## 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 $AND_k((b_1, b_2, \dots, b_k)) = t$ for some $t$. Then $b_1, b_2, \dots, b_k \in C_{t,k}$. Also, $b_1, b_2, \dots, b_k \geq t$ 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 $b_1, b_2, \dots, b_k \in C_{t,2}$.

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 $b_1, b_2, \dots, b_k \in C_{t,2}$, we can create the following $k$-tuple $(b_1, b_2, \dots, b_k)$ 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 $b_1, b_2, \dots, b_k \in C_{t,k}$. Therefore we have proved that $C_{t,k} = C_{t,2}$. □

### B. *Proof of Theorem 1*

*Theorem 1:*

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

*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 $t = 111 \cdots 1$, 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 $111 \cdots 1$ 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 $11 \cdots 1$, 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 $00 \cdots 0$, 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:

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

*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:
$P(x) : {}_{l-x}A_{t,k} = \bigcup_{0 \leq i < q} [\mathit{Suffix}_x({}_lA_{y,2}[i])]$ 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 ${}_{l-1}A_{t,k} = \bigcup_{0 \leq i < q} [\mathit{Suffix}_1({}_lA_{y,2}[i])]$. Thus, ${}_{l-1}A_{t,k} = [\mathit{Suffix}_1({}_lA_{y,2}[1]), \mathit{Suffix}_1({}_lA_{y,2}[2]), \ldots, \mathit{Suffix}_1({}_lA_{y,2}[q])]$. 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

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

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 ${}_lA_{y,k} = {}_lA_{y,2}$ 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: $z(p) = 1 \Rightarrow z(t) = z(y) - 1 \Rightarrow |{}_{l-1}C_{t,k}| = 2^{z(t)} = 2^{z(y)-1} = \frac{1}{2}|{}_lC_{y,k}|$. 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 ${}_lA_{y,k}$, 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., $(1/2^1) \cdot |{}_lC_{y,2}| = q$) of the numbers in the array ${}_lA_{y,2}$.

Case (ii) Assume that $p = 1$ (this corresponds to the example, given in Section VII, of deriving ${}_3A_{010,2}$ from ${}_4A_{1010,2}$). In this case the situation is simpler since all the elements in ${}_lA_{y,k}$ 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 $|{}_{l-1}C_{t,k}| = |{}_lC_{y,k}|$. Thus removing the leftmost digit from all the elements of ${}_lA_{y,k}$ 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 ${}_lA_{y,k}$.

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

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

## *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 $b_i$s must appear in the pre-image of each one of the $a_j$s.

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 $a_i$s, 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 $a_i$s 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, $C_{\bigvee_{j=1}^{m} a_j,k} \supseteq \bigcap_{j=1}^{m} C_{a_j,k}$. $\square$

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.

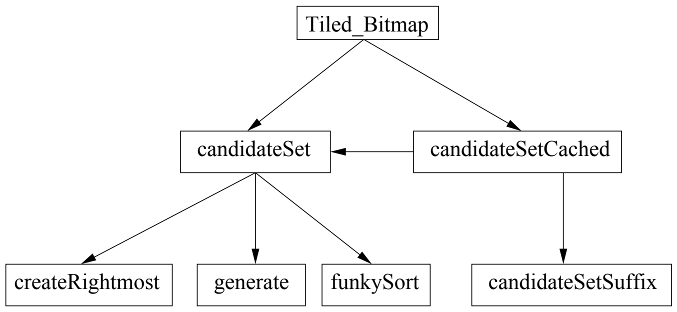

Fig. 11. The dependency graph of the functions implementing the Tiled Bitmap Algorithm.

*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

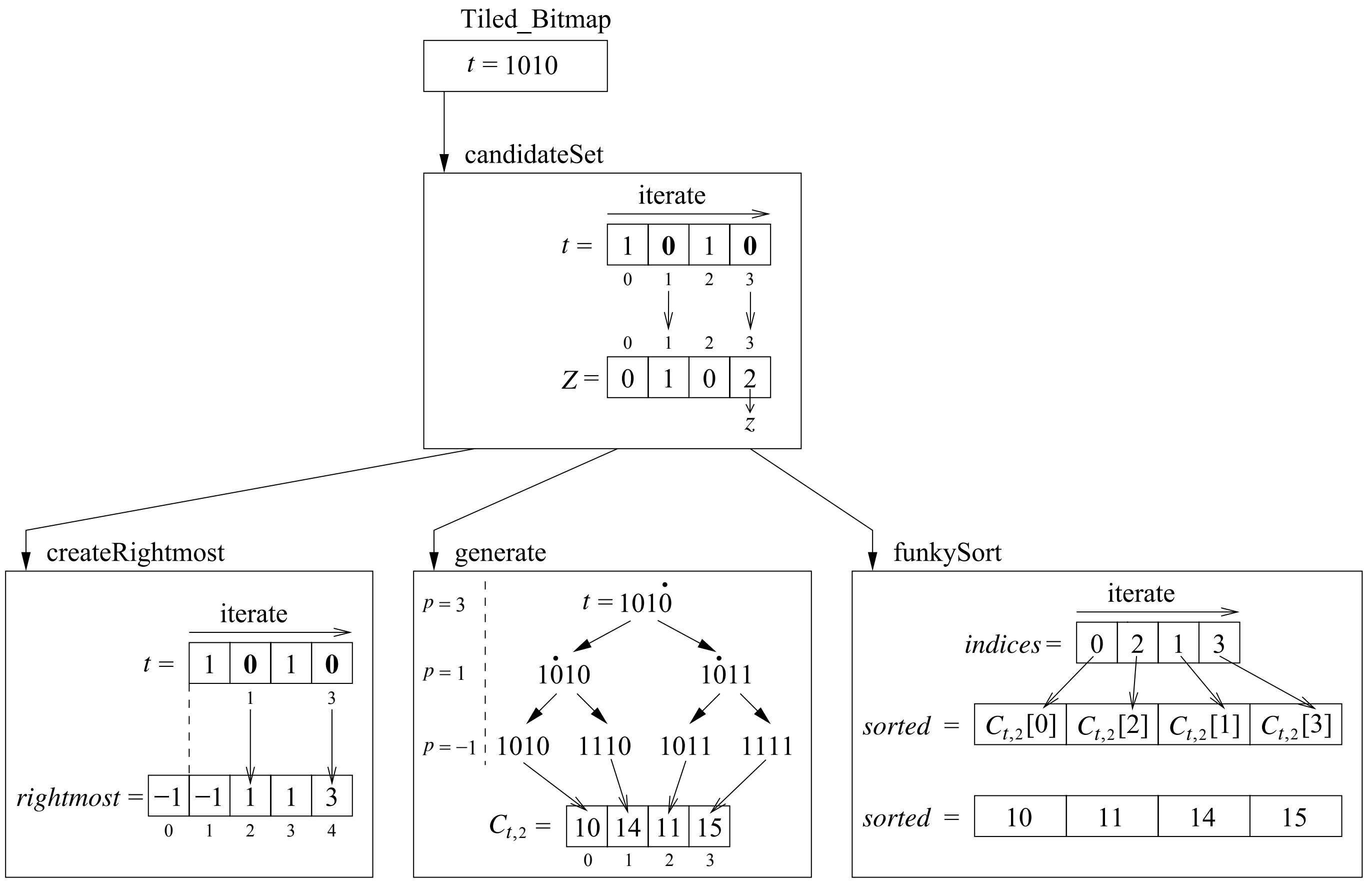

Fig. 12. Example of generation of the non-trivial candidate set for target $t = 1010$ with no cache available.

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 $\tau$ 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.