

Signature Limits: An Entire Map of Clone Features and their Discovery in Nearly Linear Time.

William Casey and Aaron Shelmire

Carnegie Mellon University, Software Engineering Institute,
Dell Secure Works
wcasey@cert.org
shelmire@counterthreatunit.com

Abstract. We address the problem of creating entire and complete maps of software code clones (copy features in data) in a corpus of binary artifacts of unknown provenance. We report on a practical methodology, which employs enhanced suffix data structures and partial orderings of clones to compute a compact representation of most interesting clones features in data. The enumeration of clone features is useful for malware triage and prioritization when human exploration, testing and verification is the most costly factor. We further show that the enhanced arrays may be used for discovery of provenance relations in data and we introduce two distinct Jaccard similarity coefficients to measure code similarity in binary artifacts. We illustrate the use of these tools on real malware data including a retro-diction experiment for measuring and enumerating evidence supporting common provenance in *Stuxnet* and *Duqu*. The results indicate the practicality and efficacy of mapping completely the clone features in data.

Keywords: Algorithm Design, Security, Analysis of Software Artifacts

1 introduction

In 2011 the security community identified a relation of provenance between the *Stuxnet* and *Duqu* malware families Chien *et al.* [2012]. The relation was substantiated by laborious reverse engineering digital artifacts¹ which reveled compelling evidence of code sharing. These reports addressed the underlying question of provenance in malware but left in question how much code sharing took place and further whether computational methods could be designed to measure and detect code sharing.

Scalable methods to triage and cluster malware using signatures have been considered in Bayer *et al.* [2009], Jang *et al.* [2011], Kang *et al.* [2012] and Lakhotia *et al.* [2013]; however each of these methods employ lossy data reductions. While

¹ Artifacts are malware binaries, files, or digital evidence of a computer/network attack.

these methods focus much attention on understanding error rates to achieve scalability, they leave open the question of whether the tradeoff between statistical power and scalability is necessary to achieve clustering methodology.

These problems provide a high level view of contemporary efforts in cyber security. Common to both problems is the need to identify and map all common strings or shared code segments termed *code clones* within a limited set of artifacts or against a reference data set of known artifacts. Calling on recent advances in suffix-data structures and succinct data structures, we consider efficient computational methods for mapping *code clones* (all copy features in data) which are both complete for provenance studies and compact enough to scale to large clustering problems.

Tree and Array construction and merge these advances with a practical model for exact code clones leading to practical methods for malware identification and triage and prioritization of reverse engineering resources.

1.1 Background.

This effort merges ideas from several distinct areas including: mathematics of measure theory, algorithm design calling on advances in suffix data structures, software engineering research which has recently suggested modes and models for code cloneage, and cyber security research which provides the motivating problems.

Code clones have been discussed in the area of software engineering where clones arise from a limited number of generating events including copy and paste, code reuse, common authorship, derived or augmented data, common linked artifacts, etc. For large software projects code cloning is an important factor for software maintenance and while the engineering benefits of cloning are debated there is general agreement that identifying clones is an important capability Kim *et al.* [2005]. Clones can be efficiently identified in large-scale software projects Kamiya *et al.* [2002] Li *et al.* [2004] where commercialized products have been developed. Recently modeling clone evolution has become an active area of research Antoniol *et al.* [2002] Livieri *et al.* [2007]. Definitions of code clones vary across the literature and are a developing area of research (see Kim *et al.* [2005] and Roy and Cordy [2007] for surveys). Our notion of code clones (presented in the next section) is novel and designed to both model gross structural features within the corpus using few quantities and be computable with suffix data structures. We present a mathematical description of a measure space making our notion of code clone comparable to all other formal notions.

The question of how to organize and represent a text corpus for optimized retrieval and search has been motivated by diverse problems in areas of information retrieval Amir *et al.* [1994], Blumer *et al.* [1987] and Ferragina and Grossi [1995], pattern matching Weiner [1973], software analysis Baker [1993], and bio-informatics Bieganski *et al.* [1994] Gusfield [1997] where there are several well developed techniques based on suffix trees McCreight [1976] Ukkonen [1985], compressed suffix trees Navarro and Mäkinen [2007], and suffix arrays Manber and Myers [1990] and Manzini and Ferragina [2004]. In addition indexing for

dynamic data sets Amir *et al.* [1994]; Ferragina and Grossi [1995] has been reported. Significant to very large data sets are Ferguson [2012] where researchers have considered applications of suffix data structures to data at scale.

In addition to the identification of longest common substrings (LCS), statistical analysis of the content space has been suggested Apostolico [2003] but not developed to the extent that useful code-clones can be identified in malware artifacts. While the topic of extending index-recallers to corpora is addressed in Bieganski *et al.* [1994], Blumer *et al.* [1987], Ferragina and Grossi [1995] and Gusfield [1997], with emphasis on suffix tree being central in Bieganski *et al.* [1994] and Gusfield [1997], our contribution develops tree-traversal and indexing arrays for quantities of *entropy*, *length*, *multiplicity*, and *file coverage* needed for discovery (i.e. “calling”) of clones in software executables.

We further consider methods to represent a set of clones that is both compact and complete. The use of suffix-trees for the analysis of set-algebra of corpus indices has been studied from a formal concept analysis and data-mining approach in Ferré [2007] where suffix trees are implemented to identify *string-scales* specific to a set lattice.

We show that this merger of a mathematical measure space for code clones combined with enhanced and tailored suffix data provides effective applications to problems in cyber security addressing provenance studies and data clustering.

2 Definitions and Clone Model.

For a string λ over a finite alphabet Σ , let $|\lambda|$ denote the string length, $\lambda[j] \in \Sigma$ the j th symbol, and $\lambda[j : k]$ the substring $\lambda[j]\lambda[j+1] \dots \lambda[k-1]$. A **corpus** is an ordered set of strings $\Omega = \{\omega_0, \omega_1, \dots, \omega_{n-1}\}$ over a common finite alphabet Σ , therefore each string $\omega_i \in \Sigma^*$ for $i \in \{0, 1, \dots, n-1\}$. The corpus size is measured by the number of strings $|\Omega| = n$, and the total length of corpus $||\Omega|| = \sum_{k=0}^{n-1} |\omega_k|$.

A **corpus region** is represented by a tuple (i, j, k) with $i < |\Omega|$ and $0 \leq j \leq k < |\omega_i|$; the first index specifies the corpus element from which the region is drawn (i.e. a given string ω_i), while the second and third indices provide the region within string ω_i beginning with and including offset j and covering up to but not including offset k . Associated with each corpus region (i, j, k) is the sub-string: $\omega_i[j : k] \in \Sigma^*$. Let \mathcal{R} denote the set of all corpus regions: $\mathcal{R} = \{(i, j, k) \mid i < |\Omega|, 0 \leq j \leq k < |\omega_i|\}$. Assume the following functions: $\text{FILE}((i, j, k)) = i$, $\text{OFFSET}((i, j, k)) = j$, and $\text{END}((i, j, k)) = k$ providing the coordinate projections for tuples in \mathcal{R} .

We identify the relation between corpus-regions and observed sub-strings by the **content map**:

$$\Gamma : \mathcal{R} \rightarrow \Sigma^* : \{(i, j, k)\} \rightarrow \omega_i[j, k].$$

We refer to the inverse of Γ as the **region recaller**; for any string $\lambda \in \Sigma^*$ a subset of matching corpus regions is returned:

$$\Gamma^{-1} : \Sigma^* \rightarrow 2^{\mathcal{R}} : \lambda \rightarrow \lambda^{-1}(\Omega),$$

with

$$\lambda^{-1}(\Omega) = \{(i, j, j + |\lambda|) \in \mathcal{R} : \omega_i[j : j + |\lambda|] = \lambda\}.$$

If a string (over Σ) is not observed in the corpus, the region recaller returns the empty set denoted \emptyset . The inverse of the empty string ϵ can be defined as \mathcal{R} without loss of generality or specificity. With Ω (and consequently Γ) fixed, we refer to $\lambda^{-1}(\Omega)$ as the **pullback** and denote it as λ^{-1} for short. The pullback of λ returns the corpus regions where the string λ is found.

The **observed language of the corpus** is the set of all strings with non-empty pullback:

$$\mathcal{L}(\Omega) = \{\lambda \in \Sigma^* \mid \lambda^{-1} \neq \emptyset\}.$$

2.1 Mathematics of Clones.

Clones are the content strings found in multiple locations of a corpus; they provide introspection and discovery opportunities for uncharacterized data. In order to concretely discuss clone concepts we describe clones mathematically as set systems in $\mathcal{L}(\Omega)$. Let Ω be a fixed corpus; we will use λ^{-1} to mean the pullback $\lambda^{-1}(\Omega)$ for any $\lambda \in \Sigma^*$. We start by introducing simple notions of code-clones and discuss how the different notions relate as nested sets. Next, we focus on statistical features of cloneage needed to be effective in malware discovery. Toward these goals we add additional qualifiers to enrich the concept of code-clones. We present a general nested model of cloneage in four parameters that will be used in applications for malware clone mapping, discovery, and measures. We indicate the underlying mathematics of this model and justify why we chose these clone quantities.

Simple Clone Concepts: A simple notion of **code-clone** is any snippet of code identified in multiple locations or in multiple files. Two definitions capturing these notions are:

$$\text{M-Clone} = \{\lambda \in \Sigma^* : |\lambda^{-1}| > 1\},$$

and

$$\text{F-Clone} = \{\lambda \in \Sigma^* : |\{\text{FILE}(x) : x \in \lambda^{-1}\}| > 1\}.$$

Note the dependencies in these models as $(\lambda \in \text{F-Clone}) \Rightarrow (\lambda \in \text{M-Clone})$. While the statement $|\{\text{FILE}(x) : x \in \lambda^{-1}\}| > 1$ is sufficient for $|\lambda^{-1}| > 1$, it is not necessary as λ may be found in each file of the corpus but never found duplicated at multiple offsets within any file.

Both of these sets are efficiently accessible using Suffix Trees Gusfield [1997]; however, for the task of malware discovery these notions are ineffective because a large volume of M-CLONE and F-CLONE may include byte padded sequences. Thus, additional considerations including the statistics of entropy are needed to distinguish a more interesting set of clones for discovery, triage and analysis.

To further generalize the notion of code clone we consider statistical measures of string content, such as the Shannon Entropy function and how it may qualify clones. Let $\lambda \in \Sigma^*$, for $v \in \Sigma$; let $X_v(\lambda) = |\{j < |\lambda| : \lambda[j] = v\}|$ be an observed

symbol count, and let $\theta_v(\lambda) = \frac{X_v(\lambda)}{|\lambda|}$ be the normalized symbol frequency. The **Entropy** for λ may be defined as $H(\lambda) = \sum_{\theta_v > 0} \theta_v \log \frac{1}{\theta_v}$.

Using the entropy function, we obtain a more useful set of clones by conjoining a lower entropy threshold to clone criteria, that is:

$$\text{M-Clone}_h = \{\lambda \in \Sigma^* : (|\lambda^{-1}| > 1) \wedge (H(\lambda) > h)\},$$

with the associated multi-file clone class as:

$$\text{F-Clone}_h = \{\lambda \in \Sigma^* : (|\{\text{FILE}(r) : r \in \lambda^{-1}\}| > 1) \wedge (H(\lambda) > h)\}.$$

This extension to the clone model provides selectability against low entropy strings such as null byte pads² which are common in binary artifacts; however, they are accidental clones which we must regard as uninteresting. In our experiments low entropy strings are often the longest common substring (LCS) and therefore a parameter such as h is necessary to recover meaningful signals from suffix-arrays.

In addition to *clone entropy* $H(\lambda)$, we further extend the concept of clones to include quantities of *clone length* denoted as $D(\lambda) = |\lambda|$, *clone multiplicity* denoted as $C(\lambda) = |\lambda^{-1}|$, and *file coverage* denoted as $F(\lambda) = |\{\text{FILE}(r) : r \in \lambda^{-1}\}|$,

Clone Model: We arrive at a **general model of clones** over the content $\mathcal{L}(\Omega)$ by letting the tuple $\langle d, h, f, c \rangle$ represent the following subset of $\mathcal{L}(\Omega)$:

$$\langle d, h, f, c \rangle = \{\lambda \in \mathcal{L}(\Omega) : (D(\lambda) > d) \wedge (H(\lambda) > h) \wedge (F(\lambda) > f) \wedge (C(\lambda) > c)\}.$$

Letting variables d, h, f, c range freely we have described a **clone class** within the context of the partial ordering of $2^{\mathcal{L}(\Omega)}$ by sub-set containment.

For $\langle d, h, f, c \rangle, \langle d', h', f', c' \rangle \in 2^{\mathcal{L}(\Omega)}$, we have the following nesting property:

$$\langle d', h', f', c' \rangle \subseteq \langle d, h, f, c \rangle \Leftrightarrow (d' \geq d) \wedge (h' \geq h) \wedge (f' \geq f) \wedge (c' \geq c).$$

Using the clone class in quantities d, h, f, c we may organize our simple clone concepts with set inclusion indicated by arrows as follows:

$$\begin{array}{ccc} \text{M-Clone}_h & = \langle 0, h, 1, 0 \rangle \leftarrow & \text{F-Clone}_h = \langle 0, h, 0, 1 \rangle \\ & \downarrow & \downarrow \\ \text{M-Clone} & = \langle 0, 0, 1, 0 \rangle \leftarrow & \text{F-Clone} = \langle 0, 0, 0, 1 \rangle \\ & \downarrow & \\ \mathcal{L}(\Omega) & = & \langle 0, 0, 0, 0 \rangle \end{array}$$

The clone class organizes the collection of clone sets in $\mathcal{L}(\Omega)$ into a nested family of *cylinder sets*. Cylinder sets (with set subtraction) may construct sets with each quantity bounded below and above; for example $\langle d_1, h_1, f_1, c_1 \rangle \setminus \langle d_2, h_2, f_2, c_2 \rangle$ specifies $\{\lambda \in \mathcal{L}(\Omega) : (d_1 \leq D(\lambda) < d_2) \wedge (h_1 \leq H(\lambda) < h_2) \wedge (f_1 \leq F(\lambda) < f_2) \wedge (c_1 \leq C(\lambda) < c_2)\}$. Two-sided bounds for each quantity provide a richer

² Zero padding a section of data is a common technique for file formats.

class of clones; we will see that two sided bounded quantities are also computable with a single pass over the suffix data structures in the TRAVERSE-TREE method presented in section 3.3. Closure under set operations (union, intersection, complement) of the cylinder sets generates a sigma algebra and therefore provides a mathematical measure space.

Justification of Clone Model. Closure of our clone class $\{\langle d, h, f, c \rangle : d \geq 0, h \geq 0, f \geq 0, c \geq 0\}$ with set negation, intersections (conjunctions), and unions (disjunctions) generates a sigma-algebra \mathcal{C} which is a coarsening of $2^{\mathcal{L}(\Omega)}$. Therefore our notion of clones provide a measure space: $\langle \mathcal{L}(\Omega), \mathcal{C} \rangle$ which approximates $\langle \mathcal{L}(\Omega), 2^{\mathcal{L}(\Omega)} \rangle$ and may be compared to other formal notions of clones. Although the dimension of this clone model is low with only four free variables we shall argue that these are simple to build into suffix array indices and sufficient for calling interesting sets of clones from malware artifacts. Further the low dimensionality reduces the search for features of a corpus quantified as regions in the parameter space of d, h, f, c . Functions *file coverage* F and *clone multiplicity* C are monotonically non-increasing in the suffix-order relation on $\mathcal{L}(\Omega)$; that is to say, if ζ is a suffix of λ then $F(\zeta) \geq F(\lambda)$ and likewise for C . However as mentioned above they measure different notions of cloneage with ratios expressing a comparison of self-similarity to similarity in the corpus at large. The *clone length* function D is monotonically increasing in the suffix-order relation as $D(\zeta) < D(\lambda)$. Therefore setting minimum values of d, f, c works to select clones from a corpus by using opposing criteria in the suffix-order on $\mathcal{L}(\Omega)$. The *clone entropy* function has no monotonic property in the suffix-order but is effective in selecting against low string entropy. Entropy selection is useful for executable modules which display wide variations including common null byte sequences.

3 Methodology: clone calling with arrays and representation for clone sets.

The main result of this section is that we adapt suffix trees/arrays to *call* or enumerate the members of $\langle d, h, f, c \rangle$ in time: $O(|\Omega| \log(|\Omega|))$. We further show that clone sets $\langle d, h, f, c \rangle$ are reducible to a much smaller subset called a *max-clone representation* by use of a suffix-relation on Σ^* . The max-clone representation admits to both meaningful visualizations and application of measures to identify and infer provenance in artifacts (discussed in the next section). We present a brief historical development of suffix data structures, subword trees, and arrays to discuss the TRAVERSAL-TREE procedure which produces the arrays enhanced with clone quantities. Our model for suffix data structures is Ukkonen’s suffix tree Navarro and Mäkinen [2007], Ukkonen [1985] and Ukkonen [1995] and we follow its terminology and developments; for further background we suggest Navarro [1999]. Since suffix arrays may emulate suffix trees Abouelhoda *et al.* [2004] our method is possible for various suffix array implementations as well. To be as general as possible we describe the minimum data requirements of suffix-tree nodes to complete the TRAVERSAL-TREE method.

3.1 Suffix Data Structures.

Given a set of strings S , an index tree (*trie*), such as the *PATRICIA trie* Morrison [1968], is a tree graph which encodes a finite state automaton (FSA) for acceptance of any input matching a member of S . Each string from S corresponds to a path from the *root* node to a *leaf*, and paths are merged by shared prefixes to form a trie (tree index). As an FSA, this structure may be considered an Aho-Corasick string matcher.

The set of all suffixes of string ω is denoted by $\sigma(\omega)$ and defined as: $\sigma(\omega) = \{\omega[0 : k] : k \in \{0, 1, \dots, |\omega|\}\}$. A **Suffix trie** for ω is constructed by creating a *PATRICIA trie* on $\sigma(\omega)$.

The *suffix trie* may be used as an entire index of all substrings because any substring of ω can be written as a prefix of a member of $\sigma(\omega)$. Further the tree structure is meaningful for the problem of content mapping as the internal branching nodes of the structure are in correspondence with redundant strings of the text, the deepest internal branch of which is called the **Longest Common Substring** (LCS) Gusfield [1997].

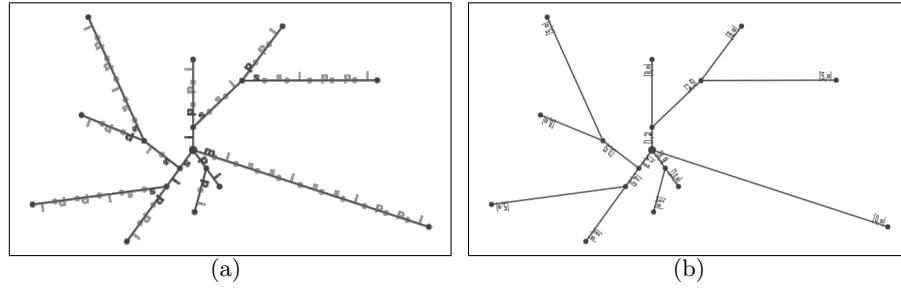


Fig. 1. (a) Suffix trie and tree for $\omega = \text{'mississippi'}$; states of the suffix trie are indicated by nodes and state transitions by edges labeled with letters. Ukkonen's suffix tree only requires explicit states (black) and is able to emulate the implicit states (gray) of the trie. The tree *root* node is in the center and the set $\sigma(\omega)$ is displayed in lexicographical order starting at angle $\frac{\pi}{2}$ and rotating 2π in a clockwise direction. Notice also the deepest branching node in the tree corresponds to longest common string 'issi'. (b) Suffix tree for $\omega = \text{'mississippi'}$. Replacement of the transition labels with offsets and length indices (referencing the input string ω) create the suffix tree. In addition each node maintains a set of children branches and a suffix pointer (not shown).

The suffix trie data structure admits to a compact representation by removing internal non-branching nodes and emulating transition-labels for implicit states (see figure 1(a)). Further there is no need to store transition-labels as they can be recovered from offsets (in ω), further reducing the space requirements for suffix tree nodes (see figure 1(b)).

For fixed and finite alphabet Σ , the resulting data structure is linear in space $O(|\omega|)$ and constructed in linear time $O(|\omega|)$ Ukkonen [1995]. Further the data structure can be traversed in linear time $O(|\omega|)$ to identify the deepest branching node and equivalently the LCS of the text (see Gusfield [1997] for additional details).

In addition to trees, suffix arrays are constructed in near linear to linear time Kärkkäinen *et al.* [2006] Kim *et al.* [2003]; Ko and Aluru [2003] and may emulate suffix trees Abouelhoda *et al.* [2004]; therefore what can be performed on Ukkonen’s tree extends in principle to many array implementations as well. More recently, succinct data structures have achieved greater compression of suffix trees and arrays for lossless index re-callers Manzini and Ferragina [2004] Navarro and Mäkinen [2007]; for example the Ferragina Manzini index structure (FM Index Manzini and Ferragina [2004]) utilizes the Burrows-Wheeler transform to compress the suffix array in memory.

3.2 Implementation: Construction of a Suffix Tree for Malware Artifacts.

In order to scale the suffix tree beyond system external memory (EM) data structures are possible Arge [1996] and Ferragina and Grossi [1995]. Beginning from Ukkonen’s suffix tree algorithm, we implemented an external memory set of c-programs for a corpus over the bytes alphabet $\Sigma = \{0, \dots, 255\}$.

We demonstrate an externalized version of Ukkonen’s suffix tree algorithm augmented to support corpus indexing (by adding file index and local offsets within the file) to leaf nodes. Below in Figure 2 we visualize a suffix tree data structure constructed to analyze string structure in *Aliser File Infector* malware artifacts.

3.3 Traversal of Ukkonen’s Suffix Tree to Create Clone Quantity Arrays.

Throughout the remainder of this section we assume a fixed corpus Ω with concatenated length $||\Omega||$ and number of artifacts $|\Omega|$, letting $\omega = \omega_0 \circ \dots \circ \omega_{|\Omega|-1}$ be the concatenation of artifacts in the corpus. The algorithm can generally be applied to any content map capable of emulating a suffix tree with the following minimum data fields for each node η of the tree: $\eta.O$ to access the offset in ω , $\eta.C$ to access children of η , $\eta.L$ to measure the length of the branch in the suffix tree between η ’s parent and η (i.e. the length of string $\omega[\eta.O : (\eta.O + \eta.L)]$)³. It is not necessary but beneficial to have a *suffix-link* $\eta.s$ pointing to the node representing the suffix of η , and for leaf nodes the *file identifier* $\eta.F$ and *local file offset* $\eta.o$.

³ String $\omega[\eta.O : (\eta.O + \eta.L)]$ cooresponds with the state transition labels from η ’s parent to η .

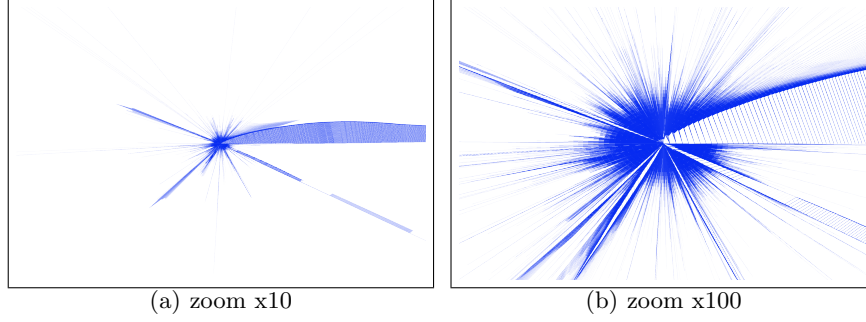


Fig. 2. Example: Suffix tree constructed for *Aliser* malware artifact family data (79 files, 6,643,712 bytes). Trees are lexicographically ordered starting from the branch cut 0 and winding counter clockwise to 2π . Notice that the bloom of wide branching and deep paths near argument 0 in the tree corresponds with substrings prefixed with null byte sequences; these are also low entropy strings.

Let $\eta.C$ be the children of state η , sorted in order by the transition character (i.e. the order of Σ). We denote the k 'th child (zero based index) of η as $\eta.C[k]$, and assume the function: $\text{CHILD}(\eta, k)$ which returns $\eta.C[k]$ if $k < |\eta.C|$ or \emptyset otherwise. Note that $|\eta.C|$ is bounded by $|\Sigma|$ for all nodes of the suffix-tree. Let *root* be the unique node not found as a child state for any other node.

Recall the correspondence of $\mathcal{L}(\Omega)$ and paths in the suffix tree: for suffix tree node η we indicate this relation with $\bar{\eta} \in \mathcal{L}(\Omega)$ where $\bar{\eta}$ is the *path string* obtained by concatenating strings upon all branches from *root* to η .

Throughout the traversal we maintain a stack⁴ of tuples denoted as \mathcal{S} . The tuples in the stack are of the following form:

$$\langle \eta, k, l, T, z, \theta, \delta \rangle.$$

With η a unique node identifier, k is a number between 0 and $|\eta.C|$ indicating how many children of η have been explored in post order, while l represents the length of $\bar{\eta}$ and supports the computation of the *clone length* function $D(\bar{\eta})$. The variable T represents the subset of corpus indices (file index) $\{0, 1, \dots, n-1\}$ indicating the covering files for $\bar{\eta}$ and supports the computation of the *clone file coverage* function $F(\bar{\eta})$. The quantity z counts the *clone multiplicity* function $C(\bar{\eta})$ by computing the total number of leaf descendants of η in the suffix tree. The value θ is a vector over alphabet symbols in correspondence with Σ supporting the computation of *clone entropy* function $H(\bar{\eta})$. Let $\langle 0 \rangle_{\Sigma}$ be a count vector (over symbols of Σ) with all values initialized to 0. Finally δ charts the topological depth in the suffix tree counting the number of nodes between the root and η .

⁴ Should the stack grow to sizes beyond system memory, externalized data structures to support a large stack are possible.

While the tree-traversal is a straightforward walk of the data structure, the ordering of computations needed to compute clone quantities D, H, F, C for qualifying in set $\langle d, h, f, c \rangle$ must be sequenced carefully so we distributed them into `PREORDERVISIT` and `POSTORDERVISIT` operations. We present the outline for traversal:

```

TRAVERSE-TREE
0: PUSH( $\mathcal{S}, \langle root, 0, 0, \emptyset, 0, \langle 0 \rangle_{\Sigma}, 0 \rangle$ )
1:  $\phi \leftarrow \epsilon$ 
2: while LENGTH( $\mathcal{S}$ )
3:   do  $\eta, k, l, T, z, \theta, \delta \leftarrow \text{POP}(\mathcal{S})$ 
4:      $\mu \leftarrow \text{CHILD}(\eta, k)$ 
5:     if  $\mu \neq \emptyset$ 
6:       PUSH( $\mathcal{S}, \langle \eta, k + 1, l, T, z, \theta, \delta \rangle$ )
7:       PUSH( $\mathcal{S}, \langle \mu, 0, l, \{\}, 0, \langle 0 \rangle_{\Sigma}, \delta + 1 \rangle$ )
8:       PREORDERVISIT( $\mathcal{S}, \phi$ )
9:     else POSTORDERVISIT( $\mathcal{S}, \eta, l, T, z, \theta, \delta, \phi$ )

```

While a node η has additional children to explore it will be pushed back onto the stack with its child index incremented (line 6), and the k th child μ will be pushed immediately after (line 7). When node η has exhausted the exploration of children, flow-control reaches line 9 where quantities for the sub-tree rooted at η will be aggregated upward to γ (the parent of η). In addition line 9 records a post-ordering of nodes in the tree, after which η will not re-enter the stack again. In addition `POSTORDERVISIT` traverses the content of $\mathcal{L}(\Omega)$ in lexicographical order. This outline completes the description of the traversal framework to compute clone quantities using suffix data structures.

Next we consider the `PREORDERVISIT` which provides the opportunity to initialize data for μ (child of η), extend the string ϕ with a contribution from μ to arrive at $\bar{\mu}$, and update θ needed to compute the entropy statistics $H(\bar{\mu})$:

```

PREORDERVISIT( $\mathcal{S}, \phi$ )
0:  $\mu, k, l, T, z, \theta, \delta \leftarrow \text{TOP}(\mathcal{S})$ 
1: if ( LEAF( $\mu$ ) )
2:    $z \leftarrow 1$ 
3:    $T \leftarrow T \cup \{\mu.F\}$ 
4:    $l \leftarrow -1$ 
5: else
6:    $l \leftarrow l + (\mu.L)$ 
7:    $\lambda \leftarrow \omega[\mu.O : (\mu.O + \mu.L)]$ 
8:    $\theta \leftarrow \theta + \langle \lambda \rangle_{\Sigma}$ 
9:    $\phi \leftarrow \phi \circ \lambda$ 

```

In line 0 of `PREORDERVISIT` we access μ 's variables stored at the stack's top (line 7 of `TRAVERSE-TREE`). The function `LEAF` may be implemented by checking the predicate: $(|\mu.C| = 0)$. In lines 2-4 we treat the case when μ is a leaf: quantities *clone multiplicity* z and *clone file cover* T are initialized and later

will be aggregated upward during the POSTORDERVISIT, and quantity l is set to -1 to indicate a suffix of ω and could be interpreted as “read to the end of corpus.” Lines 6-9 handle computations required for internal branch nodes; these include updating the string ϕ from $\bar{\eta}$ to $\bar{\mu}$ (line 9) by extracting the string from the corpus associated with the suffix link between μ and μ ’s parent η (Line 7). Updating the depth variable l from $|\bar{\eta}|$ to $|\bar{\mu}|$ is performed in line 6, and updating a running symbol count of ϕ is performed in line 8.

Next we consider the POSTORDERVISIT providing the last opportunity to perform computations obtained from node η :

```

POSTORDERVISIT( $\mathcal{S}, \eta, l, T, z, \theta, \delta, \phi$ )
0:  $\gamma, k_\gamma, l_\gamma, T_\gamma, z_\gamma, \theta_\gamma, \delta_\gamma \leftarrow \text{TOP}(\mathcal{S})$  % parent of  $\eta$  is  $\gamma$ 
1:  $z_\gamma \leftarrow z_\gamma + z$ 
2:  $T_\gamma \leftarrow T_\gamma \cup T$ 
3: PRINT(  $\eta, \eta.O, \eta.L, \eta.F, \eta.s, \delta, \langle l, h(\frac{1}{z}\theta), T, z \rangle$  )
4:  $\phi \leftarrow \phi[0 : l_\gamma]$  % RETURN STRING TO  $\bar{\gamma}$ .

```

Notice that in line 3, the printing of $\langle l, h(\frac{1}{z}\theta), T, z \rangle$ are evaluations of $D(\bar{\eta})$, $H(\bar{\eta})$, $F(\bar{\eta})$, $C(\bar{\eta})$. Line 2 and 3 of POSTORDERVISIT aggregate z the *clone multiplicity* and compute the *clone file cover* set T currently held by η upward to γ (η ’s parent) at the top of the stack at the time POSTORDERVISIT is called (note that only a parent γ can precede a child η in stack insertion: see lines 6-7 of TRAVERSE-TREE). Line 3 writes the enhanced array to output and could provide an opportunity to conduct further and more general analysis for clone membership. Finally line 4 reduces the current content string ϕ to $\bar{\gamma}$ by truncating $\eta.L$ symbols from $\bar{\eta}$.

Lemma 1 TRAVERSE-TREE is $O(|\Omega| \log(|\Omega|))$.

Proof: The total number of POP’s (line 3 of TRAVERSE-TREE) is bounded by twice the number of edges in the suffix tree and therefore bounded by $4|\Omega|$ as the maximum number of edges in a tree which is less than twice the number of leaf nodes. Therefore, the loop is performed $O(|\omega|)$ times and the complexity consideration is reduced to that of PREORDERVISIT and POSTORDERVISIT. The traversal guarantees that PREORDERVISIT and POSTORDERVISIT are called once per node.

Line 3 of PREORDERVISIT and lines 2 and 3 of POSTORDERVISIT are $\log(|\Omega|)$ set operations. Lines 7-9 of PREORDERVISIT can be analyzed by amortizing across all nodes of the tree during traversal, since the load size of λ over all nodes of the tree is no greater than loading the entire corpus. The total cost of all operations is therefore bounded by $O(|\Omega|)$. All other operations in PREORDERVISIT and POSTORDERVISIT are a $O(1)$. Therefore the entire runtime is bounded by $O(|\Omega| \log(|\Omega|))$. ♣

Note: More generally line 3 of POSTORDERVISIT could be replaced by any method that is $O(1)$ in the depth of stack \mathcal{S} and $O(\log(|\Omega|))$ to get a slightly stronger lemma allowing for additional analysis involving the relation between a node η and its parent γ , or some limited size ancestral chain, for example: $\eta, \eta.P, \eta.P.P$.

Enhancing a Suffix Array with Clone Quantities. With the runtime for TRAVERSE-TREE resolved as $O(|\Omega| \log(|\Omega|))$ we now focus on the transformation of data that line 3 of POST-ORDER-TRAVERSAL yields. While it produces a tabulated form of data that allows us to test node membership in $\langle d, h, f, c \rangle$, it also achieves a full map of all suffixes printed in lexicographical order thereby creating an *enhanced suffix array* augmented with quantities of *clone length* D , *clone entropy* of H , *clone file coverage* F and *clone multiplicity* C .

In the subsequent section we show how this map can be used to support measures leading to pairwise distance based on clones in common and clustering based on common clones. We suggest the outline above as a framework to extend notions of clones further; for example, measuring the distance to specific symbol frequency vectors such as a topic vector or witness complex De Silva and Carlsson [2004], which we plan to pursue as future work.

3.4 Max-Clone Representation.

Clone sets have an inherent redundancy which displays perplexing patterns related to the self similarity of the suffix tree data structure. To simplify matters we describe a representation of a set $\langle d, h, f, c \rangle$ that is both easy to visualize and minimal in that it selects the smallest subset of *representatives* from $\langle d, h, f, c \rangle$ for which all other members are suffixes of a representative with equal value for F and C . We shall argue that knowing all members provides no additional information beyond knowing the representation. We provide an indication of the type of reductions the representation offers in practice.

Two nodes in the externalized suffix tree are suffix-related⁵, denoted $\mu \prec \rho$, if $\bar{\rho} = x\bar{\mu}$ for some symbol $x \in \Sigma$. Given a specific clone set $\mathcal{B} = \langle d, h, f, c \rangle$ we can extend the suffix-relation \prec to members with a *level-set-suffix-relation* $\prec_{\mathcal{B}}$ on all nodes of the tree as:

$$(\mu \prec_{\mathcal{B}} \rho) \Leftrightarrow (\mu \prec \rho) \wedge (\bar{\mu} \in \mathcal{B}) \wedge (\bar{\rho} \in \mathcal{B}) \wedge (F(\bar{\mu}) = F(\bar{\rho})) \wedge (C(\bar{\mu}) = C(\bar{\rho})).$$

We define the **max-clone representation** of a clone class $\langle d, h, f, c \rangle$ as the strings associated with maximal elements of the relation $\prec_{\langle d, h, f, c \rangle}$, and we denote the *max-clone representation* as $\langle \langle d, h, f, c \rangle \rangle$. Computing the *max-clone representation* is easily seen to be $O(|\Omega|)$; see the appendix for a graph algorithm that computes the max-clone representation.

Justification: The suffix relation is a particularly appropriate order for reducing the representation (to maximal clones), because any non-representative member of the clone class is a suffix of a representative member with identical values of F, C (level set). Furthermore in applications we argue that this representation translates directly to the longest common strings of interest in data and we provide examples of how the max-clone representation may be visualized in figure 3 for *Duqu* and *Stuxnet* malware data.

⁵ Ukkonen's construction includes suffix links.

Reduction in practice: The max-clone representation is an effective data reduction in practice. In figure 3 we present a visualization of a max-clones for $\langle 1000, 2.0, 2, 2 \rangle$. In this case the number of nodes of the suffix tree quantified by $\langle 1000, 2.0, 2, 2 \rangle$ is 25,177, yet the max-clone representation comprises 7 clones located at 17 offsets in the corpus. The max-clone representation signals what and where largest relations in data may be found. In this case the max-clone representation selects 2.780868×10^{-4} fractional amount of clones from $\langle 1000, 2.0, 2, 2 \rangle$.

Conclusion: For corpus Ω the max-clone representation for $\langle d, h, f, c \rangle$ denoted $\langle\langle d, h, f, c \rangle\rangle$ is computable in $O(|\Omega| \log(|\Omega|))$ by first building suffix-data structures, traversing the suffix-data with TRAVERSE-TREE, and identifying maximal elements of $\prec_{\langle d, h, f, c \rangle}$.

4 Applications and Results.

The remainder of the paper focuses on applications of our methodology to malware artifact data. We address the motivating problems and illustrate results on actual malware data artifacts. We focus on the problem of *Stuxnet* and *Duqu* (which represents a difficult challenge in cyber security) and show the use of clone sets to identify and measure the evidence for provenance. Using the max-clone representation $\langle\langle d, h, f, c \rangle\rangle$, we sketch how to construct Jaccard similarity coefficients to compare artifacts in a pairwise manner. We present Jaccard coefficients for this problem and the results indicate that the relation between the *Stuxnet* and *Duqu* malware sets signals an overlap detectable with the Jaccard coefficient. Finally we consider the Jaccard similarity coefficients for cyber security data and provide experimental designs in terms of coverage and compression.

4.1 Mapping Clone Features and Visualization.

In Figure 3 we assemble a set of binary artifacts (the driver artifacts) matching anti-virus signatures for either *Duqu* or *Stuxnet* malware groups as studied in Chien *et al.* [2012], Falliere *et al.* [2010] where reverse engineering techniques discovered evidence supporting the hypothesis for a common *provenance* or history of development. While these discoveries were important to the security community and also (from 2012 forward) to the mainstream media, the question of identifying all the evidence supporting the findings remained open.

Conclusions: The visualization of data is useful for data triage and establishing priorities for costly reverse engineering resources. For example in the image 3 a high entropy string of sizable length is found in the slack section⁶ of binaries including both *Stuxnet* and *Duqu*.

⁶ Slack sections are areas of data not reported by the program's load table.

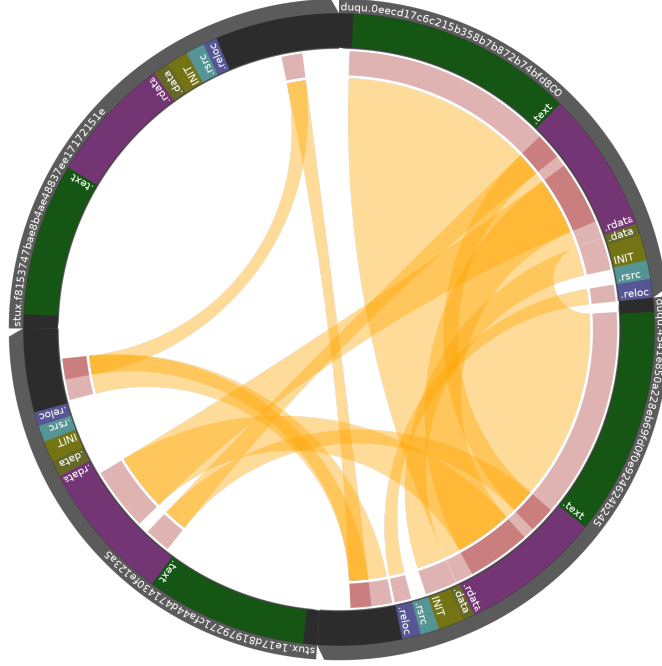


Fig. 3. Example: *Duqu* vs *Stuxnet*: visualization of $\langle\langle 1000, 2.0, 2, 2 \rangle\rangle$. Binaries of four malicious code samples (two from the *Duqu* family and two from the *Stuxnet* family) are illustrated as regions of an annulus. Small notches in the outer circumference mark the beginning of a binary and can be viewed at approximate angles of: $\frac{\pi}{2}, 0$ for the *Duqu* samples and $\frac{3\pi}{2}, \pi$ for the *Stuxnet* samples. Next files are divided into *sections* and color coded by section name *.text*, *.rsrc*, *.rdata*, *.data*, *INIT*, *reloc*. Max-Clones from $\langle 1000, 2.0, 2, 2 \rangle$ are illustrated as counter-arcs passing through the interior region and connecting orthogonally to the annular region representing the binary layouts. These counter-arcs show the locations and length of max-clones when $c > 1$. An annular region, just interior to the view of file formats, illustrates the copy number of each cloned region with a red alpha channel. Partial transparency (alpha channel) helps with signaling clone matches contained as substrings to larger matching clones.

4.2 Measurements of Shared Clone Features.

Using Figure 3, which illustrate $\langle\langle 1000, 2.0, 2, 2 \rangle\rangle$, we can identify and count the distinct number of clones as 7 max-clones with 14 distinct offsets in the corpus. In this section we develop the Jaccard similarity coefficient to measure the percentage of content in common (given a clone class) in pairs of files. We further illustrate how these measures may vary on the clone class $\langle d, h, f, c \rangle$.

Fixing the Corpus $\Omega = \{\omega_0, \dots, \omega_{n-1}\}$ and given a clone set $\langle d, h, f, c \rangle$, A Jaccard similarity coefficient for all pairs of artifacts is fairly straightforward and is computed as follows: For any subset $I \subset \{0, \dots, n-1\}$, identify all the clones

which have a region contained in all of the artifacts of I , so assume the function:

$$\text{COVER}(I) = \{\lambda \in \langle\langle d, h, f, c \rangle\rangle : I \subset \text{FILE}(\lambda^{-1})\}$$

with $\text{FILE}(\lambda^{-1}) = \{\text{FILE}(r) : r \in \lambda^{-1}\}$. For comparison of artifact i against a subset I we count the number of indices of ω_i covered by strings from $\text{COVER}(I)$.

$$A(i, I) = \sum_{a=0}^{|\omega_i|} \phi(\omega_i[a:], \text{COVER}(I))$$

with:

$$\phi(\omega[a:], S) = \begin{cases} 1 & \text{if } \exists b : \omega[a:b] \in S \\ 0 & \text{o.w.} \end{cases}$$

We introduce the Jaccard similarity coefficient as $J(I) = \frac{\sum_{i \in I} A(i, I)}{\sum_{i \in I} |\omega_i|}$ and interpret this as the percentage of a subset covered by the given clone set $\langle d, h, f, c \rangle$.

Pairwise Measures: Table 4 we compute the PAIRWISE-JACCARD by considering subsets I with $|I| = 2$. The pairwise measures are presented for a range of different clone sets to give a sense of measure dependencies on clone quantities d, h .

| Jaccard similarity coefficient | | | | | | | |
|------------------------------------|------------------------------|---------|---------|---------|------|------|------|
| clone-class $\langle d, h \rangle$ | | binary | | | | | |
| $\langle 10, 0.25 \rangle$ | $\langle 1000, 0.25 \rangle$ | | | | | | |
| $\langle 10, 2.0 \rangle$ | $\langle 1000, 2.0 \rangle$ | duqu.45 | sutx.1e | stux.f8 | | | |
| duqu.0e | | 0.91 | 0.86 | 0.41 | 0.22 | 0.25 | 0.00 |
| | | 0.87 | 0.86 | 0.31 | 0.22 | 0.14 | 0.00 |
| duqu.45 | | | | 0.51 | 0.29 | 0.36 | 0.05 |
| | | | | 0.42 | 0.29 | 0.25 | 0.05 |
| stux.1e | | | | | | 0.57 | 0.05 |
| | | | | | | 0.47 | 0.05 |

Fig. 4. Jaccard coefficients for pairwise binaries in *Duqu-Stuxnet* data set for clone classes $\langle d, h, f, c \rangle$, with $f = 2, c = 2$. (b) Fractional amount of all data covered by a clone from $\langle d, e, 2, 2 \rangle$ for various values of d, e .

Conclusions: The Jaccard index applied to pairs of files such as in the experiment with *Stuxnet* and *Duqu* binary files may indicate shared provenance or present evidence that shared provenance is a candidate mode for binaries with unknown histories. In the above computation using clone class $\langle 1000, 2, 2, 2 \rangle$ the measure of 29% pairwise identity across the family boundary turns out to be a significant amount of clone features. Further the measure can be applied to incoming samples and measured against a known dictionary of examples. Results from 5 provides useful information on how to set parameters for dictionary matching.

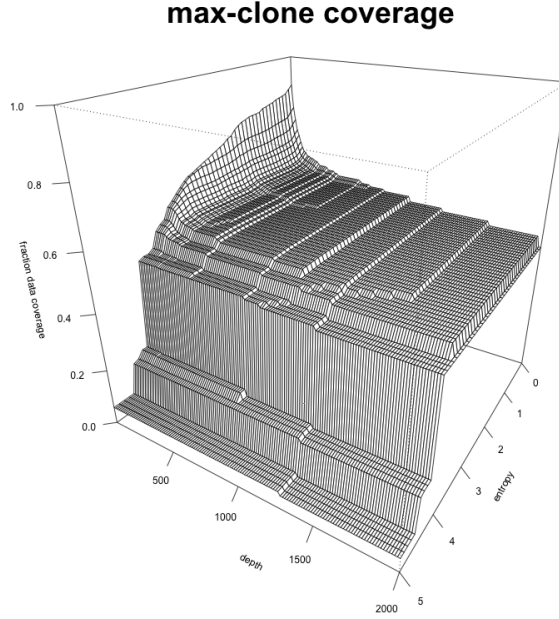


Fig. 5. Coverage: fractional amount of all data covered by a clone from $\langle d, e, 2, 2 \rangle$ for various values of d, e .

Set Algebra Measure: We construct a mixed data set including binaries from four malware families including: *Duqu*, *Poison Ivy*, *Stuxnet*, and *Zeus/Zbot*, and binaries from two operating systems: *Linux* and *Win7*. In table 6(a) we list the mixed data set used in this experiment. In the experiment we let $I = \{1, \dots, 16\}$ be the artifact index with artifact cluster identity withheld. Using the clone set $\langle 80, 0.6, 2, 2 \rangle$ we enumerate subsets of $A \subset I$ with non-empty $\text{COVER}(A)$. For each subset of $A \subset I$ we may compute the Jaccard similarity coefficient $J(A)$, and in Table 6(b) we present the rank ordering of the result.

Conclusions: The results reported in the experiment above indicate the usefulness of applying these measures to unknown data for triage or a first order pass to identify topics in data sets. While we defer a statistical treatment to a future effort the result above is useful in indicating the significance of the *Duqu*, *Stuxnet* comparison and also indicates the expectation of increased measures of clones in common in binaries chosen randomly from related activities.

Acknowledgments. We would like to thank the Members of Software Engineering Institute: Chuck Hines, Jeffrey Havrilla, Leigh Metcalf and Rhiannon Weaver for the many discussions about cyber security science. The research reported here was supported by CMU SEI line funded research program.

| binary artifact | id | artifact cluster |
|--------------------|----|-------------------|
| duqu.0e | 1 | <i>Duqu</i> |
| duqu.45 | 2 | |
| linux.bzip2 | 3 | <i>Linux</i> |
| linux.pwd | 4 | |
| linux.sed | 5 | |
| linux.su | 6 | |
| linux.tar | 7 | <i>Poison Ivy</i> |
| pi.0a..67 | 8 | |
| pi.0a..cf | 9 | <i>Stuxnet</i> |
| stux.1e..a5 | 10 | |
| stux.f8..1e | 11 | <i>Win7</i> |
| win7.calc | 12 | |
| win7.shutdown | 13 | |
| win7.soundrecorder | 14 | <i>Zeus</i> |
| zbot.20..f6 | 15 | |
| zbot.a8..8e | 16 | |

(a) Data

| $J(A)$ | subset $A \subset I$ | number of clones | comment |
|----------|-------------------------|---------------------|------------------------|
| 0.882959 | {1,2} | 7 | <i>Duqu</i> |
| 0.819336 | {8,9} | 10 | <i>Poison Ivy</i> |
| 0.268531 | {1,2,10} | 9 | <i>Duqu vs Stuxnet</i> |
| 0.122605 | {2,10} | 6 | |
| 0.077236 | {2,10,11} | 7 | <i>Stuxnet</i> |
| 0.076004 | {10,11} | 17 | |
| 0.036384 | {1,2,10,11} | 9 | <i>Duqu vs Stuxnet</i> |
| 0.028313 | {4,7} | 22 | <i>Linux</i> |
| 0.015117 | {3,4} | 2 | |
| 0.013848 | {3,5} | 4 | |
| 0.013570 | {3,6} | 2 | |
| 0.009218 | {5,6} | 1 | |
| 0.007921 | {3,4,5,6,7} | 4 | |
| 0.007230 | {13,14} | 5 | <i>Win7</i> |
| 0.004880 | {4,6} | 1 | <i>Linux</i> |
| 0.004104 | {1,2,11} | 1 | <i>Duqu vs stux</i> |
| 0.003735 | {4,6,7} | 2 | <i>Linux</i> |
| 0.003445 | {3,5,6,7} | 2 | |
| 0.003074 | {6,7} | 1 | |
| 0.002600 | {12,13,14} | 7 | <i>Win7</i> |
| 0.002521 | {12,13} | 6 | |

(b) Discovered Topics

Fig. 6. Experiment using random artifacts shows that the measure $J(A)$ for $A \subset I$ presents good recovery options for artifact triage. (a) Data set: a mixture of random samples from several distinct sets. (b) Rank by $J(A)$, (top 21 entries) with measure ≥ 0.025 shown.

Bibliography

- Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. *Replacing suffix trees with enhanced suffix arrays*. J. of Discrete Algorithms, 2(1):53–86, March 2004.
- Amihod Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. *Dynamic dictionary matching*. J. Comput. Syst. Sci., 49(2):208–222, October 1994.
- G.a Antoniol, U.b Villano, E.c Merlo, and M.a Di Penta. *Analyzing cloning evolution in the linux kernel*. Information and Software Technology, 44(13):755–765, 2002.
- Alberto Apostolico. *Pattern discovery and the algorithmics of surprise*. NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES, 183:111–127, 2003.
- Lars Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, 1996.
- Brenda S. Baker. *A theory of parameterized pattern matching: algorithms and applications*. In STOC, pages 71–80, 1993.
- Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. *A view on current malware behaviors*. In Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more, LEET’09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.
- Paul Bieganski, John Riedl, John V. Carlis, and Ernest F. Retzel. *Generalized suffix trees for biological sequence data: Applications and implementation*. In HICSS (5), pages 35–44, 1994.
- A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. *Complete inverted files for efficient text retrieval and analysis*. J. ACM, 34(3):578–595, July 1987.
- Eric Chien, Liam OMurchu, and Nicolas Falliere. *W32.duqu: the precursor to the next stuxnet*. In Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats, LEET’12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.
- Vin De Silva and Gunnar Carlsson. *Topological estimation using witness complexes*. In Proceedings of the First Eurographics conference on Point-Based Graphics, SPBG’04, pages 157–166, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.
- Nicolas Falliere, Liam O. Murchu, and Eric Chien. *W32.Stuxnet Dossier*. Technical report, Symantic Security Response, October 2010.
- Michael P. Ferguson. *Femto: Fast search of large sequence collections*. In CPM, pages 208–219, 2012.
- Paolo Ferragina and Roberto Grossi. *A fully-dynamic data structure for external substring search (extended abstract)*. In STOC, pages 693–702, 1995.

- Sébastien Ferré. *The efficient computation of complete and concise substring scales with suffix trees*. In Sergei O. Kuznetsov and Stefan Schmidt, editors, *Formal Concept Analysis, volume 4390 of Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 2007.
- Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
- Jiyong Jang, David Brumley, and Shobha Venkataraman. *Bitshred: feature hashing malware for scalable triage and semantic analysis*. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS '11*, pages 309–320, New York, NY, USA, 2011. ACM.
- Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. *Ccfinder: a multi-linguistic token-based code clone detection system for large scale source code*. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.
- Boojoong Kang, Taekeun Kim, Heejun Kwon, Yangseo Choi, and Eul Gyu Im. *Malware classification method via binary content comparison*. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium, RACS '12*, pages 316–321, New York, NY, USA, 2012. ACM.
- Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. *Linear work suffix array construction*. *J. ACM*, 53(6):918–936, November 2006.
- Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. *Linear-time construction of suffix arrays*. In *Proceedings of the 14th annual conference on Combinatorial pattern matching, CPM'03*, pages 186–199, Berlin, Heidelberg, 2003. Springer-Verlag.
- Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. *An empirical study of code clone genealogies*. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, September 2005.
- Pang Ko and Srinivas Aluru. *Space efficient linear time construction of suffix arrays*. In *J. of Discrete Algorithms*, pages 200–210. Springer, 2003.
- Arun Lakhotia, Andrew Walenstein, Craig Miles, and Anshuman Singh. *Vilo: a rapid learning nearest-neighbor classifier for malware triage*. *J. of Computer Virology and Hacking Techniques*, pages 1–15, 2013.
- Zhenmin Li, Shan Lu, Svada Myagmar, and Yuanyuan Zhou. *Cp-miner: a tool for finding copy-paste and related bugs in operating system code*. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
- S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. *Analysis of the linux kernel evolution using code clone coverage*. In *Mining Software Repositories*, 2007. ICSE Workshops MSR '07, pages 22–22, 2007.
- Udi Manber and Gene Myers. *Suffix arrays: a new method for on-line string searches*. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, SODA '90*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.
- Giovanni Manzini and Paolo Ferragina. *Engineering a lightweight suffix array construction algorithm*. *Algorithmica*, 40(1):33–50, 2004.

- Edward M. McCreight. A space-economical suffix tree construction algorithm. J. ACM, 23(2):262–272, April 1976.*
- Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM, 15(4):514–534, October 1968.*
- Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. ACM Comput. Surv., 39(1), April 2007.*
- Gonzalo Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33:2001, 1999.*
- Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. SCHOOL OF COMPUTING TR 2007-541, QUEEN’S UNIVERSITY, 115, 2007.*
- Esko Ukkonen. Finding approximate patterns in strings. J. Algorithms, 6(1):132–137, 1985.*
- Esko Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249–260, 1995.*
- Peter Weiner. Linear pattern matching algorithms. In Switching and Automata Theory, 1973. SWAT ’08. IEEE Conference Record of 14th Annual Symposium on, pages 1–11, 1973.*