	$\{4, 5, 6, 7, 12, 13, 14, 15\}$
0101	4	$\{5, 7, 13, 15\}$
0110	4	$\{6, 7, 14, 15\}$
0111	2	$\{7, 15\}$
1000	8	$\{8, 9, 10, 11, 12, 13, 14, 15\}$
1001	4	$\{9, 11, 13, 15\}$
1010	4	$\{10, 11, 14, 15\}$
1011	2	$\{11, 15\}$
1100	4	$\{12, 13, 14, 15\}$
1101	2	$\{13, 15\}$
1110	2	$\{14, 15\}$
1111	0	$\emptyset$

The running time of candidateSet is $\Theta(l + 2^{z(t)})$. Thus the algorithm is optimal most of the time, using the lower bound given in Section V, except for the very few cases when $l > 2^{z(t)}$. In terms of space complexity the algorithm given here requires $O(l + 2^{z(t)})$ space.

VII. AN OPTIMAL CANDIDATE SET ALGORITHM, GIVEN A SUMMARY SET

We define the *summary set* as the set of all candidate sets of all binary numbers of length l.

$S_{l,k} = \{C_{t,k} : \forall t \in \mathbb{B} \text{ s.t. } |t| = l\}$

For $l = 4$ and $k = 2$, the last column in Table II provides the elements of $S_{4,2}$.

We now show that for fixed k and given $S_{l,k}$ one can calculate all $S_{l',k}$ s.t. $l' < l$ without resorting to the algorithm given previously. This allows us to find the candidate set for a suffix of y whenever we already have the candidate set for y. The technique

shown below can potentially be faster. We define the *candidate array*, denoted $A_{t,k}$, to be an array which contains the elements of $C_{t,k}$ sorted in ascending numerical value. Then, $A_{t,k}[x:y]$ selects all elements in the candidate array from index x to y. (NB: $A_{t,k}[i] = A_{t,k}[i:i]$). Also, for reasons of ease and precision we annotate A with the length of the binary number whose value was previously implicit, as a leading subscript.

Given a candidate array ${}_{l}A_{y,k}$ for a specific target string y, we wish to compute the candidate array ${}_{l-x}A_{t,k}$ where t is a suffix ($l = |y| > l - x = |t| \geq 1$) of y. Each $S_{l,k}$ captures all the candidate sets for all $l' < l$. This method creates each element of $S_{l',k}$ by exploiting the fact that each of the binary numbers of length l' is a suffix of more than one corresponding binary number of length l . For example, the candidate set $C_{1010,2}$ can be computed from the candidate sets of 01010, 001010, 101010 and so on. Let $y = p \bullet t = \{0,1\}^x t$ for some prefix p of length x. Let $\textit{Suffix}_i(s)$ denote the suffix of string s starting at position i.

Let us look at some examples to develop some intuition. Given ${}_{4}A_{0010,2} = [0010, 0011, 0110, 0111, 1010, 1011, 1110, 1111]$, we wish to compute ${}_{3}A_{010,2}$. Observe that $t = 010$ is $y = 0010$ with the leftmost '0' removed. Removing the leading '0' from y results in a string t which cannot encode any numbers in the range 2^3 to $2^4 - 1$. Thus the candidate array of ${}_{3}A_{010,2}$ will have the same elements as the candidate array of ${}_{4}A_{0110,2}$ except for the numbers encoded by the extra leading digit. We know that each additional '0' present in the target string doubles the cardinality of the candidate set, thus a removal of the zero will halve the number of candidate set elements. Observe also that the elements in the second half of ${}_{4}A_{0010,2}$ have essentially the same bit pattern as the elements in the first half but with a '1' at the leftmost position instead of a '0', e.g., 1110 has the same bit pattern as 0110 apart from the bit in the leftmost position.

Thus in order to compute ${}_{3}A_{010,2}$ we can truncate the leftmost digit from all the elements in the original candidate set. By removing the leftmost digit from each of the elements in ${}_{4}A_{0010,2}$, we get 010, 011, 110, 111, 010, 011, 110, 111. The first half of the elements will have a leading '0' removed, something which will not change their numerical value, while the second half which will have a leading '1' removed will produce identical numbers of length 3 to the truncated numbers in the first half. Since the cardinality of ${}_{3}A_{010,2}$ is half that of ${}_{4}A_{0010,2}$, and since the two halves of ${}_{4}A_{0010,2}$ have the same elements after the truncation and by knowing that ${}_{3}A_{010,2} = [010, 011, 110, 111]$ we can verify that:

$$\begin{aligned} {}_{3}A_{010,2} &= [\textit{Suffix}_1({}_{4}A_{0010,2}[0]), \textit{Suffix}_1({}_{4}A_{0010,2}[1]), \\ &\quad \textit{Suffix}_1({}_{4}A_{0010,2}[2]), \textit{Suffix}_1({}_{4}A_{0010,2}[3])] \\ &= [010, 011, 110, 111] = [2, 3, 6, 7] \end{aligned}$$

Let us consider a different example with the original target string being $y = 1010$ and the same suffix $t = 010$ as before. In this case ${}_{4}A_{1010,2} = [1010, 1011, 1110, 1111]$ all elements necessarily start with a '1'. Since removing the leading '1' from y to get t does not affect the number of zeros in the strings the cardinalities of the two candidate sets is the same. Removing the leftmost '1' from all the elements of ${}_{4}A_{1010,2}$ will yield directly the desired elements of the new candidate set:

$$\begin{aligned} {}_{3}A_{010,2} &= [\textit{Suffix}_1({}_{4}A_{1010,2}[0]), \textit{Suffix}_1({}_{4}A_{1010,2}[1]), \\ &\quad \textit{Suffix}_1({}_{4}A_{1010,2}[2]), \textit{Suffix}_1({}_{4}A_{1010,2}[3])] \\ &= [010, 011, 110, 111] = [2, 3, 6, 7] \end{aligned}$$

With these valuable observations we can now state the theorem.

Theorem 2: Assume $y = p \bullet t = \{0,1\}^x t$, $0 < x < l$, $0 \leq z(t) \leq l - x$ and $q = 2^{z(t)}$. Then:

$${}_{l-x}A_{t,k} = \begin{cases} \text{N/A}, & k > 2^{l-x} & (1) \\ [t] \ , & k = 1 & (2) \\ \emptyset \ , & z(t) = 0 \wedge 1 < k \leq 2^{l-x} & (3) \\ \bigcup_{0 \leq i < q} [\textit{Suffix}_x({}_{l}A_{y,2}[i])], & & \\ & l - x \geq z(t) > 0 \wedge 2 \leq k \leq q & (4) \\ \emptyset \ , & l - x \geq z(t) > 0 \wedge k > q & (5) \end{cases}$$

Proof: Given in Appendix D. □

The strategy for computing the candidate sets using this new method is given below. First we change line 9 of the original Tiled_Bitmap function

9: $C_{temp} \leftarrow$ candidateSet(*target*, n, k)

to

9: $C_{temp} \leftarrow$ candidateSetCached(*target*, n, k)

We introduce a list, *cache*[ ] of records $(y, l_y, k, C_{y,k}, Z)$ which is updated with every call to the function candidateSet. Each such record stores the candidate set $C_{y,k}$ computed by the function, the target number y, the length l_y of y,

```
// input:  t is a binary target number
//         l is the length of t
//         k is a function index for AND_k
// output: C_{t,k}, a candidate set either computed
//                  anew or derived from C_{y,k}
1: function candidateSetCached(unsigned int t,
                               int l, int k)
2: t_start ← −1
3: for i ← 0 to cache.length−1
4:    t_start ← findSuffix(cache[i].y, t)
5:    if t_start ≥ 0 ∧ k = cache[i].k then
6:       return candidateSetSuffix (cache[i].C_{y,k},
                 t_start, k, cache[i].l_y, cache[i].Z)
7: return candidateSet (t, l, k)
```

Fig. 7. The candidateSetCached function.

the parameter k, and the corresponding Z array for y. In order to achieve this, we change the candidateSet function to store the candidate set in the cache, before returning it.

```
15:       C_{t,k} ← funkySort(z, C_{t,k})
16:    append(cache, (t, l_t, k, C_{t,k}, Z))
17:    return C_{t,k}
```

A new function candidateSetCached (Figure 7) checks to see if a pre-computed candidate set which can be used by this new algorithm, already exists.

Note that t_{start} is the index in the original string y where the suffix t starts. The running time of the candidateSetCached function is $O(l \cdot \textit{cache.length})$, which is the worst case running time for executing lines 3 and 4. The candidateSetSuffix function given in Figure 8 provides the algorithm for creating the new candidate set $C_{t,k}$ from a cached candidate set $C_{y,k}$.

Since creating the candidate set for y involves scanning all of y to find the zeros we can at the same time maintain an array which accumulates the number of zeros encountered so far during the scan. This array is the *Z* array which was created in the function candidateSet (lines 4, 7). We can index into this array using the position which suffix t starts in y and thus get the number of zeros in constant time. For example, for $y = 01101010$ and $t = 1010$ given in terms of t_{start} which is the start position of t in y, we can scan y from left to right and create the array $Z = [1, 1, 1, 2, 2, 3, 3, 4]$. This arrays gives the number zeros in every suffix of y. Thus, $z(t) = z(y) - Z[t_{start} - 1]$. In this case $t_{start} = 4$, and so $z(1010) = z(01101010) - Z[4 - 1] = 4 - 2 = 2$.

```
// input:  C_{y,k} is the original candidate set
//         t_start is the bit position at which suffix t
//            starts in y
//         l_y is the length of original target string y
//         Z array for y
// output: the candidate set C_{t,k}
1: function candidateSetSuffix (array C_{y,k},
                                int t_start, int k,
                                int l_y, array Z)
2:    C_{t,k} ← new array()
3:    l_t ← l_y − t_start
4:    z_t ← z_y − Z[t_start − 1]
5:    mask ← (1 << l_t) − 1
6:    y ← C_{y,k}[0]
7:    t ← y & mask
8:    if k < 1 ∨ k > 2^{l_t} then report Not_Defined
9:    else if k = 1 then C_{t,k} ← {t}
10:   else if (z_t = 0 ∧ 1 < k ≤ 2^{l_t} ) ∨
            (l_t ≥ z_t > 0 ∧ k > 2^{z_t}) then C_{t,k} ← ∅
11:   else if (l_t ≥ z_t > 0) ∧ (2 ≤ k ≤ 2^{z_t}) then
12:      for i ← 0 to 2^{z(t)} − 1
13:         C_{t,k}.append(C_{y,k}[i] & mask)
14:   return C_{t,k}
```

Fig. 8. The candidateSetSuffix function.

In addition, the *mask* (Figure 8, lines 7 and 13) is used as a means of setting the first x bits of each original candidate set element to zero, which is the equivalent in a sense of taking the suffix of the corresponding binary string. For example, if the candidate set element is 18, with binary representation 10010, and we want to take the suffix starting at index 2, then the *mask* $= 7$ (00111 in binary). Thus, by bitwise *AND*ing the *mask* and the element, we get $010 = 2$. Note that the masking does not simply set the higher order bits to zero but it truncates the number, i.e., the length actually decreases. This is important because we seek to derive from the candidate set of 10010 the candidate set of 010 and not the set for 00010. The latter is impossible to derive in the way described in this section since $C_{00010,2}$ is a *superset* of $C_{10010,2}$.

The "for" loop on line 12 dominates the running time of the above algorithm. Hence, the algorithm, in the worst case, runs in $\Theta(2^{z(t)})$, which is optimal.

However, we can do better by using a different representation for the candidate set of the suffix t. Since the elements of $C_{t,k}$ are contiguous elements of $C_{y,k}$ starting at position 0 then the candidate set of t can be given as a range of values. This is achieved

just by maintaining a pointer to the position $q-1$ in the candidate array of y marking the last element of $C_{t,k}$. Thus, only two numbers *mask*, and $q = 2^{z(t)}$, both of which can be computed in constant time, are needed to capture the candidate set of any suffix of target y. To create the *mask* we use l_t (as seen on line 5) which was computed from the input integer t_{start} on line 3. Obtaining q is easy since we have already computed $z(t)$ on line 4. Thus, the first and last elements of the candidate set for the suffix can be given as $C_{y,k}[0]\,\&\,\mathit{mask}$ and $C_{y,k}[q-1]\,\&\,\mathit{mask}$ respectively. This approach avoids the expensive "for" loop on line 12 and makes the algorithm run in $\Theta(1)$.

It is preferable to use the candidateSetSuffix Algorithm in one particular situation: to find the candidate set for the suffix of y whenever we already have the candidate set for y. Consider the following examples. For $l = 4$ we want to calculate $C_{1010,7}$ and $C_{010,3}$. $C_{1010,7} = \emptyset$ since $|C_{1010,7}| = 2^2 = 4 < k = 7$. In the case of $C_{010,3}$ we have $3 \geq z(t) = 2 > 0$ and $2^{z(t)} = 4 > k = 3$ so

$$\begin{aligned} {}_3A_{010,3} &= \bigcup_{1\leq i\leq 4} [\mathit{Suffix}_1({}_4A_{1010,2}[i])] \\ &= \bigcup_{1\leq i\leq 4} [\mathit{Suffix}_1[1010, 1011, 1110, 1111]] \\ &= [010, 011, 110, 111] \end{aligned}$$

and thus $C_{010,3} = \{2, 3, 6, 7\}$. If we decide to use the faster constant running time approach the result will be given as $\mathit{mask} = 0111$ and $q = 2^2 = 4$ and hence the first element in $C_{010,3}$ is ${}_4A_{1010,2}[0]\,\&\,0111 = 1010\,\&\,0111 = 010 = 2$ while the last element is ${}_4A_{1010,2}[4-1]\,\&\,0111 = 1111\,\&\,0111 = 111 = 7$.

Assume that we are auditing a variety of databases, each with a particular l value (for the example in this paper, $l = 4$). Within the forensic analyzer, we could pre-compute a summary set for l_{max}, which is the maximum of the l values that were specified for the databases that were being audited. During forensic analysis of a specific database corruption, given the resulting target string and the l value for this particular database (with $l \leq l_{max}$), this algorithm could calculate in constant time the candidate set, which consists of all the possible corrupted granules that could have yielded that target number for that value of l.

VIII. Implementation and Evaluation

Elsewhere we have introduced the Monochromatic, the RGB, and Polychromatic Algorithms [7]. All algorithms employ the same approach of tamper detection and forensic analysis by hashing transaction data and periodically validating the resulting hash chains. The main differences between the algorithms lie in the number of hash chains used and their structure. The simplest is the Monochromatic Algorithm, which sequentially hashes all data to create a hash chain that incrementally grows over the data of the entire database. The *cumulative nature* of this chain has two consequences. First, it limits the detection of corruption to a single event since periodic validations will yield a sequence of "success" results followed by a sequence of "failure" results. The interface in the transition between these two sequences marks the site of the first (oldest) corruption. Second, the cumulative nature of the chain enables a binary search on the sequence of "successes" and "failures" to locate the transition very quickly.

The RGB Algorithm augments the Monochromatic Algorithm by periodically superimposing (non-cumulative) partial hash chains over the entire database. The name of the algorithm is derived from the color-coding of the different partial hash chains. In this case, the cumulative chain can be used to perform binary search to quickly locate the oldest corruption and then switch to using the "colored" partial chains to explore the rest (more recent part) of the database. This algorithm can detect up to two corruption events.

The Polychromatic Algorithm retains the main Red, Green, and Blue partial chains of RGB and introduces more Red and Blue chains to create groups of chains similar to a tile. This has the advantage that it can arbitrarily shrink the spatial detection resolution by introducing a *logarithmic* number of hash chains as opposed to a linear number needed in RGB. The Polychromatic Algorithm, as with the RGB Algorithm, can only detect only up to two corruption events but could potentially be modified to handle multiple corruptions.

The Tiled Bitmap Algorithm introduced here can be thought of as a refinement/replacement of the Polychromatic Algorithm. The new algorithm can use the cumulative chain of the Monochromatic Algorithm (not elaborated on here). It extends the

TABLE III
RUNNING TIME COMPLEXITY OF FORENSIC ANALYSIS ALGORITHMS

Algorithm	Running Time ($R_s = 1$)
Monochromatic	$O(\lg(D/I_V))$
RGB	$O(D/I_V)$
Tiled Bitmap	$O((D \cdot \lg I_V)/I_V + D)$

TABLE IV
WORST-CASE COST/SPACE COMPLEXITY OF FORENSIC ANALYSIS ALGORITHMS

Algorithm	Cost ($R_s = 1$)
Monochromatic	$O(D)$
RGB	$O(D)$
Tiled Bitmap	$O(D \cdot (1 + \lg I_V)/I_V)$

idea of the RGB Algorithm of using partial chains, and it refines the groups of hash chains of the Polychromatic Algorithm.

The advantage of the Tiled-Bitmap Algorithm is that it lays down a *regular pattern* (a "tile") of such chains over contiguous segments of the database. What is more, it inherits all the advantages of the Polychromatic Algorithm: the chains in the tile form a bitmap which can be used for easy identification of the corruption region, and a logarithmic number of chains can be used to reduce R_s.

The other advantage of the Tiled Bitmap Algorithm is that can detect multiple corruption events (up to D of them, i.e., all granules were corrupted) something that the Monochromatic, RGB, and Polychromatic Algorithms cannot. On the other hand it suffers from false positives while the previous three algorithms do not. (More information on the rate of false positives of the Tiled Bitmap Algorithm can be found elsewhere [8].) Table III shows the running time for three of the forensic analysis algorithms (the Polychromatic Algorithm is omitted because it is replaced by the Tiled Bitmap Algorithm). We assume that the spatial detection resolution R_s is equal to 1 for simplicity. Observe that the algorithms become progressively slower because of the increased number of chains maintained and used during forensic analysis. The Monochromatic Algorithm, while being the fastest algorithm, suffers from the fact that only the first corruption event can be detected. As noted the Tiled Bitmap Algorithm can be slightly optimized by retaining the cumulative chain of the Monochromatic in order to locate the first corrupted tile by performing binary search, although this refinement does not affect its asymptotic running time.

Recall that all algorithms rely on an external notarization service in order to validate the audit log. However, each such contact costs real money. Hence, we quantify the cost of the algorithms

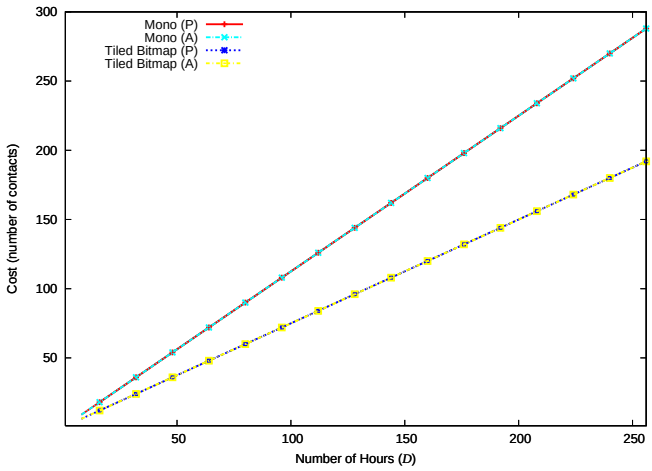

Fig. 9. The cost of the Monochromatic and Tiled Bitmap Algorithms.

as the number of contacts with the notarization service during a specific duration of the normal operation of the system, i.e., whenever a part of the database is notarized or validated. The units of the cost are therefore number of notarizations plus number of validations. We chose to deal with only notarizations and validations occurring before corruption or forensic analysis, because otherwise the cost would be dependent on the number of corruptions. This would render the comparison unfair since the Monochromatic and RGB Algorithms can only detect a limited number of corruptions. More information on the mathematical formulation of the cost can be found elsewhere [8]. It is desirable to minimize this cost for each algorithm while trying to extract as much information possible. Table IV shows the cost for each of the forensic algorithms assuming a spatial detection resolution of one hour ($R_s = 1$) and a single corruption event. In this case we observe the opposite trend compared to the one observed for the running times of the algorithms. For a sufficiently large validation interval I_V the Tiled Bitmap Algorithm has the smallest cost. This is because the ratio $(1 + \lg I_V)/I_V$ becomes less than one.

This quantification of cost also reflects the space

complexity of the forensic algorithms since each of the contacts with the external notarization service corresponds to a hash value (of chains) which must be initially computed (and re-computed for comparison during validation) and maintained within the system. None of algorithms in Table IV require extra space over the collection of hash values themselves.

A 1250-line C code implementation is available at `http://www.cs.arizona.edu/projects/tau/tbdb/`. The code implements several forensic analysis algorithms, including the candidateSet and candidateSetSuffix construction algorithms, the Tiled Bitmap Algorithm, and the Monochromatic Algorithm. This C code implementation uses the more efficient pass-by-reference for arrays and strings compared to the pseudocode given in Section VI. All algorithms were tested extensively and their theoretical costs were experimentally validated. Appendix F provides proofs of correctness for all functions introduced in this paper. We also have developed several graphical user interfaces which include a convenient visual representation of the spatial and temporal extent(s) of detected corruption(s).

Figure 9 shows the results of the experimental cost validation for the Monochromatic and Tiled Bitmap Algorithms (the RGB Algorithm has not been implemented). The experiments used the following setup: $D = 1$ to 256, $R_s = 1$, and $I_V = 8$. Rather than using the cost formulas in order notation (as given in Table IV) to create the graphs, we used more involved (and more accurate) cost functions derived for each algorithm. Note that the cost plot shows both the predicted forensic cost (denoted by "(P)" in the plot legend) and actual cost values (denoted by "(A)" in the plot legend). The actual cost values were computed by inserting appropriate counters in the C code implementation for the Monochromatic and Tiled Bitmap algorithms. The different types of symbols on the curves were added for clarity and correspond to a subset of the actual data points. As can be seen in Figure 9 the predicted and actual cost for the two algorithms are essentially identical. A more detail explanation of the derived costs, and experimental comparisons between algorithms can be found elsewhere [8].

IX. Previous Work

There has been a great deal of work on records management, and indeed, an entire industry has arisen to provide solutions for these needs, motivated recently by Sarbanes-Oxley [13] and other laws requiring compliant record storage. In this context, a "record" is a version of a document. These systems utilize magnetic disks (as well as tape and optical drives) to provide WORM storage of compliant records. We wish to extend the concept of compliant records to tuples of a table stored in a database management system.

Computer forensics is now an active field, with over fifty books published in the last ten years. However, these books are generally about preparing admissible evidence for a court case, through discovery, duplication, and preservation of digital evidence. There are few computer tools for these tasks, in part due to the heterogeneity of the data. One substantive example of how computer tools can be used for forensic analysis is Mena's book [6].

Goodrich et al. introduce new techniques for using main-memory indexing structures for data forensics [3]. They encode authentication information in the way a data structure is organized (not in the stored values) so that alterations can be detected. Their techniques are based on a new reduced-randomness construction for nonadaptive combinatorial group testing, using message authentication codes (MAC) built using cryptographically strong, one-way hash functions. In the database context, we have introduced in previous papers the approach of using cryptographic hash functions to detect database tampering [10] and of introducing additional hash chains to improve forensic analysis [7]. To the best of our knowledge there are no other competing forensic analysis algorithms for high-performance databases.

Strachey has considered table lookup to increase the efficiency of bitwise operations [11]. He provides a logarithmic time/logarithmic space algorithm for reversing the bits in a word. Our second algorithm requires only constant time, but the table must be of exponential space.

Enumerating all solutions (pre-images) is a key step in formal verification. Sheng and others have developed efficient pre-image computation algorithms [4], [9]. These algorithms are similar to the ones introduced in this paper in that they all enumerate all possible solutions. The formal verification algorithms differ in that they are computing pre-images of a state transition network, rather than of bitwise *AND* functions, as in our paper.

X. Summary

Forensic analysis commences when a crime has been detected, in this case the tampering of a database. Such analysis endeavors to ascertain when the tampering occurred, and what data was altered.

Elsewhere we proposed several forensic analysis algorithms [7]. The present paper expands upon that work by presenting the *Tiled Bitmap Algorithm*, which is cheaper and more powerful than prior algorithms. This algorithm employs a logarithmic number of hash chains within each tile to narrow down the *when* and *what*.

Checking the hash chain values produces a binary number; it is the task of the algorithm to compute the pre-image of bitwise *AND* functions of that number. This produces a *candidate set* which identifies all the potentially corrupted granules. We showed that the running time of the algorithm is linear in the length of time the database has been in existence and linear in the size of the computed candidate set. We also note that previous algorithms do not handle multiple corruption events well, whereas the Tiled Bitmap Algorithm can independently analyze corruption events occurring both in different tiles and multiple corruption events occurring within a single tile. However, the Tiled Bitmap Algorithm suffers from false positive results while prior algorithms (Monochromatic, RGB, Polychromatic) do not.

In the later parts of the paper we analyzed completely the behavior of the candidate sets and developed an optimal algorithm to produce these candidate sets. We then introduced a constant-time algorithm which is preferable in the case when the target binary number is a suffix of another binary number for which a candidate set already exists. Finally, we compared prior forensic algorithms with the Tiled Bitmap Algorithm, providing a thorough space and time complexity analysis. We discussed the implementation of the algorithms and experimentally validated their cost. The Tiled Bitmap Algorithm uses additional chains (which incur a logarithmic runtime factor) to detect multiple corruption events, while requiring fewer requests of an external notarization server.

The ultimate goal is an algorithm that retains the logarithmic performance (of the additional chains) of the Tiled Bitmap Algorithm while further simplifying the analysis within a tile, furthering narrowing the bounds on when the tampering occurred, and providing additional forensic information, such as the *direction* of the tampering, i.e., whether the information was back-dated or post-dated.

Acknowledgments

This research was supported in part by NSF grants IIS-0415101, IIS-0639106, and EIA-0080123 and with partial support from a grant from Microsoft Corporation. The reviewers were very helpful in improving the presentation.

References

[1] CSI/FBI, "Tenth Annual Computer Crime and Security Survey," July 2005, `http://www.cpppe.umd.edu/Bookstore/Documents/2005CSISurvey.pdf` (accessed April 16, 2009).

[2] P. A. Gerr, B. Babineau, and P. C. Gordon, "Compliance: the effect on information management and the storage industry," Enterprise Storage Group Technical Report, May 2003, `http://www.enterprisestrategygroup.com/ESGPublications/ReportDetail.asp?ReportID=201` (accessed April 21, 2009).

[3] M. T. Goodrich, M. J. Atallah, and R. Tamassia, "Indexing Information for Data Forensics," in *Proceedings of the Conference on Applied Cryptography and Network Security*, Springer Lecture Notes in Computer Science 3531, pp. 206–221, 2005.

[4] B. Li, M. S. Hsiao, and S. Sheng, "A Novel SAT All-Solutions Solver for Efficient Preimage Computation," in *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe*, Volume 1, February 2004.

[5] M. Malmgren, "An Infrastructure for Database Tamper Detection and Forensic Analysis," Honors Thesis, University of Arizona, May 2007. `http://www.cs.arizona.edu/projects/tau/tbdb/MelindaMalmgrenThesis.pdf` (accessed March 27, 2009).

[6] J. Mena, **Investigative Data Mining for Security and Criminal Detection**, Butterworth Heinemann, 2003.

[7] K. E. Pavlou and R. T. Snodgrass, "Forensic Analysis of Database Tampering," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 109–120, Chicago, June, 2006.

[8] K. E. Pavlou and R. T. Snodgrass, "Forensic Analysis of Database Tampering," ACM *Transactions on Database Systems* 33(4):Article 30, 47+25 pages, November 2008.

[9] S. Sheng and M. S. Hsiao, "Efficient Preimage Computation Using A Novel Success-Driven ATPG," in *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe*, Volume 1, March 2003.

[10] R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper Detection in Audit Logs," in *Proceedings of the International Conference on Very Large Databases*, pp. 504–515, Toronto, Canada, September 2004.

[11] C. Strachey, "Bitwise operations," *Communications of the ACM* 4(3):146, March 1961.

[12] U.S. Dept. of Health & Human Services, The Health Insurance Portability and Accountability Act (HIPAA), 1996, `http://www.cms.hhs.gov/HIPAAGenInfo/` (accessed April 16, 2009).

[13] U.S. Public Law No. 107-204, 116 Stat. 745. The Public Company Accounting Reform and Investor Protection Act, 2002.

APPENDIX

This appendix includes the proofs of all the theorems and lemmata mentioned in the paper, in Sections A through E. Section F is comprised of the proofs of correctness for all the functions introduced in the paper. A worked example of the candidate set generation for the target $t = 1010$ can be found in Section G.

A. Proof of Lemma 1

Lemma 1: $C_{t,k} = C_{t,2}$ if $l \geq z(t) > 0$ and $2 \leq k \leq 2^{z(t)}$. In other words, the candidate set remains invariant given that the stated conditions are met.

Proof:

First we show that $C_{t,k} \subseteq C_{t,2}$. Let $AND_k((b_1, b_2, \ldots, b_k)) = t$ for some t. Then $b_1, b_2, \ldots, b_k \in C_{t,k}$. Also, $b_1, b_2, \ldots, b_k \geq t$ because $t = \min\{C_{t,k}\}$. Consider the following 2-tuples: (b_1, t), (b_2, t), ..., (b_k, t). If we apply the AND_2 function to each 2-tuple the result is t, due to the minimality of t which masks all other binary numbers in $C_{t,k}$. Thus, all of $b_1, b_2, \ldots, b_k \in C_{t,2}$.

Conversely, we show that $C_{t,k} \supseteq C_{t,2}$. Given a series of 2-tuples (b_1, b_2), (b_3, b_4), ..., (b_{k-1}, b_k) which are pre-images of t under the function AND_2, and therefore $b_1, b_2, \ldots, b_k \in C_{t,2}$, we can create the following k-tuple $(b_1, b_2, \ldots, b_k)$ which is a pre-image of t under the AND_k function. The reason for this is because bitwise *AND*ing is an associative operation. Thus $b_1, b_2, \ldots, b_k \in C_{t,k}$. Therefore we have proved that $C_{t,k} = C_{t,2}$. □

B. Proof of Theorem 1

Theorem 1:

$$C_{t,k} = \begin{cases} \{t\} & , k = 1 \quad (1) \\ \emptyset & , z(t) = 0 \wedge k > 1 \quad (2) \\ C_{t,2} \neq \emptyset & , l \geq z(t) > 0 \wedge 2 \leq k \leq 2^{z(t)} \quad (3) \\ \emptyset & , l \geq z(t) > 0 \wedge k > 2^{z(t)} \quad (4) \end{cases}$$

Proof:

Case (1): $k = 1$

We want to find the binary numbers that map to t. In this case $k = 1$, i.e., the pre-image is unique and not *AND*ed with another number to produce t. The function is essentially the identity function so the candidate set is $C_{t,1} = \{t\}$.

Case (2): $z(t) = 0$, $k > 1$

Since $z(t) = 0$ the target binary number is $t = 111\cdots1$, i.e., a binary string of only '1's. We require that k (at least 2) binary numbers are *AND*ed in order to produce t. Suppose these numbers exist. Also, the formulation of the problem requires that they are all distinct. Then at least one of them will have a '0' as a digit because $111\cdots1$ is the only number of length l with no zeros. But this implies that their image under the *AND* function will also have at least one '0' digit which contradicts the fact that the target binary number t has $z(t) = 0$. Therefore, no such k numbers can exist. Thus $C_{11\cdots1,k} = \emptyset$ for $k > 1$.

Cases (3) and (4) are closely related.

Case (3) $l \geq z(t) > 0 \wedge 2 \leq k \leq 2^{z(t)}$:

Lemma 1 provides this case.

Case (4) $l \geq z(t) > 0 \wedge k > 2^{z(t)}$:

Here the target binary number has at least one '0' and we require at least two binary numbers to be *AND*ed in order to produce t. Only binary numbers which have at least as many '1's, and at the same positions, as the target string can achieve this. Thus the positions of the '1's are fixed and only the positions with zeros in t can have variations, i.e., 1 or 0. This explains why the cardinality of the candidate set is $2^{z(t)}$: there are $z(t)$ positions (the number of zeros) and each can independently take two values. If k exceeds the cardinality of $|C_{t,2}| = 2^{z(t)}$ then we are trying to find k-tuples which have a greater number of components than the total number of distinct binary numbers in $C_{t,2}$. This would force repetition in the components and this by definition is prohibited. Thus no such k-tuples can exist and $C_{t,2}$ will be empty. □

The proof reveals a very simple characterization for the candidate sets. A candidate set, in essence, comprises all the binary numbers which have '1's at the same positions as the target t and have at least as many total number '1's as t. Starting with our example target string $t = 1010$, all the elements in $C_{1010,2}$ will have the form 1␣1␣ where ␣ could be 1 or 0. More specifically, $C_{1010,2} = \{1010, 1011, 1110, 1111\}$. This explains why a binary string of all '1's, denoted by $11\cdots1$, appears in all the candidate sets $C_{t,2}$ (except its own, i.e., $C_{11\cdots1,2}$), whereas, a binary string of all '0's, denoted by $00\cdots0$, appears only in its own candidate set. (See also the discussion at the end of Section III for more intuition on this.)

The proof also implies that the target binary number will always be an element of its own candidate set, and actually the smallest such element, i.e., $t = \min\{C_{t,k}\}$. Other elements will have one or more '1's in positions that have '0's in t, and thus will be larger than t. This puts a lower bound of $\Omega(2^{z(t)})$ on the creation of a specific candidate set. This is because one must spend $2^{z(t)}$ time to create all of the $2^{z(t)}$ combinations.

C. Proof of Lemma 2

Lemma 2: For $k = 2$, the candidate sets of all the binary numbers of length l are unique.

Proof:

Case (1): We have $C_{t,2}$ and $C_{t',2}$ with $t \neq t'$, $|t| = |t'|$, and $z(t') \neq z(t)$ where $z(t)$ and $z(t')$ are the number of zeros of targets t and t' respectively. Assume without loss of generality that $z(t) > z(t')$. Then, since both numbers have the same length there exists at least one position in t where t has a '0' and t' has a '1'. Since t' has a '1' at that position then *all* the numbers in its candidate set will have a '1' at that same position. This is not the case with the numbers in $C_{t,k}$ since they can have either a '0' or a '1' at that position. Therefore $C_{t,2} \neq C_{t',2}$.

Case (2): We have $C_{t,2}$ and $C_{t',2}$ with $t \neq t'$ and both t' and t have the same number of zeros $z(t)$. This implies they also have the same number of '1's since they both have the same length. However, for the two numbers to be different, there must exist at least one position in t where t has a '0' and t' has a '1'. Using the same argument as before this implies that $C_{t',2} \neq C_{t,2}$. □

Given this lemma, for $k = 2$, there are 2^l candidate sets of the binary numbers of fixed length l, i.e., $|S_{l,2}| = 2^l$.

D. Proof of Theorem 2

Theorem 2: Assume $y = p \bullet t = \{0,1\}^x t$, $0 < x < l$, $0 \leq z(t) \leq l - x$ and $q = 2^{z(t)}$. Then:

$$
{}_{l-x}A_{t,k} = \begin{cases} \text{N/A}, & k > 2^{l-x} \quad (1) \\ [t] \ , & k = 1 \quad (2) \\ \emptyset \ , & z(t) = 0 \wedge 1 < k \leq 2^{l-x} \quad (3) \\ \bigcup_{0 \leq i < q} [\text{Suffix}_x({}_lA_{y,2}[i])], & \\ \qquad l - x \geq z(t) > 0 \wedge 2 \leq k \leq q \quad (4) \\ \emptyset \ , & l - x \geq z(t) > 0 \wedge k > q \quad (5) \end{cases}
$$

Proof:

Case (1):

The candidate set is not defined when we try to deduce a candidate set for a binary number of length $l - x$ given that the (original) k is greater than the total number of possible numbers that can be created using $l - x$ digits, i.e., 2^{l-x}. This is true since as discussed at the beginning of the paper this would force repetition of a binary number.

Case (2), Case (3) and Case (5):

These follow directly from the proof of Theorem 1.

Case (4):

It is worth elucidating here the nature of the number q. This number can be thought of as the cardinality of the candidate set of the suffix t: $q = 2^{z(t)}$ according to Corollary 1. It can alternatively be defined as $q = (1/2^{z(p)}) \cdot |{}_lC_{y,2}|$, that is, it is the cardinality of the candidate set of the original target y scaled down by a power of 2. This power of 2 is given by the number of zeros present in the truncated prefix p. Regarding q in this respect is consistent with the its initial assumption as $q = 2^{z(t)}$. This is true since $y = \{0,1\}^x t \Rightarrow z(y) = z(p) + z(t)$, which in turn implies that $q = (1/2^{z(p)}) \cdot |{}_lC_{y,2}| = 2^{z(y)}/2^{z(p)} = 2^{z(t)}$.

We prove case (4) by induction on x. Define proposition:

$P(x) : {}_{l-x}A_{t,k} = \bigcup_{0 \leq i < q} [\textit{Suffix}_x({}_lA_{y,2}[i])]$ for $(l - x \geq z(t) > 0) \wedge (2 \leq k \leq 2^{z(t)})$ and $q = 2^{z(t)}$.

Basis of induction: Prove $P(1)$ is true.

Let $x = 1$. Here the prefix p is a single bit. We have that $q = (1/2^{z(\{0,1\})}) \cdot |{}_lC_{y,2}| = 2^{z(t)}$, $y = \{0,1\} \bullet t$ and we want to prove that ${}_{l-1}A_{t,k} = \bigcup_{0 \leq i < q} [\textit{Suffix}_1({}_lA_{y,2}[i])]$. Thus, ${}_{l-1}A_{t,k} = [\textit{Suffix}_1({}_lA_{y,2}[1]), \textit{Suffix}_1({}_lA_{y,2}[2]), \ldots, \textit{Suffix}_1({}_lA_{y,2}[q])]$. What $P(1)$ essentially claims is that the new candidate array ${}_{l-x}A_{t,k}$ can be computed by simply selecting the first q elements of the candidate array ${}_lA_{y,k}$ and removing the leftmost digit from each such element selected.

Case (i) Assume that $p = 0$ (this corresponds to the example, given in Section VII, of deriving ${}_3A_{010,2}$ from ${}_4A_{0010,2}$). With respect to this first digit of the target binary string y we can divide the elements of its candidate array into two groups: those which have a '1', and those which have a '0' at that leftmost position. Due to the way these elements are created, resulting in the elements of the candidate array being sorted in increasing order, the elements with a '1' for a leftmost digit must all

$$
\begin{aligned}
{}_{l-(x+1)}A_{t',k} &= {}_{(l-x)-1}A_{t',k} && \text{apply basis of induction}\\
&= \bigcup_{0\le i<\hat{q}} [\textit{Suffix}_1({}_{l-x}A_{\{0,1\}t',2}[i])] && \text{where } \hat{q} = \frac{1}{2^{z(\{0,1\})}}|{}_{l-x}C_{\{0,1\}t',2}|\\
&= \bigcup_{0\le i<\hat{q}} [\textit{Suffix}_1({}_{l-x}A_{\{0,1\}t',k}[i])] && \text{candidate set is invariant when } k \le 2^{z(t')}\\
&= \bigcup_{0\le i<\hat{q}} [\textit{Suffix}_1({}_{l-x}A_{t,k}[i])] && \text{since } \{0,1\}\bullet t' = t\\
&= \bigcup_{0\le i<\hat{q}} [\textit{Suffix}_1((\bigcup_{0\le j<q} [\textit{Suffix}_x({}_{l}A_{y,2}[j])])[i])] && \text{by inductive hypothesis}\\
&= \bigcup_{0\le i<\hat{q}} [(\bigcup_{0\le j<q} [\textit{Suffix}_1(\textit{Suffix}_x({}_{l}A_{y,2}[j])])[i])] && \text{the suffix and union operations commute}\\
&= \bigcup_{0\le i<q'} [\textit{Suffix}_1(\textit{Suffix}_x({}_{l}A_{y,2}[i]))]\\
&= \bigcup_{0\le i<q'} [\textit{Suffix}_{x+1}({}_{l}A_{y,2}[i])]
\end{aligned}
$$

Fig. 10. The inductive step of Theorem 2.

appear after those with a '0' at the same position. Depending upon its position, each digit encodes the numbers in the range 2^{i-1} to $2^i - 1$ where i ($1 \le i \le l$) is the position of the digit numbering the string from right to left. So by removing the leading '0' from y results in a string t which cannot encode any numbers in the range 2^{l-1} to $2^l - 1$. Thus the candidate array of ${}_{l-1}A_{t,k}$ will have the same elements as the candidate array of ${}_{l}A_{y,k} = {}_{l}A_{y,2}$ except for the numbers encoded by the extra leading digit. But we know that each additional '0' introduced doubles (the position can be filled by a '0' or a '1') the count of numbers that can be encoded which implies removing a '0' will halve the count of numbers encoded: $z(p) = 1 \Rightarrow z(t) = z(y) - 1 \Rightarrow |{}_{l-1}C_{t,k}| = 2^{z(t)} = 2^{z(y)-1} = \frac{1}{2}|{}_{l}C_{y,k}|$. Thus the two groups of elements mentioned in the beginning will be equinumerous: the elements in the second half have essentially the same bit pattern as the elements in the first half but with a '1' at the leftmost position instead of a '0'. By removing the leftmost digit from each of the elements in ${}_{l}A_{y,k}$, the first half will have a leading '0' removed, something which will not change their numerical value, while the second half which will have a leading '1' removed will produce identical numbers of length $l-1$ to the truncated numbers in the first half. This is the reason why ${}_{l-1}A_{t,k}$ will comprise the suffixes starting at position 1, of the elements in the first half (i.e., $(1/2^1) \cdot |{}_{l}C_{y,2}| = q$) of the numbers in the array ${}_{l}A_{y,2}$.

Case (ii) Assume that $p = 1$ (this corresponds to the example, given in Section VII, of deriving ${}_{3}A_{010,2}$ from ${}_{4}A_{1010,2}$). In this case the situation is simpler since all the elements in ${}_{l}A_{y,k}$ can only start with a '1'. Since the number of zeros in t remains unaltered ($z(p) = 0 \Rightarrow z(y) = z(t)$), this implies that $|{}_{l-1}C_{t,k}| = |{}_{l}C_{y,k}|$. Thus removing the leftmost digit from all the elements of ${}_{l}A_{y,k}$ will yield directly the desired elements of the new candidate set since each of the truncated elements will have the same numerical value as their binary number counterparts of length l with a '0' at the leftmost position. Again the new candidate array ${}_{l-1}A_{t,k}$ will comprise the suffixes starting at position 1, of the q ($= (1/2^0) \cdot |{}_{l-1}C_{t,k}|$) first elements (in this case all of them) of ${}_{l}A_{y,k}$.

Inductive step: Prove that $P(x) \longrightarrow P(x+1)$

We assume that ${}_{l-x}A_{t,k} = \bigcup_{0\le i<q}[\textit{Suffix}_x({}_{l}A_{y,2}[i])]$, where $q = (1/2^{z(p)}) \cdot |{}_{l}C_{y,2}|$, and $y = p \bullet t = \{0,1\}^x t$ is true and seek to use this inductive hypothesis to prove ${}_{l-(x+1)}A_{t',k} = \bigcup_{0\le i<q'}[\textit{Suffix}_{x+1}({}_{l}A_{y,2}[i])]$ where $\{0,1\}t' = t \Rightarrow y = \{0,1\}^x t = \{0,1\}^x\{0,1\}t' = \{0,1\}^{x+1}t'$,

and $q' = (1/2^{z(p)+z(\{0,1\})}) \cdot |_l C_{y,2}|$. The inductive step is shown in Figure 10. By the first principle of mathematical induction the initial proposition is true. □

E. Proof of Theorem 3

Candidate sets also exhibit the following fundamental property: they are related (specifically, through set intersection) to the candidate sets of the constituent binary numbers that combine (through logical *OR*) to form the target.

Theorem 3: Let $C_{t,k}$, $t \in \mathbb{B}$, and $a_1, a_2, \ldots, a_m \in \mathbb{B}$ s.t. $\bigvee_{j=1}^{m} a_j = t$ for some $m \leq 2^{|t|}$ and let also $2 \leq k \leq 2^{z(t)}$. Then:

$$C_{t,k} = C_{\bigvee_{j=1}^{m} a_j,k} = \bigcap_{j=1}^{m} C_{a_j,k}$$

Proof:
Forward direction $\Longrightarrow$:
Let $AND_k((b_1, b_2, \ldots, b_k)) = t$. This implies $b_1, b_2, \ldots, b_k \in C_{t,k}$. We need to show that $b_1, b_2, \ldots, b_k \in \bigcap_{j=1}^{m} C_{a_j,k}$. By definition we know that $b_1 \wedge b_2 \wedge \ldots \wedge b_k = t$. However, we are also given that $\bigvee_{j=1}^{m} a_j = t$. Thus, $\bigvee_{j=1}^{m} a_j = t = \bigwedge_{i=1}^{k} b_i$. Therefore, we must prove that for every b_i ($1 \leq i \leq k$) there exists a series of $k-1$ distinct binary numbers (and different from b_i), $d_1, d_2, \ldots, d_{k-1}$ such that $b_i \wedge d_1 \wedge d_2 \wedge \ldots \wedge d_{k-1} = a_j \Rightarrow b_i, d_1, d_2, \ldots, d_{k-1} \in C_{a_j,k}$ for each $a_j, 1 \leq j \leq m$. In other words, each one of the b_is must appear in the pre-image of each one of the a_js.

We proceed to show how to produce all the requisite $b_i, d_1, \ldots, d_{k-1}$ given a specific b_i and a_j pair. Let x be the number of '1's in the binary number t, y be the number of '1's in a specific b_i, and w the number of '1's in a specific a_j. Then $y \geq x$ since b_i must have at least the same number of '1's, and at the same positions, as the target number t. This is true for all b_i since for a '1' to appear at a specific position in t then *all* the binary numbers b_i, which when *AND*ed produce t, must have a '1' at the same position. Likewise, $x \geq w$ since a_j must have at most the same number of '1's as the target number t. Again, this is true for all a_j since for a '1' to be preserved at a specific position in t at least one of the a_j must have a '1' at that same position. Using the observation above we begin with some b_i and pick d_1 to be a_j. This works because we want a number d_1 which has a zero at the same positions as a_j does, in order to mask any '1's b_i has at those positions. d_1 should also have a '1' wherever a_j does, so that the '1'is preserved after the *AND* operation. Note that if a_j has a '1' at a certain position we are guaranteed to have a '1' at the same position in b_i because t will have a '1' at that position (as discussed previously). All the rest of the $k-2$ binary numbers can be created from a_j and there are enough of them: $2^{z(a_j)} - 1$ (the '-1' is there because we are excluding a_j itself) where $z(a_j)$ is the number of zeros in a_j. We are given that $k \leq 2^{z(t)}$ and since $w + z(a_j) = x + z(t) = l$ and $x \geq w$ then $z(t) \leq z(a_j)$. Thus, $k - 2 < k \leq 2^{z(t)} \leq 2^{z(a_j)} \Rightarrow k - 2 \leq 2^{z(a_j)} - 1$. This implies that each of the b_i is an element of each of the $C_{a_j,k}$ and therefore an element of their intersection. Thus, $C_{\bigvee_{j=1}^{m} a_j,k} \subseteq \bigcap_{j=1}^{m} C_{a_j,k}$.

Backward direction $\Longleftarrow$: Conversely, let $b \in \bigcap_{j=1}^{m} C_{a_j,k}$. Then $(b \in C_{a_1,k}) \wedge (b \in C_{a_2,k}) \wedge \ldots (b \in C_{a_m,k})$. This implies that b has a '1' at the same positions as a_1, b has a '1' at the same positions as a_2 and so on until a_m. Thus the fact that b belongs to all the candidate sets of the a_is, fixes the positions of the '1's while the remaining positions could be '0' or '1'. Thus b captures a certain set of numbers. Now, consider $\bigvee_{j=1}^{m} a_j = t$. We know that t, as a result of an *OR* operation, will have a '1' wherever at least one a_i has a '1' at that position, and a '0' wherever all a_is have a '0' at that position. The candidate set of target t comprises all the numbers which have a '1' at the same position as t and at least as many '1's as t, i.e., wherever t has a '0' the pre-images can have a '0' or a '1'. But this is exactly the same set of numbers captured by b so $b \in C_{t,k}$. Therefore, $C_{\bigvee_{j=1}^{m} a_j,k} \supseteq \bigcap_{j=1}^{m} C_{a_j,k}$. □

This lemma provides a pleasing symmetry between the logical *AND* in the definition of the candidate set and the logical *OR* used above to form the target.

F. Proofs of Correctness

In this section we provide proofs of correctness for the various algorithms proposed. To make this easier, Figure 11 shows the dependency graph between the functions implementing the Tiled Bitmap Algorithm. A directed edge from node A to node B is interpreted as "function A (may) call(s) function B." We will provide the proofs by considering the functions in a bottom-up fashion.

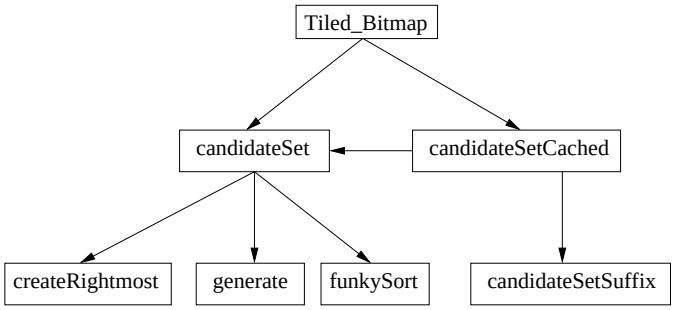

Fig. 11. The dependency graph of the functions implementing the Tiled Bitmap Algorithm.

Lemma 3: The createRightmost function (Figure 4), given a binary target t of length l creates an array named *rightmost* of size $l+1$. An element *rightmost*$[p]$ ($0 \leq p \leq l$) is the index of the rightmost zero in t to the left of index p (in t), non-inclusive. If such an index does not exist or is not defined, then *rightmost*$[p] = -1$.

Proof: At position p we need to know the position of the rightmost zero to the left of p. Hence, we scan the target from left to right and mark in *rightmost*$[p]$ (where $p = l - i - 1$) the index j at which we observed the latest zero. The use of the *flag* variable is required because we need to remember in the next iteration what digit we saw in the current iteration (lines 8–9). If we saw a zero (line 7) the value of j is updated and stored in *rightmost*$[p]$; otherwise the previous value is used (line 10). Note that on line 8 during the iteration for which $i = -1$, the left shift amount in the conditional becomes negative (i.e., the value is shifted to the right). This does not affect correctness since this is the last iteration. □

Lemma 4: The generate function (Figure 5), given a binary target t and a position of one of the 0s in t, enumerates $C_{t,k}$, that is, all 2^z binary numbers derived from t.

Proof: For an arbitrary p and t the function creates two subsets of $C_{t,k}$. The first subset is created by the recursive call on line 4 and comprises all the elements which have $t[p] = 0$ and for all digits to the left of $t[p]$: if the digit of t is 1 it stays as 1 in the enumeration, and if the digit is 0 it is either 0 or 1 in the enumeration. These two cases correspond to lines 4 and 5 in the recursive call.

The second subset is created by the recursive call on line 5 and comprises all the elements which have $t[p] = 1$ and for all digits to the left of $t[p]$: if the digit of t is 1 it stays as 1 in the enumeration, and if the digit is 0 it is either 0 or 1 in the enumeration. □

Lemma 5: The funkySort function (Figure 6), given $C_{t,k}$ resulting from the generate function, returns the array sorted in ascending order.

Proof: The sort is "funky" because it is linear and is based on the particular way generate( ) enumerates the elements of $C_{t,k}$. As discussed in Section VI this function first computes an array of indices (lines 8–13), which requires linear time, and then simply scans the indices array to arrive at the sorted $C_{t,k}$, also requiring linear time. □

Lemma 6: The candidateSetSuffix function (Figure 8), given a candidate set $C_{y,k}$ and the index t_{start} at which the suffix t starts in y, computes the candidate set $C_{t,k}$.

Proof: The candidateSetSuffix algorithm is a direct translation of Theorem 2 into code. Line 8 of the function corresponds to case (1) of the theorem. Line 9 corresponds to case (2), line 10 to cases (3) and (5), and finally, lines 11–13 correspond to case (4). The mathematical proof of the theorem's correctness can be found in Appendix D. □

Lemma 7: The candidateSet function (Figure 3), given a target number t, computes $C_{t,k}$ in ascending order.

Proof: The first part of the function is a direct translation of Theorem 1 into code. Line 8 of the code is correct by definition of the Cartesian product in Section V. Line 9 of the code corresponds to case (1) of the theorem. Lines 10–11 correspond to cases (2) and (4), and lines 12–15 correspond to case (3). The mathematical proof of the theorem's correctness can be found in Appendix B.

The correctness of createRightmost is established in Lemma 3. The correctness of generate is guaranteed by calling the function with *rightmost*$[l]$ and by Lemma 4. The function funkySort guarantees that $C_{t,k}$ is sorted by Lemma 5. Given the correctness of the algorithms this function depends on, calling the functions createRightmost, generate, and funkySort (lines 13, 14, 15), in that sequence, candidateSet yields the desired result. □

Lemma 8: The candidateSetCached function (Figure 7), given a target number t and a *cache* that was previously computed on line 16 of the

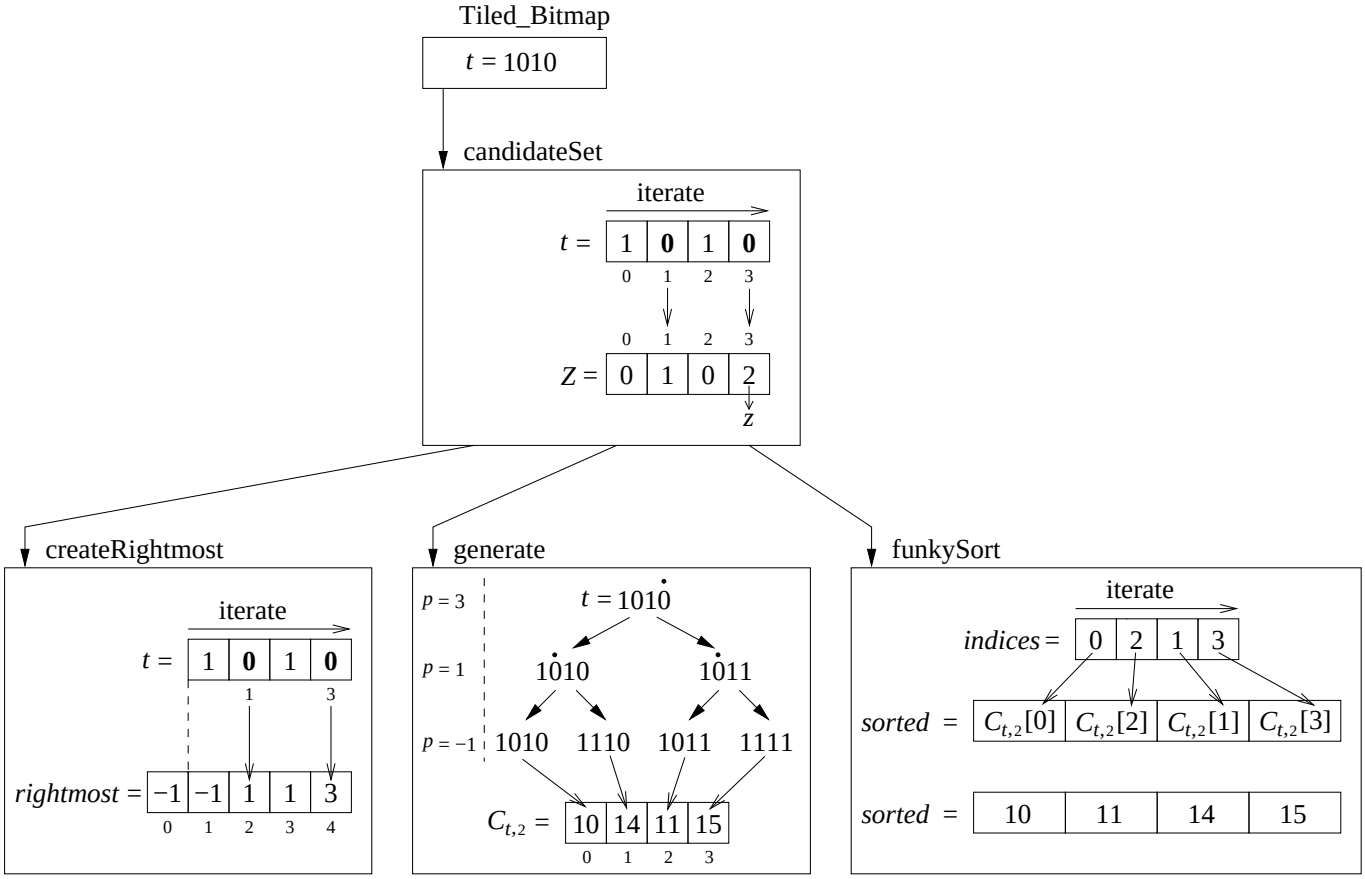

Fig. 12. Example of generation of the non-trivial candidate set for target $t = 1010$ with no cache available.

candidateSet function (Section VII), returns the candidate set $C_{t,k}$ either computed anew or derived from $C_{y,k}$.

Proof: In this function we assume that the start of the suffix can be computed correctly by *findSuffix* (not given). If the suffix exists then t_{start} will be greater or equal to 0 so the only task left is to decide (depending on the k associated with the *cache*) whether to call the candidateSetSuffix or the candidateSet function. Given that the two functions are correct by Lemmata 6 and 7, respectively, candidateSetCached yields the desired result. □

Theorem 4: The Tiled_Bitmap function (Figure 2), given a time of first validation failure, returns the set of possible corrupted granules.

Proof: The function iterates through all tiles (line 4) and checks each tile ending at time τ if it is corrupted or not (line 5). If it is, Tiled_Bitmap either calls candidateSet (line 9, Figure 2) or candidateSetCached (replacement line 9, Section VII) so that the candidate set is generated. Once the candidate set (C_{temp}) is correctly computed, the granules are renumbered to reflect their global position (line 11). □

G. Example of Candidate Set Generation

This section describes the creation of the candidate set for the specific target $t = 1010$, illustrated in Figure 12. We assume here the candidate set needs to be created from scratch, i.e., we are not dealing with a trivial case and a cache does not exist. The rectangles in the figure denote functions whose name appears above the box. The solid-tipped arrows in the figure denote function calls while the open-tipped arrows denote a correspondence between numbers or the direction of iteration.

Initially, the target $t = 1010$ is constructed by the Tiled_Bitmap function and then is passed to candidateSet. Within candidateSet the Z array is created (lines 5–7 of Figure 3) by inspecting the bits in the target from left to right and marking at each position in Z how many 0s have been encountered thus far. At index 0 the value in Z is 0 because at

the same index 0 in t the bit is not 0 but 1. On the other hand, at index 1 the value in Z is 1 because at index 1 in t the bit is 0. For this reason, the last element in Z is equal to the number of zeros in t and this count (in this case, 2) is stored in variable z.

The value in z is used throughout the generation of the candidate set (as witnessed in the pseudocode). However, the Z array itself is only used in the candidate set generation algorithm employing a cache and therefore will not be discussed from here on.

The createRightmost function is called by candidateSet in order to construct the *rightmost* array. The function iterates from right to left checking again for zeros and remembering at the current iteration/index i the bit value in the previous index $i-1$ (during the previous iteration). The index of the most recently encountered zero is stored in the current *rightmost* index. This is because, *rightmost*$[i]$ gives the index of the rightmost zero to the left of index i, non-inclusive. The value -1 is stored if such an index is not defined. Thus, in our example *rightmost*$[0] = -1$ because at $t[0]$ there aren't any bits to the left of it and hence the index of the rightmost 0 is not defined. Similarly, *rightmost*$[1] = -1$ because at the same index/bit position 1 in t the only number to the left of $t[1]$ is the 1 at $t[0]$; hence no rightmost zero is defined. On the other hand, at index 2 in t a rightmost zero to the left of $t[2]$ is defined, viz., it is the zero at index 1, $t[1] = 0$. Hence the index 1 at which the zero appears in is stored at index 2 in the *rightmost* array. Note that in order to avoid failing to register the index of the last zero (i.e., 3) the iteration has to go one step beyond the last bit of t.

Using the *rightmost* array and the target t the candidateSet function calls the recursive function generate. In Figure 12 the rectangular box of the generate function shows the results of the recursive calls in the form of a binary tree. At each level in the tree the index p of the zero under consideration is given to the left of the tree and at the same time marked over the position of the binary number with a solid black dot. Recall that at each recursive call the new index p' is the value stored at *rightmost*$[p]$. The terminating condition is satisfied when the index p is not defined, i.e., $p = -1$. The leaves of the binary tree are the elements of the candidate set which are shown in decimal form in the $C_{t,2}$ array. Clearly the elements are not enumerated in sorted order.

For this reason, candidateSet calls the function funkySort. The funkySort function first creates an array (*indices*) by staring from 0 and adding successively decreasing powers of 2 (starting from 2^{z-1}) to the results of the addition just produced (see also Section VI). The elements of the *indices* array when used to index into $C_{t,2}$ produce in the *sorted* array the sorted elements of the candidate set. In our example, the *indices* array is constructed by starting with 0 then adding $2^{z-1} = 2^1 = 2$ to 0 to get 2. Then adding 2^0 to 0 and 2 in order to create the indices 1 and 3. Then iterating through *indices* from left to right and using the values 0, 2, 1, 3 to index into $C_{t,2}$ accomplishes the sorting in *sorted*$= \{10, 11, 14, 15\}$. This works because of the particular way the generate function enumerates the candidate set elements.