Interaction Testing, Fault Location, and Anonymous Attribute-Based Authorization

by

Erin Lanus

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2019 by the
Graduate Supervisory Committee:

Charles J. Colbourn, Chair
Gail-Joon Ahn
Douglas C. Montgomery
Violet R. Syrotiuk

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

This dissertation studies three classes of combinatorial arrays with practical applications in testing, measurement, and security. Covering arrays are widely studied in software and hardware testing to indicate the presence of faulty interactions. Locating arrays extend covering arrays to achieve identification of the interactions causing a fault by requiring additional conditions on how interactions are covered in rows. This dissertation introduces a new class, the anonymizing arrays, to guarantee a degree of anonymity by bounding the probability a particular row is identified by the interaction presented. Similarities among these arrays lead to common algorithmic techniques for their construction which this dissertation explores. Differences arising from their application domains lead to the unique features of each class, requiring tailoring the techniques to the specifics of each problem.

One contribution of this work is a conditional expectation algorithm to build covering arrays via an intermediate combinatorial object. Conditional expectation efficiently finds intermediate-sized arrays that are particularly useful as ingredients for additional recursive algorithms. A cut-and-paste method creates large arrays from small ingredients. Performing transformations on the copies makes further improvements by reducing redundancy in the composed arrays and leads to fewer rows.

This work contains the first algorithm for constructing locating arrays for general values of d and t. A randomized computational search algorithmic framework verifies if a candidate array is $(\bar{d}, t)$-locating by partitioning the search space and performs random resampling if a candidate fails. Algorithmic parameters determine which columns to resample and when to add additional rows to the candidate array. Additionally, analysis is conducted on the performance of the algorithmic parameters to provide guidance on how to tune parameters to prioritize speed, accuracy, or a combination of both.

This work proposes anonymizing arrays as a class related to covering arrays with a higher coverage requirement and constraints. The algorithms for covering and locating arrays are tailored to anonymizing array construction. An additional property, homogeneity, is introduced to meet the needs of attribute-based authorization. Two metrics, local and global homogeneity, are designed to compare anonymizing arrays with the same parameters. Finally, a post-optimization approach reduces the homogeneity of an anonymizing array.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and seemingly endless patience of my advisor, Professor Charles Colbourn. His knowledge and guidance were instrumental to the completion of this research. I am grateful for his willingness to take a chance on me with my non-traditional background and his encouragement of my atypical approaches to problems. I could not have asked for a better mentor, and I hope that I am able to live up to the high standard he has set.

I also thank my dissertation committee - Professor Gail-Joon Ahn, Professor Douglas Montgomery, and Professor Violet Syrotiuk - for many years of advice and feedback, contributing to my development as both a student and a researcher. They each brought expertise in different areas to prompt ideas I would not have considered and, in doing so, helped shape the direction of this research.

I thank my family for their unwavering encouragement. Last, my research over the last five years has been supported by a fellowship from the National Physical Science Consortium.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

In this dissertation, we explore three problems in two areas within the computer security domain that appear unrelated, yet are solved by similar techniques due to their fundamental similarities. Interaction testing and fault location fall within the testing area of computer security. In an interaction testing scenario, it is required that interactions of factors up to a specified size are all covered by some test. Fault location requires the same coverage of interactions but with the additional requirement that sets of interactions be covered in different sets of tests to ensure that faulty interactions are distinguishable from non-faulty interactions. Anonymous attribute-based authorization is a problem within access control. When authorization is conducted on the basis of attributes instead of by presenting an identity, anonymous authorization is possible. That is, a subject can prove that she is authorized to use a resource without revealing her identity. For anonymous authorization to be meaningful, it is required that subjects are guaranteed some degree of anonymity. When a set of attributes of a certain size is presented, the system should not be able to identify the active subject from a set of other subjects who possess the same attributes.

Fundamentally, these three problems are all t-restrictions. A formal definition is given in [21]. Less formally, a t-restriction problem can be formulated as an $N \times k$ array with v_i symbols in column c_i where, for every way to select t of the columns, the set of demands is satisfied by the rows of the array. Covering arrays are t-restrictions used in interaction testing. The rows of the covering array represent tests, the columns represent factors, and the symbols represent factor levels. The demand for covering arrays is that every t-way interaction appears in some test.

Covering perfect hash families can be used to build covering arrays. These are also t-restrictions where, for every t-set of columns, some row contains a covering tuple. Locating arrays are extensions of covering arrays with the additional demand that sets of (at most) d interactions of size (at most) t appear in different sets of rows. The problem of anonymity in attribute-based authorization can also be formulated as a t-restriction. To address this problem, we propose a new combinatorial design, an anonymizing array, that ensures that every credential of size t in the array appears in either zero or at least r rows. The use of anonymizing arrays for attribute distribution guarantees that when policies are composed as conjunctions of (at most) t attributes, an authorized subject is granted access while being identified by the system with probability not greater than $\frac{1}{r}$.

While the applications are different, each problem can be formulated as a t-restriction. Therefore, techniques that work for one problem can often be applied to the others. Specifically, we are interested in efficient computational constructions for these arrays. That is not to say the same solution works for all three. Each application has specific features that map to different demands for the t-restriction, resulting in arrays with different properties. The general technique must be tailored to the specific problem.

We present the necessary background for interaction testing in § 1.1, fault location in § 1.2, and anonymous attribute-based authorization in § 1.3. We develop constructions for covering arrays via Sherwood covering perfect hash families in Chapter 2 and constructions for locating arrays in Chapter 3. Anonymizing arrays are proposed for the first time, so we develop definitions, metrics for comparing anonymizing arrays, and algorithms for constructing and improving anonymizing arrays in Chapter 4. The topic chapters are devoted to the specifics of each t-restriction problem. We present conclusions from each problem as well as a discussion of their commonalities in Chapter 5.

1.1 Interaction Testing

Testing of software and hardware systems for faults is a major undertaking. In addition to testing single components, also called factors, some faults may only occur when a combination of factors are set to specific levels, called an interaction [37]. The number of interactions grows rapidly as problem parameters increase, and in many practical scenarios, exhaustive testing of all combinations is not possible [45]. Most faults are caused by interactions of only a few factors, typically at most four to six, and testing all t-way combinations of factors is often sufficient [35, 37]. Systematically testing for failing interactions is the focus of combinatorial testing. Covering arrays are combinatorial designs used to construct test suites for interaction testing that ensure every t-way interaction is covered by some test. Yet, the number of tests in a covering array grows logarithmically in the number of factors [26]. Since each test incurs a cost and the test combination space can become infeasibly large for exhaustive testing, the goal in constructing covering arrays is typically to achieve the desired level of coverage in the fewest tests possible [45].

Formally, a *covering array* $\mathsf{CA}(N;t,k,v)$ of *strength* t is an $N \times k$ array on v symbols where every t-way interaction appears in at least one row with $t \leq k$. A t-way *interaction* is a t-tuple of pairs where the first element of the pair is a column index and the second element is a symbol. If $(c_1, ..., c_t)$ with $c_i \in \{1, ..., k\}$ is a tuple of column indices and $(\sigma_1, ..., \sigma_t)$ with $\sigma_i \in \Sigma, |\Sigma| = v$ is a tuple of symbols over the array alphabet Σ, then the tuple $\{(c_i, \sigma_i) : 1 \leq i \leq t\}$ is a t-way interaction. In other words, a covering array is an $N \times k$ array where every $N \times t$ subarray contains each of the v^t t-tuples of v symbols at least once. In the context of interaction testing, the columns are factors, the symbols are factor levels, the rows are tests, and the covering array itself is a test suite. A covering array ensures that every interaction of up to t factors appears in some test in the suite.

When factors do not share the same number of levels, the definition can be generalized to include different levels for each of the factors [27]. A *mixed-level covering array* $\mathsf{MCA}(N; t, k, (v_1, ...v_k))$ of strength t is an $N \times k$ array with v_i the number of levels for factor i where, for every $N \times t$ subarray, every combination of the levels for those factors appears at least once. When several factors j share the same number of levels v_i the exponential notation v_i^j is used. As testing some interactions may be dangerous, impossible, or not required, *constraints* may be placed on the covering array stating that some interactions must not occur.

The application of interaction testing requires that a test suite be explicitly constructed, yet covering array construction can be hard. Approaches can be categorized as random, greedy, heuristic search, and mathematical [45]. Aside from purely random construction, few approaches are feasible for constructing covering arrays for large numbers of factors. Both options for random construction – selecting a fixed number of rows and constructing an array that is a covering array with high probability, and choosing rows at random until all interactions are covered – can lead to many more rows than needed, and the first does not guarantee to result in a covering array. The computational approaches, greedy and heuristic search, build the array iteratively, moving through intermediate arrays or building one piece at a time. Greedy algorithms typically build one row or one column at a time by seeking to maximize new coverage. Heuristic search techniques typically begin with a candidate array and modify the array informed by the particular heuristic until it achieves full coverage. Mathematical methods use features of the underlying algebra and are categorized as direct constructions that make the entire array at once and recursive constructions that build the array from smaller ingredient pieces in a cut-and-paste approach or by column replacement [16]. Mathematical methods may be proven to find covering arrays within guaranteed bounds or provide smaller arrays than computational approaches. However, mathematical methods are typically limited to specific parameter values, such as certain values for factor levels. In contrast, computational approaches are generally applicable to a wide variety of parameters, but may not find the optimal solution, may not provide guarantees of how good the solution is, and may be inefficient for large numbers of factors or high strength.

Some approaches combine mathematical and computational methods. Covering perfect hash families (CPHFs) and Sherwood covering perfect hash families (SCPHFs), a restriction of CPHFs, are used as ingredients for column replacement via permutation vectors to construct covering arrays [54], and they provide the best asymptotic upper bounds on the number of rows for covering arrays and lead to efficient algorithms [50]. The problem of constructing a covering array reduces to the problem of searching for an SCPHF.

Formally, a *perfect hash family* $\mathsf{PHF}(N; k, v, t)$ is an $N \times k$ array on v symbols for which every $N \times t$ subarray contains at least one row where the t columns all contain distinct symbols. The t columns are said to be *separated* by this row. A covering perfect hash family is a PHF with the additional requirement that the t columns not only contain distinct symbols, but that the t symbols form a covering tuple.

Let $(\beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_{t-1}^{(i)})$ be the base v representation of symbol $i \in \{0, 1, \ldots, v^t-1\}$. A permutation vector $\overrightarrow{(h_0, h_1, \ldots, h_{t-1})}$ is the v^t length vector with symbol $\beta_0^{(i)} \times h_0 + \beta_1^{(i)} \times h_1 + \ldots + \beta_{t-1}^{(i)} \times h_{t-1}$ in position i for $0 \leq i \leq v^t - 1$ with arithmetic in $\mathsf{GF}(v)$. Table 1.1 gives an example of computing the $\overrightarrow{(1, 2, 0)}$ permutation vector for $v = 3, t = 3$. A set of t permutation vectors $T = \{(h_0^{(1)}, \ldots, h_{t-1}^{(1)}), \ldots (h_0^{(t)}, \ldots, h_{t-1}^{(t)})\}$ is *covering* if and only if the system of linear equations $S = \{(h_0^{(i)} \cdot \gamma_0) + \ldots + (h_{t-1}^{(i)} \cdot \gamma_{t-1}) = 0 : 1 \leq i \leq t\}$ with unknowns $\Gamma = \{\gamma_r : 0 \leq r \leq t-1\}$ does not have a

Table 1.1: Computing the $\overrightarrow{(1,2,0)}$ permutation vector, $v = 3, t = 3$

i	$(\beta_0, \beta_1, \beta_2)$	$\beta_0 \times h_0 + \beta_1 \times h_1 + \beta_2 \times h_2$	$\overrightarrow{(1,2,0)}$
0	000	0*1 + 0*2 + 0*0	0
1	100	1*1 + 0*2 + 0*0	1
2	200	2*1 + 0*2 + 0*0	2
3	010	0*1 + 1*2 + 0*0	2
4	110	1*1 + 1*2 + 0*0	0
5	210	2*1 + 1*2 + 0*0	1
6	020	0*1 + 2*2 + 0*0	1
7	120	1*1 + 2*2 + 0*0	2
8	220	2*1 + 2*2 + 0*0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
26	222	2*1 + 2*2 + 2*0	0

non-zero solution. That is, a covering set of t permutation vectors T is one in which the v^t t-tuples appear exactly once. If any t-tuple appears more than once, then at least one does not appear at all, and T is *noncovering*.

A *covering perfect hash family* $\mathsf{CPHF}(N; k, v^t, t)$ is an $N \times k$ array on v^t symbols, v a prime power, for which every $N \times t$ subarray contains a covering tuple in at least one row. A covering array $\mathsf{CA}(Nv^t; t, k, v)$ is constructed from the $\mathsf{CPHF}(N; k, v^t, t)$ by replacing each of the symbols in the CPHF with the corresponding v^t-length vector of v symbols. For each of the $\binom{k}{t}$ sets of t columns T, there is some row j of the CPHF that contains a covering tuple. The covering array contains all v^t combinations in the set of rows that results from the column replacement of row j for T.

As $\beta_1 = ... = \beta_{t-1} = 0$ for the first v entries of every permutation vector, the symbols in these entries are dependent upon β_0 alone. The subspace restriction that fixes $h_0 = 1$ forces the first v entries of every permutation vector to be the first v symbols. When a covering array is constructed from a CPHF where every permutation vector has $h_0 = 1$, then there are N repetitions of the v constant valued rows. $N - 1$ of these are redundant and can be removed. A *shortened permutation vector* of length

c_1	c_2	c_3	c_4	c_5
00	10	01	11	20
00	10	20	01	01

Figure 1.1: Example $\mathsf{SCPHF}(2; 5, 3^2, 3)$

$v^t - v$ is obtained by removing the first v entries of a permutation vector where $h_0 = 1$. For shortened permutation vectors, the definition of covering is modified to exclude the requirement that the constant t-tuples appear. A *Sherwood covering perfect hash family* $\mathsf{SCPHF}(N; k, v^{t-1}, t)$ is an $N \times k$ array on v^{t-1} symbols for which every $N \times t$ subarray contains a covering tuple in at least one row. An $\mathsf{SCPHF}(N; k, v^{t-1}, t)$ is used to construct a $\mathsf{CA}(N(v^t - v) + v; t, k, v)$ by replacing each symbol in the SCPHF with the corresponding shortened permutation vector, and appending v constant rows, one for each of the symbols in $\mathsf{GF}(v)$.

Figure 1.1 gives an example $\mathsf{SCPHF}(2; 5, 3^2, 3)$. Table 1.2 shows the shortened permutation vectors corresponding to the symbols in columns c_1, c_2, and c_5 of row 1. The constant rows that are removed as shown in gray. The repeated tuples indicate that this not a covering tuple. Table 1.3 shows the shortened permutation vectors for these same columns in row 2 of the SCPHF. Every tuple appears, indicating that this tuple is covering.

The density algorithm is a greedy one-row-at-a-time computational approach for searching for PHFs [15]. In this approach, a partial PHF is assumed, and the *density* of a partial row is defined as the number of not-yet-separated t-sets of columns a row is expected to separate if the row were completed uniformly at random. A column in a partial row that has been assigned a symbol is *fixed* and one that has not is *free*. A *partial row* is one that has some free columns. The density of a row containing only free columns can be computed and each time a column is selected to be fixed, there

Table 1.2: Noncovering tuple for $v = 3, t = 3$ with shortened permutation vectors

i	$\overrightarrow{(00)}$	$\overrightarrow{(10)}$	$\overrightarrow{(20)}$
0	0	0	0
1	1	1	1
2	2	2	2
3	0	1	2
4	1	2	0
5	2	0	1
6	0	2	1
7	1	0	2
8	2	1	0
9	0	0	0
10	1	1	1
11	2	2	2
12	0	1	2
13	1	2	0
14	2	0	1
15	0	2	1
16	1	0	2
17	2	1	0
18	0	0	0
19	1	1	1
20	2	2	2
21	0	1	2
22	1	2	0
23	2	0	1
24	0	2	1
25	1	0	2
26	2	1	0

is some choice of symbol for that column that does not reduce the row density. The algorithm described in [15] goes a step further and chooses the symbol that maximizes the density. Guaranteeing to do at least as well as the average allows this algorithm to provide an upper bound on the number of rows for the PHF.

An SCPHF provides a compact representation of a covering array. This covering array construction reduces the problem to finding an SCPHF and search techniques here may be more efficient by reducing the search space, allowing for covering arrays of higher strength and larger numbers of symbols and factors to be constructed.

Table 1.3: Covering tuple for $v = 3, t = 3$ with shortened permutation vectors

i	$\overrightarrow{(00)}$	$\overrightarrow{(10)}$	$\overrightarrow{(01)}$
0	0	0	0
1	1	1	1
2	2	2	2
3	0	1	0
4	1	2	1
5	2	0	2
6	0	2	0
7	1	0	1
8	2	1	2
9	0	0	1
10	1	1	2
11	2	2	0
12	0	1	1
13	1	2	2
14	2	0	0
15	0	2	1
16	1	0	2
17	2	1	0
18	0	0	2
19	1	1	0
20	2	2	1
21	0	1	2
22	1	2	0
23	2	0	1
24	0	2	2
25	1	0	0
26	2	1	1

The search technique in [54] uses backtracking on a candidate array initialized with constraints to reduce the search space and included a number of efficiency improvements. This technique resulted in several new covering arrays that outperformed arrays constructed by other methods. However, the initialization still exhibits significant randomness, and the size of the resulting array depends on both the structure of the initial array and the amount of execution time afforded to backtracking. The disadvantage of constructing covering arrays from SCPHFs is that it imposes the restriction that the number of symbols in each column must be a prime power and uniform across the entire array.

1.2 Fault Location

Covering arrays are used to detect the existence of faults involving t or fewer factors, but they are not guaranteed to indicate the specific interactions causing a fault. Each interaction is only guaranteed to be present somewhere in the test suite, but it may be the case that several interactions are present only once and in the same failing test. In that case, it is not possible to determine which is faulty from the test results. Consider the problem of testing a system of attribute-based access control rules [34]. Access control rules in disjunctive normal form are satisfied by sets of attributes, resulting in a "grant" decision. If no rule is satisfied by a set, a "deny" decision results. A test is a presentation of a set of attributes to the system which results in a "grant" or a "deny" decision. Suppose two sets of attributes that satisfy some rule are both present in the same test and the rule for only one of them fails. The outcome for the test is still "grant" due to the other set. Therefore, the fault caused by the failing set is masked by the other set.

To locate the failing set, a "pseudo-exhaustive" method is proposed using a combination of two arrays, G_{TEST} and D_{TEST} [34]. G_{TEST} is constructed to ensure that every satisfying set of attributes occurs in its own row, separate from any other satisfying set. D_{TEST} is constructed so that every set of attributes up to the strength t that is not satisfying appears in some row. That is, D_{TEST} is a covering array where every non-satisfying set of attributes appears and the satisfying sets are constraints. Each test in G_{TEST} should result in a "grant" and each test in D_{TEST} should result in a "deny." Testing the system with both arrays locates a faulty satisfying set, but failures by non-satisfying sets can still mask each other, requiring additional testing. This example demonstrates the applicability of combinatorial design solutions within the attribute-based access control domain and exhibits the type of solution desired for fault location. It also shows that an arbitrary covering array alone does not automatically provide fault location without additional properties.

The first subproblem of fault localization – identifying failure-inducing combinations – is concerned with the same issue of identifying the location of a fault [36]. A general strategy is to run a set of tests and obtain the pass/fail status, after which one of several strategies follow: 1) testing each interaction of a failing test separately, 2) creating classification trees to estimate the likelihood that an interaction is faulty, 3) generating new sets of tests based on "suspiciousness" of the interactions in the failing tests. An example of the third scenario is implemented in BEN [24], an approach that ranks interactions based on suspiciousness of the components of the interaction, suspiciousness of the interaction itself, and suspiciousness of the environment. New tests are generated and executed for the most suspicious interactions, and interactions that appear in new tests that pass are removed from the set of suspicious interactions. This process is repeated iteratively until it reaches a stopping condition. An approach combining test generation and fault localization iteratively generates and executes one test at a time, entering a fault localization mode if the test fails, until the desired coverage is attained [47].

Some testing scenarios are not suited to adaptive testing. If environmental variables cannot be well controlled, such as in the testbed evaluation of wireless network conferencing in [52], additional tests run later may introduce uncontrollable sources of variation. In this case, all tests need to be known in advance, and a complete test suite designed to locate faults non-adaptively is desirable.

Colbourn and McClary proposed locating arrays as combinatorial objects to locate faults non-adaptively [20]. Similar to covering arrays, locating arrays have the additional property that if the set of tests that covers a set of d t-way interactions is identified, no other set of d t-way interactions are tested in exactly the same tests. A test suite designed from a locating array indicates which interactions are faulty if there are no more than d faults. Correctly estimating the number of faults expected to exist is an additional challenge. Locating arrays have practical application, so in addition to the theoretical work providing mathematical constructions for restricted cases, explicit constructions that work for a wide variety of problem instances are needed.

A *locating array* $\mathsf{LA}(N; d, t, k, v)$ is an $N \times k$ array on v symbols. As with covering arrays, t is the *strength*. When the number of symbols for each column is not uniform, a locating array with v_i levels for the ith column, $i \in \{1, ..., k\}$, is a *mixed-level locating array* (MLA). The list of symbols $(v_1, v_2, ..., v_k)$ is the *type* [20] and the exponential notation v_i^j can be used when j factors all have v_i levels.

Let $\mathcal{I}_t$ be the set of all t-way interactions. Let $\rho(T)$ denote the rows in which interaction T appears, $T \in \mathcal{I}_t$, and define $\rho(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \rho(T)$. An array is (d, t)-locating if $\rho(\mathcal{T}_1) = \rho(\mathcal{T}_2) \iff \mathcal{T}_1 = \mathcal{T}_2$ whenever $\mathcal{T}_1, \mathcal{T}_2 \in \mathcal{I}_t$ and $|\mathcal{T}_1| = |\mathcal{T}_2| = d$. If an array is (d, t)-locating, the set of rows for every unique set of d t-way interactions is unique. The definition can be extended to allow for sets of size at most d, denoted by $\bar{d}$ and interactions of size at most t, denoted by $\hat{t}$.

We use the notation $\mathsf{LA}(d, t, k, v)$ to describe the class of arrays that are (d, t)-locating and $\mathsf{LA}(N; d, t, k, v)$ when a particular instance of a locating array is constructed with N rows. We use similar notation for covering arrays, e.g. $\mathsf{CA}(t, k, v)$, and $\mathsf{CA}(N; t, k, v)$.

A small example MLA with 10 rows and 3 columns that is (1,2)-locating is in Table 1.4. The rows for each of the interactions are shown in Table 1.5. One can check that every 2-way interaction appears in a unique set of rows. The MLA is not (2,2)-locating as $\rho(\{\{(c_1, 0)(c_2, 0)\}, \{(c_2, 0)(c_3, 0)\}\}) = \rho(\{\{(c_1, 0)(c_2, 0)\}, \{(c_1, 1)(c_3, 0)\}\}) = \{2, 7, 8\}$.

Table 1.4: $\mathsf{MLA}(10; 1, 2, 3, (2, 2, 3))$

	c_1	c_2	c_3
1	1	1	2
2	0	0	0
3	1	0	2
4	0	1	0
5	1	0	1
6	1	1	1
7	1	0	0
8	0	0	2
9	0	1	2
10	0	1	1

Table 1.5: Rows of 2-way interactions in $\mathsf{MLA}(10; 1, 2, 3, (2, 2, 3))$

	c_1, c_2	c_1, c_3	c_2, c_3
00	{2,8}	{2,4}	{2,7}
10	{3,5,7}	{7}	{4}
01	{4,9,10}	{10}	{5}
11	{1,6}	{5,6}	{6,10}
02		{8,9}	{3,8}
12		{1,3}	{1,9}

Locating arrays for fault location are fairly new and few techniques have been explored for their construction. Colbourn and McClary give necessary conditions for locating arrays to exist, detail relationships between variations of locating arrays and similar combinatorial objects called detecting arrays, and give a construction for a $(1, t)$-locating array from a covering array of strength $t + 1$, as well as a construction for a (d, t)-locating array from a covering array of strength $t + d$. They prove that every $\mathsf{CA}(N; t + d, k, v)$ is (d, t)-detecting and every (d, t)-detecting array is $(\bar{d}, t)$- and (d, t)-locating.

In these constructions, the higher strength provides a "witness," a column not involved in the coverage of an interaction but that "separates" instances of the interaction from instances of other interactions involving some of the same columns. For example, suppose a strength two locating array is derived from a strength three covering array and includes the subarray below with three columns and two rows.

c_0	c_1	c_2
0	0	0
0	0	1

Symbols in column c_2 serve as the witness for the location of $(t-1)$-way interactions involving other columns. That is, any interaction containing the 1-way interaction $\{(c_2, 0)\}$ cannot have exactly the same rows as the interaction $\{(c_0, 0), (c_1, 0)\}$ as this 2-way interaction also appears with $\{(c_2, 1)\}$ to satisfy coverage for $t = 3$. Then $\rho(\{(c_2, 0)\})$ and $\rho(\{(c_2, 1)\})$ are necessarily disjoint. When a $(1, t-1)$-locating array is built from a strength t covering array, however, this may use more rows than required, as there are v "witnesses" due to the repetition of the lower strength interactions appearing with each of the symbols of the higher strength interaction, not just the one witness needed. Constructing locating arrays from covering arrays of higher strength can be useful as a standard for comparison, as discussed in § 3.2.2.

The rest of the available literature consists of bounds and constructions for a restricted number of faults, i.e., a certain value for d. Denote by $\mathsf{LAN}(d, t, k, v)$, the locating array number, the fewest rows for which a (d, t)-locating array can exist. Colbourn, Fan, and Horsley determine precisely $\mathsf{LAN}(1, 1, k, v)$ [11]. Colbourn and Fan develop three recursive constructions for $(1, \bar{2})$-locating arrays using $(1, \bar{2})$-locating array, $(1, \bar{1})$-locating array, and strength three covering array ingredients, all on the same number of symbols v [17].

Lower bounds for $\mathsf{LAN}(1, t, k, v)$ when $k \geq t \geq 2$ and $v \geq 2$ are known, as are optimal constructions (where the number of rows meets the bound) for some infinite series of locating arrays based on design theory principles when the number of symbols v in the array meets certain restrictions [60]. The lower bound is used as a starting point to find instances of (1,2)-locating arrays defined by constraints fed to a SAT solver, and the method is useful when the size of the locating array is quite small; results are reported for $v = 2, k \in \{2, ..., 25\}$ and $v = 3, k \in \{2, ..., 13\}$ [32]. A greedy one-row-at-a-time algorithm constructs (1,2)-locating arrays by computing $\rho(T)$ for all $\binom{k}{t}v^t$ interactions simultaneously, assigning interactions i and j to the same equivalence class when $\rho(i) = \rho(j)$, and then repeatedly creating new rows to distinguish interactions until all equivalence classes contain a single item; results are reported for $v \in \{2, 5, 10\}$ and $k \in \{5, 10, 15, 20\}$ [43]. A lower bound of $2v^t$ exists for $(\bar{2}, t)$-locating arrays and can be met when constructed from orthogonal arrays with certain defined properties [55].

Locating arrays are proposed as designs for screening experiments to identify significant effects when the number of design points of the full factorial becomes infeasible, as locating arrays limit confounding of main effects and interactions up to strength t by requiring that non-identical sets of interactions do not appear in the same sets of rows [1, 22]. Due to experimental error, missing or outlier responses, requiring that interaction sets be separated by more than one test allows location. The separation *distance*, δ, between two sets of d t-way interactions is the number of tests in which interactions from one or the other but not both sets appear. Seidel, Sarkar, Colbourn, and Syrotiuk implemented conditional expectation, column resampling, and local optimization algorithms to construct arrays that are $(\bar{1}, \bar{2})$-locating with $\delta \leq 4$ [52]. Conditional expectation is too storage and time intensive for large numbers of factors, and results from conditional expectation are reported only up to type 3^{20}. Comparison against the column resampling and local optimization results for two mixed-level locating arrays appears in § 3.2.3. We direct interested readers to surveys on the state of locating array research and its applications [12, 13].

Locating arrays have practical application for fault location in combinatorial testing, but no previous approach has proposed to build locating arrays for general values of d and t, particularly for mixed-level locating arrays.

1.3 Attribute-Based Authorization

Attribute-based access control (ABAC) is a logical access control model wherein access control decisions are made on the basis of attributes. The National Institute of Standards and Technology (NIST) defines ABAC as "an access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions" [28]. Attributes are characteristics of a subject expressed as name-value pairs and may be based on the subject's real-world identity (e.g., $name = Martha$, $age = 34$) or in-system characteristics (e.g., $role = administrator$). ABAC can be configured to support policies available under other models of access control such as discretionary access control, mandatory access control, and role-based access control (RBAC) [28]. One feature that makes ABAC attractive is that it can be used as global access control on a heterogeneous system where subsystems are each employing a different access control model. Similarly, hybrid models have been proposed, such as adding attributes to RBAC [33].

Unlike other access control models, ABAC is inherently identity-less. Determining whether a subject is authorized is characteristically done on the basis of the attributes the subject possesses and does not require knowledge of the identity of the subject. This may greatly simplify privilege management. Making decisions on the basis of attributes rather than identities removes the need to update access control lists or capability lists to add or remove privileges related to individual subjects. Policies can be written dynamically to achieve fine-grained access control without creating roles for each possible subset of users in advance, resulting in “role explosion” [33]. ABAC is desirable in systems with a large number or frequently changing set of users.

Attribute-Based Encryption (ABE) is another general attribute-based method. ABE is first introduced as Fuzzy Identity-Based Encryption (FIBE) where identities are sets of attributes. In FIBE, if an identity matches on d or more of the attributes used to encrypt a ciphertext, the private key affiliated with the identity can be used to decrypt [49]. The access structure employed by FIBE is restricted to a threshold gate the size of which is fixed at setup time. In Key-Policy ABE, private keys are access control policies and ciphertexts are encrypted over attributes [25]. A private key that contains a policy that is satisfied by the attributes of a ciphertext can be used to decrypt the ciphertext. Conversely, Ciphertext-Policy ABE (CP-ABE) encrypts ciphertexts with access control policies [3]. A private key that contains attributes that satisfy the policy can be used to decrypt.

CP-ABE has been proposed to mediate authenticated key exchange [48]. As a simplified example, suppose the service encrypts a session key in a policy and broadcasts the message. A subject that has a private key containing attributes that satisfy the policy is able to decrypt the message and obtain the key. The subject begins communicating with the service by encrypting messages in the session key. The service now knows that the subject communicating via the session key is authorized based on possession of the required attributes to obtain the key.

A feature of attribute-based systems is that they can be used to achieve anonymous access control, granting access to authorized subjects and denying access to unauthorized subjects without knowledge of the identity of the subject. This feature is natural to systems implementing pure ABAC as “in many models, access requests are not conducted directly by the user but indirectly through a session that may contain a subset of the user's attributes" [53]. In some instances, the subject uses an application that allows the subject to choose which attributes to present to the system. In others, a credential such as a card preloaded with a subset of attributes represents the subject to the system. Unless the subject's identity is used as an attribute, anonymous access control is a possible byproduct of ABAC.

While often desirable to subjects, anonymity is not always considered to be a feature due to the potential conflict between anonymity and auditability. Whether these two goals conflict is dependent upon the definition of auditability for the system. One definition of auditability "the ability to easily determine the set of users who have access to a given resource or the set of resources a given user may have access to (sometimes referred to as a 'before the fact audit')" makes it possible to have anonymity and auditability [53]. ABAC systems may not be well suited for easily computing the set of users capable of completing an action given a policy. However, the ability to compute these sets does not require the system to track the actions performed by subjects. Some systems may make it difficult or impossible to determine the set depending on how attributes are managed, such as in instances of multiple attribute-granting authorities or lack of requiring subjects to register their attributes with the system. Yet, fixing this problem for auditability does not necessitate a loss of anonymity. In our proposal, all attributes of all subjects are registered with the system. We can achieve both this definition of auditability and anonymity concurrently. If the goal of auditability is to be able to identify the subject who completed an operation, anonymity against the system and auditability are necessarily at odds. The purpose of this work is to perform attribute distribution to achieve a specified degree of anonymity. A dual outcome is that we also show when anonymity of the specified degree cannot be achieved.

Whether anonymity is achieved and even the kind of anonymity intended is typically dependent upon the ABAC implementation. One approach is to provide certificates to prove possession of attributes. Still, transactions involving the same certificates may be linked, the certificates may require the subject's public key certificate, or the approach may require sending all certificates for potentially relevant attributes [2]. The solution in [2] employs cryptographic zero knowledge proofs to allow subjects to prove possession of just the requisite attributes without revealing additional information. ABE provides another mechanism for anonymous authorization. A system employing CP-ABE, LOCATHE, includes a mode in which the subject does not provide its identity to the system [48]. Instead, the service knows the subject is authorized based on possession of attributes. The authors of LOCATHE make the stronger claim that the service cannot uniquely identify the subject.

While attribute-based authorization decisions typically do not require knowledge of the identity of the subject in order to make an authorization decision, this is not a guarantee that the identity cannot be deduced. For example, in the LOCATHE system, the subject must register with the service running LOCATHE to receive a key, and thus the service knows all of the users in the system as well as their attributes. If a policy can be composed so that only one subject can satisfy the policy and this policy is used to encrypt the session key, the service knows with certainty the identity of the subject communicating with it. "Anonymous ABE" uses hidden credentials which can be used to retrieve a session key anonymously, but the receiver anonymity is based on "plausible deniability" due to the fact that anyone can request the message, not just the intended recipient [31]. Plausible deniability fails if the message is decrypted to gain a session key used to authenticate or obtain authorization, as this proves that someone with the correct credentials decrypted the message.

k-anonymous attribute-based access control ensures that if a subject submits a set of attributes to a service, k previous requests with identical attributes are stored with the service [56]. The protocol involves negotiation between a “credential submitter” and “policy enforcer.” The “bootstrapping” phase of the protocol builds up the set of “attribute assertions” that serve as previous requests.

Chapter 2

INTERACTION TESTING

Covering arrays are combinatorial designs used to construct test suites for interaction testing, and they can be constructed from covering perfect hash families (CPHFs) by column replacement. CPHFs serve as compact representations of covering arrays and lead to efficient search algorithms. Sherwood covering perfect hash families (SCPHFs) employ a subspace restriction to reduce the number of rows in the final covering array. Necessary background and definitions for this chapter are provided in § 1.1 of the introduction. The contribution of the current work is to develop a greedy conditional expectation algorithm to efficiently construct intermediate-size covering arrays via SCPHFs. Additionally, two recursive algorithms that operate on small to intermediate-size SCPHF ingredients are used to construct larger covering arrays. This chapter is organized as follows. The conditional expectation algorithm is developed in § 2.1 and the recursive algorithms are in § 2.2. Results are presented in § 2.3.

2.1 Sherwood Covering Perfect Hash Family Construction

2.1.1 Conditional Expectation (CE) Algorithm

In this section, we develop a greedy, one-row-at-a-time conditional expectation (CE) algorithm to construct SCPHFs inspired by the density algorithm for PHFs [15]. CE is based on the fact that the number of noncovering tuples can be computed. That is, $nc_t(v)$ is the number of distinct, ordered noncovering tuples given t and v. How to count distinct noncovering tuples is discussed in [54]. For $t = 3$, $nc_3(v)$ is computed exactly given v as $nc_3(v) = 3!\binom{v}{3} \times (v^2 + v)$. For $t = 4$, an upper bound is easily computed, $nc_4(v) < 4!\binom{v^2}{4} \times (v^3 + v^2 + v)$, and a general upper bound on $nc_t(v)$ is given for larger t. For the purposes of CE, $nc_t(v)$ is computed exactly for t and v by explicitly constructing the list of noncovering tuples. The algorithm repeatedly checks whether tuples are covering, so precomputation is more time efficient than computing whether a tuple is covering each time one is checked. The number of noncovering tuples is smaller than the number of covering tuples, so it is more efficient to store the list of noncovering tuples. An optimization to reduce the storage for the noncovering list is discussed towards the end of this section.

The probability of selecting a covering tuple when symbols are chosen uniformly at random is computable and so is the expected number of previously uncovered t-sets covered by a random row. The probability of selecting a covering tuple when t of the v^{t-1} symbols are chosen at random is

$$P_{=}\frac{(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1)-nc_t(v)}{(v^{t-1})^t}.$$

CE builds an SCPHF one row at a time. Suppose some rows are completed and row ρ is a new row with all columns free. For a t-set of columns $T \subseteq C, |T| = t$, write $\lambda(T) = 0$ if there is a covering tuple on T in some completed row; otherwise, write $\lambda(T) = 1$. Write $value(\rho)$ for the number of t-sets covered for the first time by ρ. Before any columns are fixed, the probability that a particular t-set is covered is P. Therefore, its expectation is $\lambda(T)P$. By linearity of expectation, the expectation of the sum is the sum of the expectations, so the expected $value(\rho)$ is

$$E[value(\rho)] = \sum_{T\subseteq C} \lambda(T)P.$$

In a partial row, when the symbols in the fixed columns have been chosen to meet or exceed the expectation, there is always a choice of symbol for a free column that does not reduce the expectation. CE computes the expectation for the symbols of a free column and selects one that meets or exceeds the expectation. When considering a free column c_i, only columns that are involved in some uncovered t-set with c_i contribute to the expectation for each of the possible symbols examined for c_i. These columns are the *remaining neighborhood of* c_i. Further, when all columns in a t-set are free, any choice of symbol while fixing the first column in the set results in the same expectation. Therefore, CE restricts consideration to the *partially fixed remaining neighborhood* of c_i, R_i, as the remaining neighborhood of column c_i excluding t-sets where all columns are free. Let P_T be the probability of selecting a covering tuple for the t-set T when T contains fixed and free columns. Define $value(R_i, c_i, \sigma) = \sum_{T \subseteq R_i} \lambda(T) P_T$, the expected number of t-sets covered for the first time if c_i is fixed to σ. This allows for local decisions without computing the expectation for the entire row for every symbol in a free column.

Most of the computation is determining the expected number of t-sets first covered by fixing a symbol, so improvements in efficiency of implementation made here have a substantial effect on practical time performance. These include:

1. *Standard first row.* Rather than using CE search to construct the first row, a standard first row fixes symbol $i \mod v^{t-1}$ in column c_i. Minimizing the number of repetitions of any one symbol provides reasonable avoidance of non-covering tuples without requiring any computation when $k > v^{t-1}$. The use of a standard first row drastically decreases search time when k is large in practice and covers a substantial number of t-sets up front, though typically fewer than the number found by search when k is smaller relative to v^{t-1}.

2. *Early exit.* As parameters v, t, and k increase, checking every symbol becomes time prohibitive. A threshold between 0 and 1 inclusive is a tunable parameter,

which is used to determine when the algorithm stops with an acceptable symbol. When $\frac{value(R_i,c_i,\sigma)}{|R_i|} \geq threshold$, σ is acceptable. As long as the threshold is at least P, the bounds hold. Setting $threshold = 1$ results in finding an optimal symbol.

3. *Choice of column to fix.* Given a partial array of strength three, some pair of columns c_i, c_j appears together in remaining neighborhoods of other columns most frequently. Placing the same symbol in c_i, c_j prevents coverage for all triples involving them. To avoid this, repeatedly select the most frequently occurring pair of columns and, if free, fix symbols for these columns in the usual way. This is followed by repeatedly selecting a free third column, c_ℓ so that c_i, c_j is in the remaining neighborhood of c_ℓ and fixing a symbol in c_ℓ. This optimization is not used in earlier conditional expectation methods. It is not clear how to apply this to strength four or if the effort spent on this look-ahead is worth the extra computation as opposed to choosing the next column to fix sequentially or randomly.

4. *Checking tuples for coverage quickly.* A tuple that contains a repeated symbol cannot be covering, so these are identified as noncovering immediately. There are more distinct covering tuples than distinct noncovering ones, so it is more efficient to compute and store the distinct noncovering tuples. A list uses the least storage, but sequential access requires looking at $\frac{nc_t(v)}{2}$ tuples on average. Instead, the tuples are stored in a $(v^{t-1})^t$ indicator matrix for direct access, with an index corresponding to each element in the t-tuple. To reduce the amount of memory needed by factor of v^{t-1}, distinct noncovering tuples are stored in a canonical form with the first element as 0. Given a tuple in noncanonical form, $([h_1^{(1)}, \ldots, h_{t-1}^{(1)}], \ldots, [h_1^{(t)}, \ldots, h_{t-1}^{(t)}])$, the additive inverse of the first tuple

element is added to each tuple element with arithmetic in $\mathsf{GF}(v)$ to obtain $([h_1^{(1)} - h_1^{(1)}, \ldots, h_{t-1}^{(1)} - h_{t-1}^{(1)}], \ldots, [h_1^{(t)} - h_1^{(1)}, \ldots, h_{t-1}^{(t)} - h_{t-1}^{(1)}])$. Because the first element is always 0 in canonical form, this index is omitted from the lookup array. This requires that the conversion always maps noncovering to noncovering and covering to covering, which we prove in Theorem 1.

Theorem 1. *The classes of noncovering and covering tuples are closed under arithmetic when performed component-wise in $\mathsf{GF}(v)$.*

Proof. The classes of noncovering and covering tuples are closed under addition, subtraction, multiplication, and division when performed component-wise in $\mathsf{GF}(v)$ as shown in the following three lemmas. □

Lemma 1.1. *The classes of noncovering and covering tuples are closed under multiplication.*

Proof. A set of t shortened permutation vectors $T = \{(h_1^{(1)}, \ldots, h_{t-1}^{(1)}), \ldots, (h_1^{(t)}, \ldots, h_{t-1}^{(t)})\}$ is covering if and only if the system of linear equations $S = \{\gamma_0 + (h_1^{(i)} \cdot \gamma_1) + \ldots + (h_{t-1}^{(i)} \cdot \gamma_{t-1}) = 0 : 1 \leq i \leq t\}$ with unknowns $\Gamma = \{\gamma_j : 0 \leq j \leq t-1\}$ does not have a nonzero solution [54]. Define

$$S_{(\mu_0,\mu_1,\ldots,\mu_{t-1})} = \{\mu_0 \cdot \gamma_0 + (h_1^{(i)} \cdot \mu_1) \cdot \gamma_1 + \ldots + (h_{t-1}^{(i)} \cdot \mu_{t-1}) \cdot \gamma_{t-1} = 0 : 1 \leq i \leq t\}$$
$$T_{(\mu_1,\ldots,\mu_{t-1})} = \{(h_1^{(1)} \cdot \mu_1, \ldots, h_{t-1}^{(1)} \cdot \mu_{t-1}), \ldots, (h_1^{(t)} \cdot \mu_1, \ldots, h_{t-1}^{(t)} \cdot \mu_{t-1})\}$$

where $0 < \mu_j < v$. That is, $S_{(\mu_0,\mu_1,\ldots,\mu_{t-1})}$ is the system S with each coordinate j of each equation multiplied by μ_j, and $T_{(\mu_1,\ldots,\mu_{t-1})}$ is the set of transformed permutation vectors. Assume that T is covering and suppose that $T_{(\mu_1,\ldots,\mu_{t-1})}$ is noncovering. Then $S_{(\mu_0,\mu_1,\ldots,\mu_{t-1})}$ has a solution with an assignment of a nonzero value to some $\gamma_j \in \Gamma$. Then because $S_{(\mu_0,\mu_1,\ldots,\mu_{t-1})} = \{\mu_0 \cdot \gamma_0 + (h_1^{(i)} \cdot (\mu_1 \cdot \gamma_1)) + \ldots + (h_{t-1}^{(i)} \cdot (\mu_{t-1} \cdot \gamma_{t-1})) = 0 : 1 \leq i \leq t\}$, this solution provides a nonzero solution, $\gamma_j \mu_j$, for T. But this contradicts the assumption that T is covering. Therefore, the preimage for a noncovering tuple transformed by multiplication must be a noncovering tuple.

Now, suppose that T is noncovering, so there is a nonzero solution as an assignment of values to Γ. Then this same assignment serves as proof of a nonzero solution to $S_{(\mu_0,\mu_1,\ldots,\mu_{t-1})}$. Then $T_{(\mu_1,\ldots,\mu_{t-1})}$ must also be noncovering, so the image must be a noncovering tuple. Then the class of noncovering tuples is closed under multiplication. For every T and $\{\mu_j : 0 \leq j \leq t-1\}$ defined as above, there is a $T_{(\mu_1,\ldots,\mu_{t-1})}$. Covering tuples cannot map to noncovering tuples, and noncovering tuples cannot map to covering tuples, so the class of covering tuples must be closed under multiplication. $\square$

Lemma 1.2. *The classes of noncovering and covering tuples are closed under addition.*

Proof. Assume $T = \{(h_1^{(1)}, \ldots, h_{t-1}^{(1)}), \ldots, (h_1^{(t)}, \ldots, h_{t-1}^{(t)})\}$ is covering. Then $\Gamma = \{\gamma_j = 0 : 0 \leq j \leq t-1\}$ is the only solution. Define for $0 \leq \alpha_j < v$ and $1 \leq i \leq t$

$$\begin{aligned} T_{(\alpha_1,\ldots,\alpha_{t-1})} &= \{(h_1^{(1)} + \alpha_1, \ldots, h_{t-1}^{(1)} + \alpha_{t-1}), \ldots, (h_1^{(t)} + \alpha_1, \ldots, h_{t-1}^{(t)} + \alpha_{t-1})\} \\ S_{(\alpha_1,\ldots,\alpha_{t-1})} &= \{\gamma_0 + ((h_1^{(i)} + \alpha_1) \cdot \gamma_1) + \ldots + ((h_{t-1}^{(i)} + \alpha_{t-1}) \cdot \gamma_{t-1}) = 0\} \\ &= \left\{ \sum_{j=1}^{t-1} \gamma_j \alpha_j + \gamma_0 + (h_1^{(i)} \cdot \gamma_1) + \ldots + (h_{t-1}^{(i)} \cdot \gamma_{t-1}) = 0 \right\} \end{aligned}$$

Suppose $T_{(\alpha_1,\ldots,\alpha_{t-1})}$ is noncovering, so there is a nonzero solution. Then that assignment to Γ serves as proof of a nonzero solution to the system $S' = \{\gamma_0' + (h_1^{(i)} \cdot \gamma_1) + \ldots + (h_{t-1}^{(i)} \cdot \gamma_{t-1}) = 0 : 1 \leq i \leq t\}$ where $\gamma_0' = \sum_{j=1}^{t-1} \gamma_j \alpha_j + \gamma_0$. But this contradicts the assumption that T is covering. Therefore, the preimage for a noncovering tuple transformed by addition must be a noncovering tuple.

Now, suppose that T is noncovering, so there is a nonzero solution to S' as an assignment of values to Γ. Then this same assignment serves as a nonzero solution to $S_{(\alpha_1,\ldots,\alpha_{t-1})}$. Then $T_{(\alpha_1,\ldots,\alpha_{t-1})}$ must also be noncovering, so the image must be a noncovering tuple. Then the class of noncovering tuples is closed under addition.

For every T and α_j defined as above, there is a $T_{(\alpha_1,\ldots,\alpha_{t-1})}$. Covering tuples cannot map to noncovering tuples, and noncovering tuples cannot map to covering tuples, so the class of covering tuples must be closed under addition. $\square$

Lemma 1.3. *The classes of noncovering and covering tuples are closed under subtraction and division.*

Proof. Subtraction is defined in the field in terms of addition so that $a - b = a + (-b)$ where $-b$ is the unique element so that $(-b)+b = 0$; $(-b)$ is the negative of b. Division is defined in the field in terms of multiplication so that $a/b = a \cdot b^{-1}$ where b^{-1} is the unique element so that $b^{-1} \cdot b = 1$; b^{-1} is the inverse of b. Since both operations are defined in terms of operations under which the classes are closed, the classes are closed under these operations as well. $\square$

The CE algorithm is presented in Algorithm 1.

2.1.2 Conditional Expectation Bounds

CE provides an upper bound on the number of rows needed for the SCPHF when rows are constructed that cover at least as many new t-sets as the average. Let

$$\begin{aligned}\beta &= 1 - \frac{(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1) - nc_t(v)}{(v^{t-1})^t} \\ &= \frac{(v^{t-1})^t - (v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1) + nc_t(v)}{(v^{t-1})^t}\end{aligned}$$

be the probability that a particular t-set is *not* covered in a particular row. Then, β^N is the expected value of the "non-coverage" of a particular t-set after N rows are randomly created, where $\beta = 1$ indicates that it is not covered and $\beta = 0$ indicates that it is covered. There are $\binom{k}{t}$ t-sets to cover, so by linearity of expectation, the expected value of the "non-coverage" of all t-sets after N rows is $\binom{k}{t}\beta^N$. Setting this

Algorithm 1: Conditional Expectation (CE)

```
input  : $k$ columns, a set of $v^{t-1}$ symbols $\Sigma$, strength $t$, $threshold$
output: $\mathbf{A}, N$
begin
    Create an empty array $\mathbf{A}$
    Set $N = 0$
    Add all $\binom{k}{t}$ $t$-sets of columns to list $L$
    while $|L| > 0$ do
        Add a row $\rho$ with all columns free to $\mathbf{A}$
        for $c_i$ in the set of columns do
            Make an empty list $F$
            Compute $R_i \subseteq L$
            for $T \in R_i$ do
                if all $c_j \in T, c_j \neq c_i$ are fixed then
                    Add $T$ to $F$
            for $\sigma \in \Sigma$ do
                Compute $value(R_i, c_i, \sigma)$
                if $\frac{value(R_i, c_i, \sigma)}{|R_i|} \geq threshold$ then
                    Set $\mathbf{A}[\rho][c_i] = \sigma$
                    Break
            if $c_i$ is free then
                Set $\mathbf{A}[\rho][c_i] = \max_{\sigma \in \Sigma}(value(R_i, c_i, \sigma))$
            for $T \in F$ do
                if $T$ is covering then
                    Remove $T$ from $L$
        Increment $N$
    Return $\mathbf{A}, N$
```

quantity to be less than 1 (Equation 2.1) and solving for the smallest N for which the inequality holds gives a bound on the number of SCPHF rows after which all t-sets are expected to be covered (Equation 2.2). The upper bound in terms of rows in the covering array after column replacement is in Equation 2.3.

$$\binom{k}{t}\beta^N < 1 \tag{2.1}$$

$$\frac{\log\binom{k}{t}}{\log(\frac{1}{\beta})} = \frac{\log\binom{k}{t}}{\log\frac{(v^{t-1})^t}{(v^{t-1})^t-(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1)+nc_t(v)}} < N \tag{2.2}$$

$$\frac{\log\binom{k}{t}}{\log\frac{(v^{t-1})^t}{(v^{t-1})^t-(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1)+nc_t(v)}}(v^t-v)+v < N(v^t-v)+v \tag{2.3}$$

2.2 Recursive Algorithms

In this section, we develop two recursive algorithms, Composition and Affine Composition, both of which use an $\mathsf{SCPHF}(N;k,v^{t-1},t)$ ingredient array to construct an $\mathsf{SCPHF}(N+N';mk,v^{t-1},t)$.

2.2.1 Composition (COMP) Algorithm

In the Composition (COMP) algorithm, m identical copies of the ingredient array $\mathbf{A}$ are placed horizontally to form $\mathbf{A}'$, an $N \times mk$ array on v^{t-1} symbols. All t-sets located entirely within a subarray are covered as the ingredient is an SCPHF. Building the remaining list of uncovered t-sets occurring between subarrays does not require checking whether tuples are noncovering; this is known precisely from the column indices. The CE algorithm is employed to build the final N' rows to cover the remaining t-sets and complete the SCPHF.

The remaining list is straightforward to compute. Consider each column in $\mathbf{A}'$ as having an index computed from the subarray ℓ it is contained in and the column y

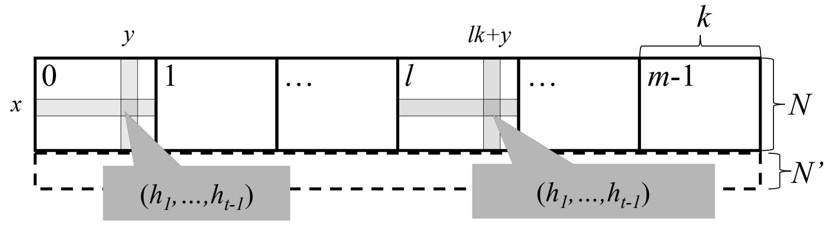


Figure 2.1: Duplicated columns of the ingredient array in an SCPHF by COMP

from the ingredient of which it is a replicate. The computed index is $\ell k + y$. (See Figure 2.1 for an illustration.) A t-set is covered if and only if the column portion of the index is distinct for all columns in the t-set. When these are not distinct, two columns in the t-set correspond to the same column from the ingredient, and identical columns cannot contain a covering tuple. When all columns of the t-set correspond to different columns from the ingredient, these must have a covering tuple in some row as they are covered somewhere in the ingredient.

The advantage of COMP is that it covers a large fraction of the t-sets of columns quickly with no additional work by using an ingredient array. Another advantage is that it is useful for comparing the bounds obtained for composition versus direct construction. COMP is presented in Algorithm 2.

2.2.2 Affine Composition (AF-COMP) Algorithm

The primary disadvantage of COMP is that the duplication caused by the k sets of m identical columns results in more remaining t-sets than necessary to be covered by CE. The Affine Composition (AF-COMP) algorithm attempts to avoid identical columns by applying affine transformations. (See Figure 2.2 for an illustration.) By Theorem 1, the classes of covering and noncovering tuples are closed under arithmetic in $\mathsf{GF}(v)$, and therefore, applying an affine transformation to an SCPHF yields an-

Algorithm 2: Composition (COMP)

```
input : $\mathbf{A}$ an $N \times k$ strength $t$ array on $v^{t-1}$ symbols, $m$ a number of copies
output: $\mathbf{A}'$ an $(N+N') \times km$ strength $t$ array on $v^{t-1}$ symbols, $N+N'$
begin
    Create an empty $N \times mk$ array, $\mathbf{A}'$
    for row $0 \leq x < N$ do
        for subarray index $0 \leq \ell < m$ do
            for column index $0 \leq y < k$ do
                Set $\mathbf{A}'[x][\ell k + y] = \mathbf{A}[x][y]$
    for $\binom{km}{t} - \binom{k}{t}m^t$ t-sets $T$ containing at least two columns with the same column index do
        Add $T$ to remaining list
    Run CE algorithm on $\mathbf{A}'$ to build the final $N'$ rows
    Return $\mathbf{A}, N + N'$
```

other SCPHF. In fact, the rows of an SCPHF are independent, so a different affine transformation can be applied to each row within an SCPHF. Given an $\mathsf{SCPHF}(k, v, t)$ with permutation vectors $\overrightarrow{(h_1^{(i)}, \ldots, h_{t-1}^{(i)})}$, an affine transformation is a choice of multipliers $1 \leq \mu_j < v$ and adders $0 \leq \alpha_j < v$, $1 \leq j \leq t-1$ that results in the permutation vector $\overrightarrow{(\mu_1 h_1^{(i)} + \alpha_1, \ldots, \mu_{t-1} h_{t-1}^{(i)} + \alpha_{t-1})}$. There are v choices for the adder and $v-1$ choices for the multiplier, and each pair of choices can be made for the $t-1$ coordinates of a permutation vector and N rows of the ingredient. Therefore, a single ingredient can result in $(v(v-1))^{N(t-1)}$ SCPHFs by affine transformation. Because it is unclear how to choose the "best" set of affine transformations, AF-COMP selects them randomly.

Each subarray is still an SCPHF, so t-sets within a subarray are covered. Constructing the first ingredient covers $\binom{k}{t}$ t-sets, and by composition, another $(m-1)\binom{k}{t}$ are covered "for free." However, it is no longer the case that t-sets with distinct indices with respect to the column portion are still covered as closure under affine

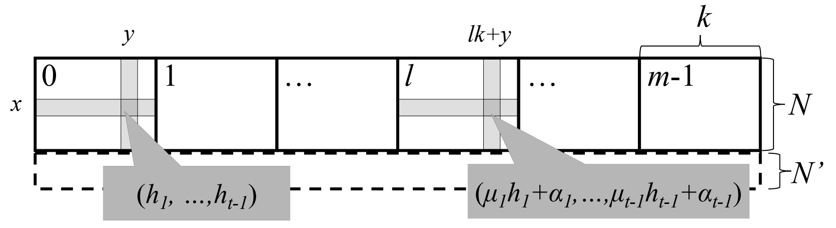


Figure 2.2: Transformed columns in an SCPHF by AF-COMP

transformation does not hold when different affine transformations are applied. AF-COMP must check the t-sets between subarrays to compute the remaining list. The CE algorithm is again employed to build the final N' rows and complete the SCPHF. The AF-COMP algorithm is given in Algorithm 3.

2.2.3 Composition Bounds

When COMP is used, $\binom{km}{t} - \binom{k}{t}m^t$ t-sets remain after constructing the copies. There are $\binom{km}{t}$ total t-sets in the composed array and $\binom{k}{t}$ ways to select t distinct columns from the original SCPHF with m choices of subarray for each of the t columns. Recall that the upper bound for the number of rows, N_k, for the ingredient with k columns is

$$\frac{\log\binom{k}{t}}{\log\frac{1}{\beta}} = \frac{\log\binom{k}{t}}{\log\frac{(v^{t-1})^t}{(v^{t-1})^t-(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1)+nc_t(v)}} < N_k$$

Let N_{COMP} be the number of rows constructed during the run of CE in the COMP algorithm, with the total number of rows bounded by $N_k + N_{COMP}$. Then, an upper bound on the number of rows added during COMP is the smallest N_{COMP} for which

$$\frac{\log(\binom{km}{t} - \binom{k}{t}m^t)}{\log\frac{1}{\beta}} = \frac{\log(\binom{km}{t} - \binom{k}{t}m^t)}{\log\frac{(v^{t-1})^t}{(v^{t-1})^t-(v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1)+nc_t(v)}} < N_{COMP}$$

holds. When the construction is done entirely by CE, the upper bound is the smallest

```
Algorithm 3: Affine Composition (AF-COMP)
input  : A an $N \times k$ strength $t$ array on $v^{t-1}$ symbols, $m$ a number of copies
output: A′ an $(N + N') \times km$ strength $t$ array on $v^{t-1}$ symbols, $N + N'$
begin
    Create an empty $N \times mk$ array, $\mathbf{A}'$
    for row $0 \leq x < N$ do
        for subarray index 0 do
            for column index $0 \leq y < k$ do
                Set $\mathbf{A}'[x][\ell k + y] = A[x][y]$
        for subarray index $1 \leq \ell < m$ do
            Choose a transformation
              $\{\{\mu_1, \alpha_1\}, \cdots, \{\mu_t, \alpha_t\}\}, 1 \leq \mu \leq v, 0 \leq \alpha \leq v$ at random
            for column index $0 \leq y < k$ do
                Set $\mathbf{A}'[x][\ell k + y]$ to the image of $\mathbf{A}[x][y]$ under the
                  transformation
    for $\binom{km}{t} - \binom{k}{t}m$ t-sets $T$ containing columns spanning more than one
      subarray do
        if $T$ is not covered in $\mathbf{A}'$ then
            Add $T$ to remaining list
    Run CE on $\mathbf{A}'$ to build the final $N'$ rows
    Return $\mathbf{A}', N + N'$
```

N_{km} for

$$\frac{\log \binom{km}{t}}{\log \frac{1}{\beta}} = \frac{\log \binom{km}{t}}{\log \frac{(v^{t-1})^t}{(v^{t-1})^t - (v^{t-1})(v^{t-1}-1)\cdots(v^{t-1}-t+1) + nc_t(v)}} < N_{km}.$$

$N_k \leq N_{km}$ and $N_k < N_k + N_{COMP}$ when $m > 1$, as expected. An SCPHF with a multiple of k columns cannot require fewer rows than the SCPHF with k. The duplication in COMP always results in needing at least one additional row to cover the t-sets that include at least one column with the same column index.

The relationship between $N_k + N_{COMP}$ and N_{km} is not as clear. Experimentally, evidence suggests that $N_k + N_{COMP} > N_{km}$, and this is likely true when k and m are small. Asymptotically when v and t are fixed, $N_{km} = O(log((km)^t))$ and

$N_k + N_{COMP} = O(log(k^t)) + log((km)^t - k^t m^t)$. When m is small and k approaches infinity, both are $O(log(k^t))$.

It is unclear if there is an efficient way to compute *a priori* how many t-sets remain after constructing copies in AF-COMP and thus how to compute $N_{AF-COMP}$. Closure under affine transformation for the covering and noncovering classes ensure that no more than $\binom{km}{t} - \binom{k}{t}m$ t-sets remain, but this bound is larger than the one obtained for COMP. Computing the uncovered t-sets after AF-COMP is problematic without knowing the tuples in the columns and the transformations applied.

2.3 Results

2.3.1 Evaluation of CE

CE constructed the arrays in Tables 2.1 and 2.2, and these covering arrays improve upon the previously known results for the parameters t, k, and v. Random Extension is a method to extend an SCPHF by one column. Columns are randomly generated until one is found that can be appended to an $\mathsf{SCPHF}(N; k, v, t)$ to produce an $\mathsf{SCPHF}(N; k+1, v, t)$ or an iteration limit is reached. CE creates good ingredients to be further extended by Random Extension (RE), and results from the combined method CE-RE are published in [19]. For many ranges of values for $t = 3$, CE-RE improved upon the best previously known results.

One of the time optimizations, using a standard first row, may cause noncovering tuples to be placed that are avoided by the CE search method when k is small, resulting in more rows. The benefit of constructing the first row by CE search is reduced as k increases, and the standard first row provides significant time savings, as it takes minimal time to construct, yet covers a large percent of the $\binom{k}{t}$ t-sets of columns. For strength three, the first row options of standard versus search tend

Table 2.1: Strength three CAs

CA(605; 3, 179, 5)	CA(725; 3, 375, 5)	CA(965; 3, 1518, 5)
CA(1351; 3, 174, 7)	CA(2023; 3, 1066, 7)	CA(1016; 3, 56, 8)
CA(2024; 3, 228, 8)	CA(3032; 3,1550,8)	CA(2889; 3, 294, 9)
CA(4329; 3, 1525, 9)	CA(5291; 3, 442, 11)	CA(8749; 3, 400, 13)
CA(16336; 3,957,16)	CA(9809; 3, 65, 17)	CA(19601; 3, 765, 17)
	CA(13699; 3, 74, 19)	

Table 2.2: Strength four CAs

CA(1516; 4, 63, 4)	CA(1768; 4, 84, 4)	CA(3725; 4, 97, 5)
CA(4345; 4, 125, 5)	CA(9583; 4, 56, 7)	CA(11977; 4, 72, 7)
CA(14371; 4, 98, 7)	CA(16765; 4, 108, 7)	CA(26217; 4, 40, 9)

to result in the same number of rows when three rows are needed in the SCPHF. Experimental support for the algorithmic option of a standard first row is provided by one example given here. Constructing an $\mathsf{SCPHF}(120, 5^3, 4)$, which becomes a $\mathsf{CA}(4345; 4, 120, 5)$, both options result in the same number of rows, $N = 7$. They also have the same number of remaining t-sets after the first row, 1,644,570, which is 20% of the total 8,214,570 t-sets. The effect on running time is substantial. Construction of the SCPHF takes 1299.9 minutes when the first row is constructed by search and 263.1 minutes with the standard first row.

2.3.2 Comparison of COMP to CE

The advantages of the composition algorithms are decreased memory and time. CE requires too much memory to build an $\mathsf{SCPHF}(5000, v, 3)$ due to adding all of the t-sets to the remaining list initially, whereas COMP reduces the storage of the remaining list to just those t-sets that contain two or more columns with the same column portion of the index from the ingredient. COMP builds an $\mathsf{SCPHF}(9; 1000, 5^2, 3)$ by making ten copies of an $\mathsf{SCPHF}(5; 100, 5^2, 3)$ in 38 seconds, 2.7% of the time used by the CE algorithm, while resulting in just one extra row. At the time this work was conducted

Table 2.3: Covering arrays constructed by COMP

Ingredient	m	SCPHF	Covering array
$\mathsf{SCPHF}(6; 375, 5^2, 3)$	5	$\mathsf{SCPHF}(8; 1000, 5^2, 3)$	CA(965; 3, 1000, 5)
$\mathsf{SCPHF}(6; 375, 5^2, 3)$	5	$\mathsf{SCPHF}(9; 1875, 5^2, 3)$	CA(1085; 3, 1875, 5)
$\mathsf{SCPHF}(8; 1000, 5^2, 3)$	5	$\mathsf{SCPHF}(11; 5000, 5^2, 3)$	CA(1325; 3, 5000, 5)
$\mathsf{SCPHF}(6; 714, 7^2, 3)$	7	$\mathsf{SCPHF}(9; 4998, 7^2, 3)$	CA(3031; 3, 4998, 7)

and prior to the development of CE-RE, COMP found covering arrays that compare favorably to the best known arrays, including those in Table 2.3.

2.3.3 Comparison of COMP to AF-COMP

Experimental data in Table 2.4 supports the intuition that AF-COMP outperforms COMP. In all cases, fewer t-sets remaining after AF-COMP correspond to fewer rows in the final array and shorter running time for CE to build the final N' rows. In general, the improvement increases as problem parameters increase. The one case where the trend does not hold is $m = 2, v = 5$. Randomness in the algorithm likely selected a set of affine transformations that performed better than the cases for six and ten copies. Table 2.5 shows that the benefit of AF-COMP over both COMP and directly constructing by CE becomes more significant as t increases. The improvement of AF-COMP over COMP is substantial even when the number of copies m is quite small.

2.3.4 Comparison of Composition Strategies

There are many choices of k and m for composition to achieve an SCPHF with km columns. Figure 2.3 compares a direct construction by CE for a CA(3,1000,5) against composition constructions by both COMP and AF-COMP. The notation $m \times (k)$ indicates m copies were made of a k column ingredient. Time, presented in seconds, includes the time to construct the ingredient with N rows by CE, perform

Table 2.4: Comparison of AF-COMP to COMP by total rows produced (N) and number of remaining t-sets (R) after composition phase for $\mathsf{SCPHF}(50m, v, 3)$

m	v	COMP(N)	AF-COMP(N)	COMP(R)	AF-COMP(R)	% of COMP(R)
2	3	10	8	4900	433	8.84%
6	3	13	11	221500	18096	8.17%
10	3	14	12	1108500	91359	8.24%
2	4	8	7	4900	323	6.59%
6	4	10	8	221500	9886	4.46%
10	4	11	9	1108500	47017	4.24%
2	5	7	6	4900	206	4.20%
6	5	8	7	221500	11599	5.24%
10	5	9	7	1108500	56004	5.05%

Table 2.5: Comparison of AF-COMP to COMP by number of remaining t-sets (R) after composition phase for $\mathsf{SCPHF}(64m, 4^3, 4)$ and comparison to total $\binom{km}{t}$ t-sets

m	COMP(R)	AF-COMP(R)	% of COMP(R)	Total	% of Total
2	501984	1398	0.278%	10668000	0.013%
4	12136384	33246	0.274%	174792640	0.019%
6	68434080	191325	0.280%	891881376	0.021%

the composition, and run CE again to produce the final N' rows. In all cases, AF-COMP results in fewer rows and less time than COMP. Additionally, AF-COMP is much faster than CE and, in two of the three cases, produces a covering array with the same number of rows as CE.

The performance of COMP on the response variable time shows an interesting trend. As the size of the ingredient decreases and the number of copies increases, the time to construct increases. When many copies are made, it is likely that the repeated structure results in more uncovered t-sets. These must be covered by the second execution of CE, which is more time intensive. The time savings by constructing the 50 column ingredient versus the 200 column ingredient does not make up for the time executing CE on the 1000 column SCPHF. This trend appears to be reflected in AF-COMP, though substantially reduced. This suggests that to obtain the time

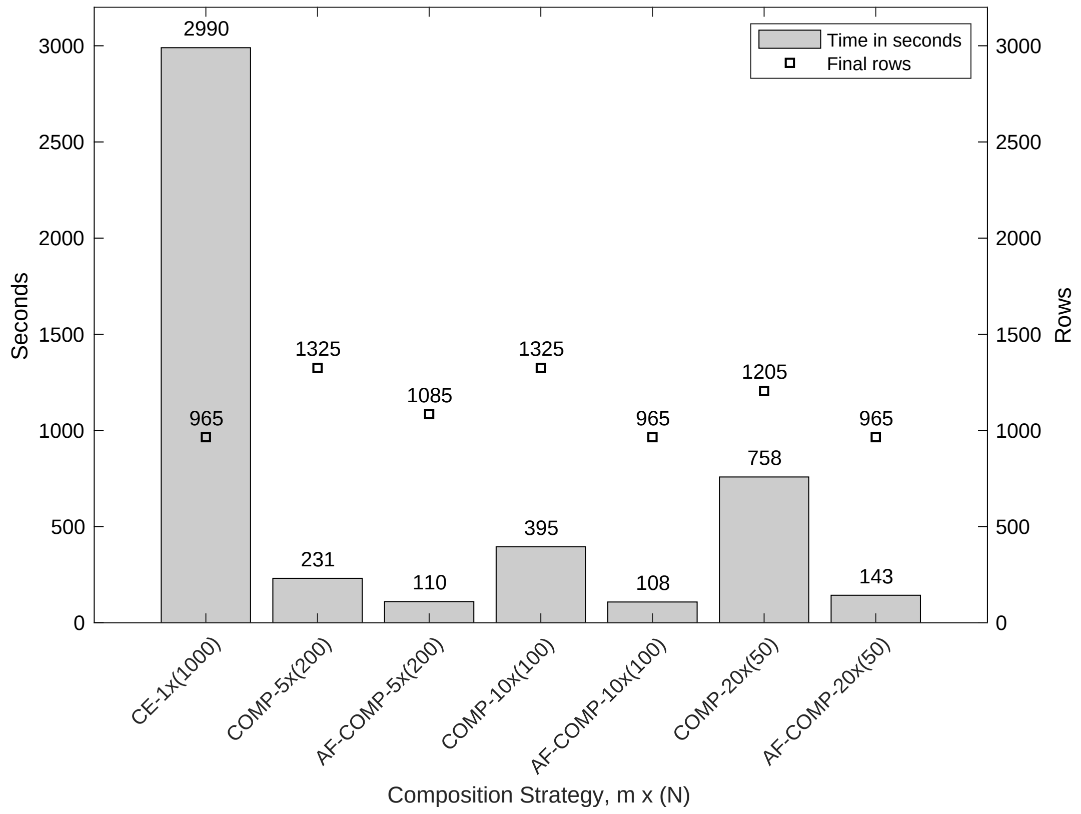


Figure 2.3: Comparison of composition strategies by size of ingredient and number of copies for construction of $\mathsf{CA}(3, 1000, 5)$ via SCPHFs

benefit of composition, a reasonable strategy is to keep the number of copies small. For rows, both COMP and AF-COMP produce the fewest rows when the smallest ingredient and largest number of copies are used. As is often the case, there exists a tradeoff between time efficiency and accuracy. Then a strategy that prioritizes time efficiency and accuracy may be to choose k and m to be in the middle.

Instead of performing one large composition, repeated compositions can be performed to build up to the desired km. Figure 2.4 compares different strategies in terms of the final number of rows and total running time to again create a $\mathsf{CA}(3{,}1000{,}5)$. For repeated compositions, the notation $\ell \times (m \times (k))$ indicates m copies were made of a k column ingredient followed by ℓ copies of the resulting km column SCPHF. For a third

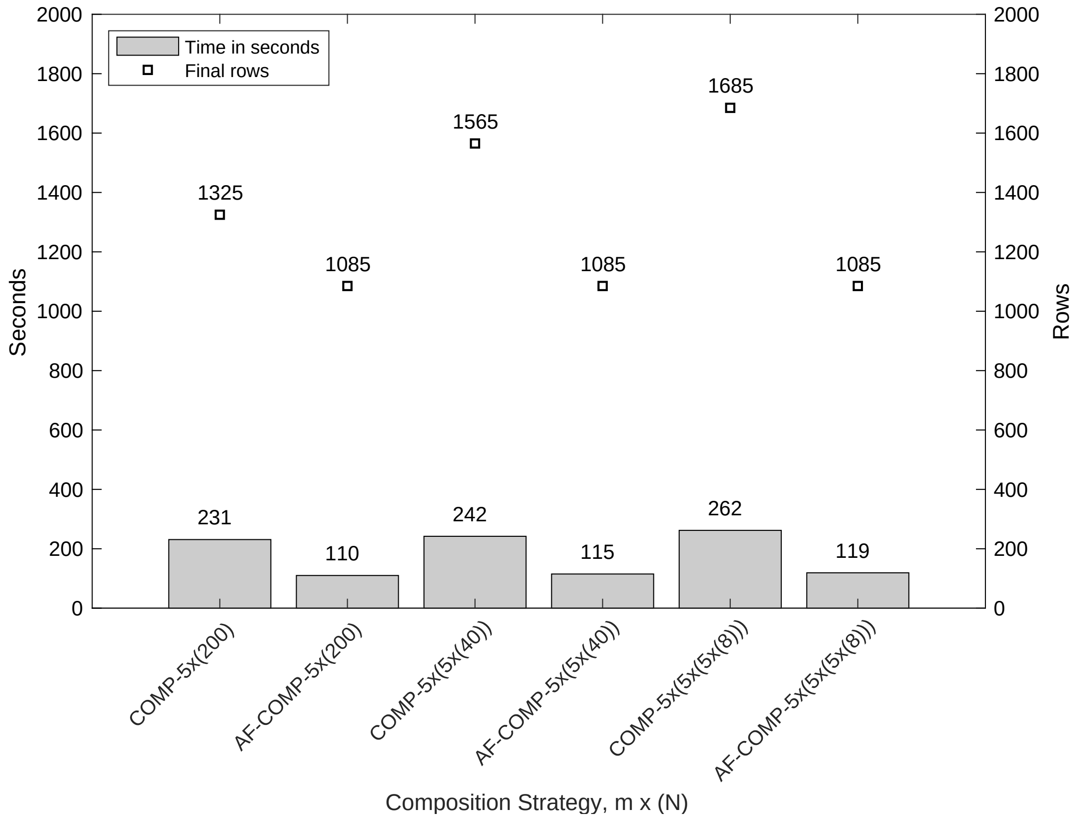


Figure 2.4: Comparison of composition strategies by repeated compositions for construction of $\mathsf{CA}(3, 1000, 5)$ via SCPHFs

composition, the number of copies is placed to the left and the pattern continued. For COMP, when smaller ingredients are used with repeated compositions, the result is more time and more rows, suggesting that recursively performing COMP is not a useful strategy. Recursive AF-COMP suffers from increased time, but again only marginally, and it does not produce additional rows. It seems likely that the culprit is the repeated structure in COMP that is compounded by recursive compositions. AF-COMP reduces the repeated structure through the affine transformations.

Chapter 3

FAULT LOCATION

Covering arrays are employed to create test suites for interaction testing as they indicate the presence of a faulty interaction in some test in the suite. Locating arrays determine the interaction or set of interactions causing the fault when the number of faults is at most d. The extra property of fault location comes at the cost of more rows and increased complexity in constructing the array. Necessary background and definitions for this chapter are provided in § 1.2 of the introduction. The contribution of this work is to develop an algorithm to construct mixed-level locating arrays for general d and t and to provide an algorithmic framework to accommodate speed, measured primarily in seconds but also in iterations of the algorithm, versus accuracy, measured in number of rows, i.e., being closer to $\mathsf{LAN}(d, t, k, v)$. This chapter is organized as follows. The Partitioned Search with Column Resampling (PSCR) construction for locating arrays is presented in § 3.1. Parameter tuning results and guidance, comparison of PSCR to other constructions, and constructing larger locating arrays from ingredients using PSCR are explored in § 3.2.

3.1 Locating Array Construction

3.1.1 Overview

The Partitioned Search with Column Resampling (PSCR) algorithm combines Moser-Tardos-style column resampling with partitioning the search space to only those sets of interactions that could share the same rows [38]. PSCR constructs arrays that are $(\bar{d}, t)$-locating, ensuring that sets of at most d t-way interactions appear in unique sets of rows.

PSCR is inspired by Moser-Tardos resampling [42], but differs in important ways from the implementation of column resampling for locating arrays in [52]. Moser-Tardos column resampling for locating array construction begins with a randomly selected candidate array with the number of rows meeting the Lovász Local Lemma bound. The interactions of the array are checked in an arbitrary but fixed ordering until a "bad event" that violates the conditions is found, i.e., two sets of t-way interactions, $\mathcal{T}_1, \mathcal{T}_2$ with $|\mathcal{T}_1| \leq d, |\mathcal{T}_2| \leq d$ and $\rho(\mathcal{T}_1) = \rho(\mathcal{T}_2)$. Call these bad events *collisions*. When the first collision is identified, all columns in $\mathcal{T}_1$ and $\mathcal{T}_2$ are resampled, and checking begins again in the same fixed order. The fixed ordering allows for an expectation to be computed on the number of resamplings needed to find a solution in which there are no collisions, and Moser and Tardos prove that the expected number of resamplings is polynomial in k when d, t, and the numbers of levels are fixed.

PSCR exploits the fact that not all sets of interactions need to be compared. The search space is partitioned to systematically "guess" the row that must be in the set $\rho(\mathcal{T}_1) = \rho(\mathcal{T}_2)$ and only sets that include that row are compared. As resamplings occur, interactions may appear in different rows and so may be compared in a different order. That is, PSCR is driven by an ordering on rows rather than interactions. Additionally, PSCR starts with some initial number of rows and adds a new row when the progress made in reducing the number of collisions drops below a threshold. Lastly, the number of columns involved in a single collision can be as high as $2dt$. When d and t are not small relative to k, a single resampling can rewrite a large portion of the array. When the candidate array is close to satisfying the property, this may introduce more collisions and reverse progress. PSCR takes a more conservative approach, and the number of columns to resample, either one or t, is tunable.

It is unclear if the change in ordering is sufficient to invalidate the assumptions made and the potential impact on the convergence in the Moser-Tardos method. In practice, however, PSCR can construct locating arrays with fewer rows than the Moser-Tardos column resampling method when the algorithmic parameters are correctly tuned (see § 3.2.3).

3.1.2 Verification

The first step to constructing a locating array is verifying when a candidate array is $(\bar{d}, t)$-locating. Given a candidate array, $\mathbf{A}$, any verifier must check that all t-way interactions are covered in some row (condition 1) and the rows corresponding to a set of at most d t-way interactions are not the same as those for another set of at most d t-way interactions (condition 2).

The verifier creates a two-dimensional incidence matrix $\mathbf{I}$ where the $|\mathcal{I}_t|$ rows represent interactions and the N columns represent rows in $\mathbf{A}$. That is, row i of $\mathbf{I}$ represents interaction i in $\mathbf{A}$, and $\mathbf{I}[i][n] = 1$ if and only if $n \in \rho(i), 1 \leq n \leq N$. If the sum of any row is 0, then $\mathbf{A}$ does not meet condition 1. For condition 2, the naive approach is to conduct pairwise comparisons: for each way to choose at most d rows of $\mathbf{I}$, take the union of the rows and compare this to all other ways to choose at most d. If any two unions are equal, $\mathbf{A}$ is not a locating array.

The number of t-way interactions, $|\mathcal{I}_t|$, is $\sum_{i \in T, |T|=t} \prod v_i$, or $\binom{k}{t} v^t$ for uniform v. Let S be the number of sets of at most d t-way interactions,

$$S = \sum_{i=1}^{d} \binom{|\mathcal{I}_t|}{i}.$$

The number of comparisons that must be made is $\binom{S}{2}$ which can become infeasible quickly.

Another approach is to build the sets intelligently so that if there are two sets of interactions that share the same rows, they can be found with fewer comparisons. Assume there is some $\mathcal{T} \neq \mathcal{T}'$ for which $\rho(\mathcal{T}) = \rho(\mathcal{T}')$. Then there is some row $r \in \rho(\mathcal{T})$ such that r has the smallest index, and the search space can be partitioned into N parts so that not all sets have to be compared, just those for which r is the minimum element of $\rho(\mathcal{T})$.

To accomplish the partition, interpret the rows of indicator matrix $\mathbf{I}$ as binary numbers for sorting the interactions of $\mathbf{A}$ in increasing value of the representation of $\rho(i)$. Build N trees such that for every interaction set $\mathcal{T}$ in tree r, r is the smallest element of $\rho(\mathcal{T})$. That is, tree r does not include any interaction that appears in a row of $\mathbf{A}$ with index smaller than r. Each node u of the tree stores the following information: a set of interactions $\mathcal{T}_u$, a set of rows $\mathcal{R}(u)$ representing $\rho(\mathcal{T}_u)$ from $\mathbf{A}$, and $depth(u)$ at this node. The root node of tree r, $root(r)$, has the zero vector for $\mathcal{R}(r)$, $\mathcal{T}_r = \emptyset$, and $depth(r) = 0$. A node w is added as a child of $root(r)$ for every interaction i that has r as the smallest element of $\rho(i)$. For example, $root(2)$ has a child for every interaction i where $\mathbf{I}[i][2] = 1$ and $\mathbf{I}[i][0] = \mathbf{I}[i][1] = 0$. The interaction i is stored as $\mathcal{T}_w$, $\rho(i)$ is stored as $\mathcal{R}(w)$, and $depth(w) = 1$. For node x representing the addition of interaction j to form the set $\mathcal{T}_x$ in tree r with $depth(x) < d$, add a node y as a child of x for each interaction ℓ where index $\ell < j$ in the sorted indicator matrix $\mathbf{I}$. Node y stores $\mathcal{T}_y = \mathcal{T}_x \bigcup \{\ell\}$, $\mathcal{R}(y) = \mathcal{R}(x) \bigcup \rho(\ell)$, and $depth(y) = depth(x) + 1$.

As tree r is built, for every created node u, add $\mathcal{R}(u)$ to a sorted list, $\mathbf{L}_r$. If $\mathcal{R}(u)$ is already in $\mathbf{L}_r$, this is a collision, and $\mathcal{R}(u) = \mathcal{R}(x)$ for some two nodes, representing two distinct sets of interactions of size $\leq d$, and so $\mathbf{A}$ is not a locating array. Once all collisions are found, the participating interaction sets, $\mathcal{T}_u$ and $\mathcal{T}_x$, can be stored in a table of collisions, and the memory for tree r can be reused to create tree $r + 1$ and memory for list $\mathbf{L}_r$ reused for $\mathbf{L}_{r+1}$. Finding collisions throughout the tree and not just at the leaf nodes ensures that the locating arrays found are $(\bar{d}, t)$-locating.

This method checks each of the sets of interactions once. Each level 1 (child of a root) node is unique; the level 1 nodes in tree r are all interactions that have r as the smallest row in which they appear. These are the $\binom{|\mathcal{I}_t|}{1}$ sets. Each successive child of a node is unique; take a level 1 node and add each of the interactions that have smaller index in $\mathbf{I}$ successively to create each of the level 2 nodes. These are the $\binom{|\mathcal{I}_t|}{2}$ sets. This holds true down to the level d nodes.

The naive approach makes pairwise comparisons of all sets of size at most d and is therefore $\mathcal{O}(S^2)$. Our approach creates a node for each of the sets once, $\mathcal{O}(S)$. However, in each step, the sorted list $\mathbf{L}_r$ is maintained via insertion sort. Operating on lists of size $\frac{S}{N}$ on average, the sorted list adds $\mathcal{O}((\frac{S}{N})^2)$ steps for each of the N trees, or $\mathcal{O}(\frac{S^2}{N})$ total. Then, our approach is $\mathcal{O}(S^2)$ as well, but a practical benefit in running time is from partitioning the space into the N groups. Replacing the insertion sorted list with a hash table may reduce the running time to $\mathcal{O}(S)$.

Figure 3.1 demonstrates how tree 2 is built given the sample indicator matrix, $\mathbf{I}$, for a candidate locating array with $d = 2$, h interactions, and four rows. Interactions b and c have row 2 as their smallest row, so they are added as level 1 nodes. They are the only such interactions as while interaction h appears in row 2, it also appears in row 1 and so is processed in tree 1. Interaction a is added to b and interactions b and a are individually added to interaction c to form level 2 nodes. When inserting the $\mathcal{R}$ values into the sorted list $\mathbf{L}$ for each node, collisions are found that involve the sets $\{\{c\}\}, \{c, b\}\}$ and $\{\{b, a\}\}, \{c, a\}\}$. These are added to the collision table $\mathbf{C}$ for processing in the repair step. Up to four trees, one per row of the candidate array $\mathbf{A}$, are created in this phase of the algorithm. Options that indicate whether to build all trees before attempting to repair are discussed in the next section.

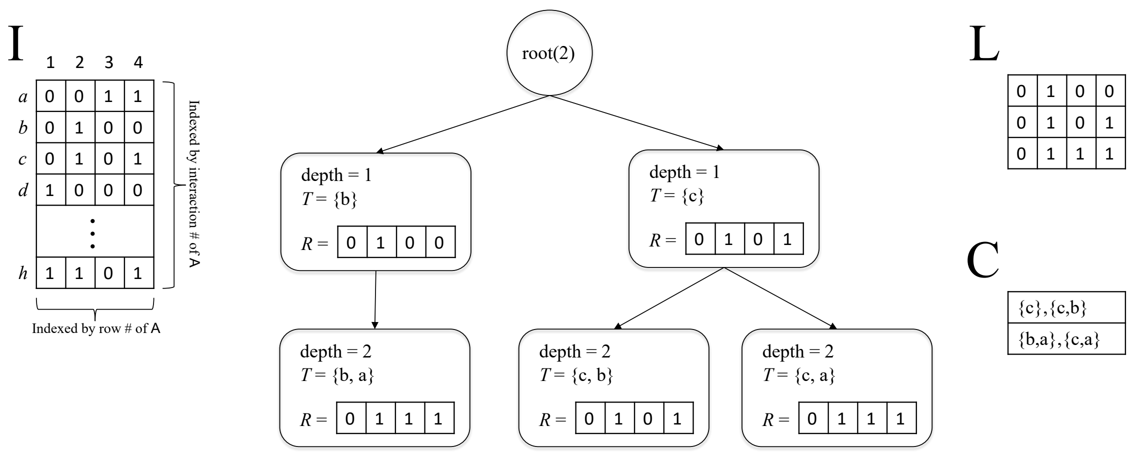

I
1 2 3 4
a 0 0 1 1
b 0 1 0 0
c 0 1 0 1
d 1 0 0 0
h 1 1 0 1
Indexed by interaction # of A
Indexed by row # of A
root(2)
depth = 1 T = {b} R = 0 1 0 0
depth = 1 T = {c} R = 0 1 0 1
depth = 2 T = {b, a} R = 0 1 1 1
depth = 2 T = {c, b} R = 0 1 0 1
depth = 2 T = {c, a} R = 0 1 1 1
L
0 1 0 0
0 1 0 1
0 1 1 1
C
{c},{c,b}
{b,a},{c,a}

Figure 3.1: Example of tree 2 in verification step of PSCR

3.1.3 Repair

If verification fails, an attempt is made to repair the array with column resampling and try verification again. Algorithmic parameters — collision handling option (CHO), α, *adaptive-*α, *avoid*, *maxIterations*, *initialN* — can be set to prioritize speed or accuracy. Given a candidate array with *initialN* rows, PSCR executes the following two routines in a loop until it produces a locating array or it meets the threshold for maximum iterations allowed (*maxIterations*).

1. *Verify coverage (condition 1).* If coverage fails, resample a randomly chosen column for the interaction that fails to appear, and continue this loop of verification and resampling until the array satisfies condition 1. In an early version of the algorithm, all columns for the non-appearing interaction were resampled, but when k is small relative to t, this appears to cause too much disruption in the candidate array and can affect accuracy. For large numbers of factors, resampling all t columns is a viable alternative to increase speed and is less likely to affect accuracy. Resampling is conducted with a weighted probability

on symbols such that the more often a symbol appears in the rows of a column, the less likely it is to be selected in future rows with a cap of $\lceil \frac{N}{v} \rceil$. If the resampling loop has completed v^t times without passing, a row is added to the candidate array. This routine may add more rows than are necessary to achieve coverage. The parameter *avoid* set to true prevents adding a row in this subroutine if the candidate array has satisfied the coverage condition previously. When condition 1 is satisfied, proceed to verify condition 2.

2. *Verify location (condition 2).* This routine includes the verifier from the previous section. If condition 2 is satisfied, the candidate is a locating array. If not, collision handling attempts to improve the candidate array. Additionally, if a threshold is reached for non-progress, a randomly chosen row is added.

Five collision handling options reflect differences in the size of the array to check before exiting with a collision and how to resample once collisions are found. The first three options select one interaction implicated in a collision and resample all of the columns in the interaction. The fourth option instead resamples a single column. The fifth option drives execution to one of the other options based on conditions of the candidate.

1. *Greedy* ($CHO = 0$). When the first collision is found, exit the verifier. Randomly choose a colliding interaction set $\mathcal{T}$ from the set of two colliding interaction sets, randomly choose an interaction $T \in \mathcal{T}$, and resample all columns of T.

2. *Random* ($CHO = 1$). Find all collisions, then randomly choose a colliding interaction set $\mathcal{T}$ from the set of all colliding interaction sets, randomly choose an interaction $T \in \mathcal{T}$, and resample all columns of T.

3. *Interaction* ($CHO = 2$). Find all collisions, then compute the interaction T involved in the most collisions, and resample all columns of T.

4. *Column* ($CHO = 3$). Find all collisions, then resample the single column involved in the most collisions.

5. *Adaptive* ($CHO = 4$). The adaptive option exits the verification phase if the percentage of colliding interactions is higher than a threshold for the number of sets checked so far. Let collision rate $= \frac{\text{colliding interaction sets}}{\text{interaction sets checked}}$ and completion $= \frac{\text{interaction sets checked}}{S}$ where S is the number of sets of at most d t-way interactions as defined in § 3.1.2. This implementation exits early if the inequality

$$\text{(collision rate)} \geq -0.5\text{(completion)} + 0.5$$

holds. Additionally, the collision handling option executed is based on the percentage of interaction sets that are colliding, percentage colliding $= \frac{\text{colliding interaction sets}}{S}$. This implementation uses the following boundaries:

$$1 \geq \text{(percentage colliding)} > 0.25 \rightarrow CHO = 1$$
$$0.25 \geq \text{(percentage colliding)} > 0.05 \rightarrow CHO = 2$$
$$0.05 \geq \text{(percentage colliding)} > 0.00 \rightarrow CHO = 3$$

A *move* is changing $\mathbf{A}$ via column resampling. After each move, the algorithm checks to see if it is making *progress*, defined as reducing the number of collisions more than a fixed distance from the mean given by the tunable parameters α and *adaptive*-α. The number of collisions found in an iteration is counted and a z-score for the iteration is computed and compared to the mean of the *history*, the last 10 recorded collision counts. The collision counts for the first 10 iterations are always recorded as they seed the history. The algorithm avoids allowing the number of collisions to climb upward in a series of small steps by fixing the history mean to the lowest mean seen so far; only collision counts with non-positive z-scores are recorded. Riskier moves, those that produce less favorable z-scores, are allowed towards the beginning of execution to attempt to avoid getting stuck at a local optimum by using a measure for $time = \frac{iteration}{maxIterations}$ in decisions.

It may be difficult or even impossible to create a locating array with the current number of rows. The threshold for progress is defined by $\alpha + 0.25 * time$. When *adaptive*-α is false, the threshold is simply α; when true, the threshold increases as time passes. The intention behind *adaptive*-α is to allow more rows to be added early in execution, giving the algorithm the most time possible to use these added rows to find a solution, but to prevent unnecessary rows from being added late in execution.

A z-score falls into one of four zones:

1. *Rollback.* A move with a z-score of $z \geq 1 - time$ produces many more collisions than the mean. This move is disallowed and triggers a rollback to the previous state of **A**. The gradually decreasing upper limit on acceptable z-scores allows for riskier moves early in execution but requires moves to make better progress as time runs out.

2. *Allow, add a row.* A move may be allowable, but produces a positive z-score and, therefore, a number of collisions that is greater than the mean. This zone is defined as $0 < z < 1 - time$. The number of collisions produced is not added to the history so that the history mean never increases, and a row is added.

3. *Allow, add a row, and record.* A move with a non-positive z-score has no more collisions than average, but it may not be making as much progress as desired. The threshold is compared to the percentage of the normal distribution to the left of the z-score; if $percent > \alpha$, the z-score does not reside in the left tail

defined by α, so it is not progressing. A row is added and the number of collisions is added to the history.

4. *Allow and record.* If $percent \leq \alpha$, the algorithm is progressing, so no new row is added. The number of collisions is added to the history.

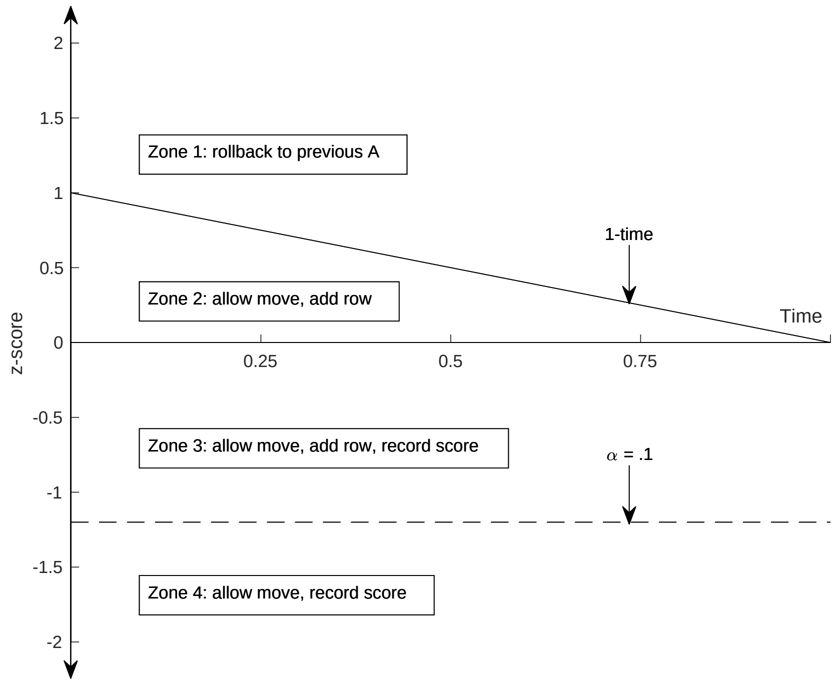
The zones for $\alpha = 0.1$ when α is constant are depicted in Figure 3.2. When $\alpha = 0.1$, a z-score of $z = -1.2$ is in zone 3, but $z = -1.3$ is in zone 4. Figure 3.3 illustrates the zones for *adaptive*-α as the threshold increases proportionally to *time*. Setting *adaptive*-α to true has the result that a move towards the beginning of execution that results in an added row might not result in an added row towards the end of execution. For example, near the end of execution, $z = -0.3$ is in zone 3 and $z = -0.4$ is in zone 4, while both are in zone 3 at the beginning.

3.1.4 Partitioned Search with Column Replacement (PSCR) Algorithm

A high level view of PSCR is presented in Algorithm 4.

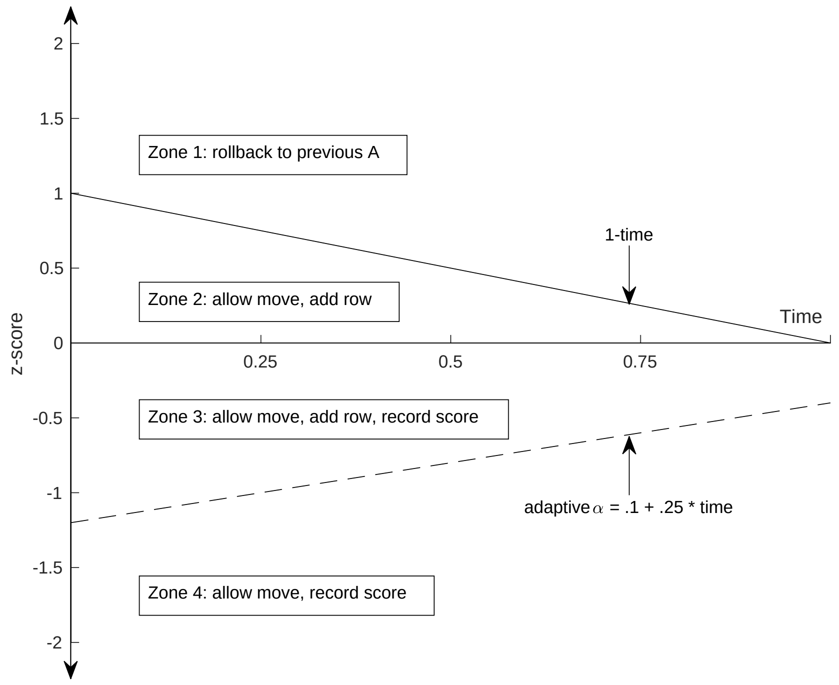
3.2 Results

In this section, results from PSCR are produced and analyzed along three dimensions. The performance of the tunable parameters is analyzed to develop an algorithmic framework to accommodate accuracy versus speed and provide guidance on parameter settings. Results from PSCR are compared against constructions from higher strength covering arrays as well as against reported results for implementations of column resampling and local optimization. Additionally, using PSCR to build locating arrays when provided with various ingredient arrays is explored.



2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
z-score
Zone 1: rollback to previous A
1-time
Zone 2: allow move, add row
Time
0.25
0.5
0.75
Zone 3: allow move, add row, record score
α = .1
Zone 4: allow move, record score

Figure 3.2: Zones for constant $\alpha = 0.1$



2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
z-score
Zone 1: rollback to previous A
1-time
Zone 2: allow move, add row
Time
0.25
0.5
0.75
Zone 3: allow move, add row, record score
adaptive α = .1 + .25 * time
Zone 4: allow move, record score

Figure 3.3: Zones for *adaptive*-α , initial $\alpha = 0.1$

Algorithm 4: Partitioned Search with Column Replacement (PSCR)

```
input  : $\mathbf{A}, d, t, k, v, N, CHO, \alpha, adaptive\text{-}\alpha, avoid, maxIterations$
output: $\mathbf{A}, N, seconds, iteration$
begin
    $iteration = 1$
    $resamplings = 0$
    repeat
        Make a copy of $\mathbf{A}$ as $\mathbf{A}'$
        Create $\mathbf{I}$, $|\mathcal{I}_t| \times N$ boolean matrix
        while some interaction does not appear in $\mathbf{A}$ do
            if (avoid is false OR $\mathbf{A}$ never satisfied coverage) AND resamplings
              of loop $\geq v^t$ since row added then
                add a randomly selected row to $\mathbf{A}$, increment $N$
            for each interaction $i \in \mathcal{I}_t$ do
                set $\mathbf{I}[i][j] = 1 \iff$ interaction $i$ appears in row $j$ of $\mathbf{A}$
                if the sum of row $i$ in $\mathbf{I}$ is 0 then
                    randomly select and resample one of the columns of
                      interaction $i$ in $\mathbf{A}$
        Sort $\mathbf{I}$
        for each column $n$ of $\mathbf{I}, 1 \leq n \leq N$ do
            Create $root(n)$
            for every row index $i \leq N$, $n$ the smallest $r$ for which $\mathbf{I}[i][r] = 1$ do
                Add child node $u$ to $root(n)$ with $\mathcal{T}_u = \{i\}$, $\mathcal{R}(u) = \mathbf{I}[i]$, and
                  $depth(u) = 1$
                Insert $\mathcal{R}(u)$ in list $\mathbf{L_n}$
                for level $1 \leq currDepth < d$ do
                    for every node $w$ in level $currDepth$ do
                        for every row index $j \in \mathbf{I}$ such that $j <$ index for $w$ do
                            Add child node $x$, $\mathcal{T}_x = \mathcal{T}_w \bigcup \{j\}$,
                              $\mathcal{R}(x) = \mathcal{R}(w) \bigcup \mathbf{I}[j]$, $depth(x) = depth(w) + 1$
                            if inserting $\mathcal{R}(x)$ in $\mathbf{L_n}$ causes collision then
                                add the collision to a set of collisions, $\mathbf{C_{iteration}}$
                                if CHO=0 then
                                    exit
        if $\mathbf{C_{iteration}} = \emptyset$ then
            output $\mathbf{A}, N, seconds, iteration$
        else
            Compute z-score of $\mathbf{C_{iteration}}$ compared to history
            Set $\mathbf{A} = \mathbf{A}'$ if disallowed
            Add a row if not progressing enough
            Update history with $|\mathbf{C_{iteration}}|$
        Resample columns of $\mathbf{A}$ based on CHO, percent colliding
        Increment iteration
    until $iteration = maxIterations$
    Reached $maxIterations$ without resolving all collisions
```

d	t	k	v	maxIterations
1	1	1000	7	600
1	2	25	6	250
1	3	8	4	300
2	1	20	5	500
2	2	5	3	350

Table 3.1: Blocks for $5 \times 5 \times 3 \times 2 \times 2$ full factorial

3.2.1 Parameter Tuning

The following algorithmic parameters are tuned to prioritize speed or accuracy: *collision handling option* (CHO) with five levels, α with range $(0.0003, 0.5)$, and two binary parameters, *adaptive-*α and *avoid.* To determine the effect on the number of rows in the constructed locating array and the time required to construct it, a full factorial designed experiment is conducted with four replicates with response variables rows (N), iterations, and seconds. The response variables are converted to z-scores to make the analysis straightforward over blocks that may have wildly different ranges for these variables. Three values for $\alpha \in \{0.001, 0.2, 0.4\}$ are selected and blocks are formed from five assignments of problem parameters as blocks described in Table 3.1.

The tunable parameter *maxIterations* is highly problem dependent, so it is not included as a factor in the designed experiment. If *maxIterations* is set too low, it is possible that PSCR may not find a locating array before timing out; if set too high, the time-dependent algorithmic features deciding when to rollback to a previous state and *adaptive-*α may never have the intended impact. A reasonable value for the iterations needed to complete execution for each block is estimated by requiring that at least 90% of the initial 240 runs in a block complete by successfully constructing a locating array. Initial run success rates for the blocks range from 0.9-0.98 with a mean of 0.94. Any runs that do not complete in the maximum iterations allowed are executed again until a locating array is produced for that parameter setting. Despite this, the effect of choice of $maxIterations$ is confounded with blocks.

Statistical analysis is completed in JMP 14.0.0 separately on each response variable and for each block [51], as well as running the analysis on the full factorial with blocks included in the model and without. In all cases, the model included all effects (main effects and all interactions). Results of the JMP Effect Tests are compared side by side and any effect that is not significant in any analysis is removed. The comparison is shown in Figure A.1 in the appendix, and effects that are likely to be real given their significance and prevalence in the blocks are highlighted. The main effect of blocks is not significant, but blocks often appear in significant interactions, suggesting that something about how d, t, k, v or $maxIterations$ are chosen impacts the algorithm. Main effects CHO and α and the second degree interaction of these are significant for all response variables, and main effects *avoid* and *adaptive*-α may be significant for iterations and seconds.

The main effect of CHO on each of the response variables as the mean standardized score across all blocks is shown in Figure 3.4. Greedy CHO results in many more rows than any of the other options, but is also the fastest. For the random, interaction, and column options, there is a general positive correlation of rows, iterations, and seconds, indicating that the longer the algorithm runs, the more rows are added, with resampling the most involved column producing the fewest rows the fastest, on average. This is the opposite of what was originally hypothesized; that is, that the random option is fastest and least accurate. It may be the case that, in general, the time spent to analyze the collision table and choose a "good" column to resample is still faster than the time to conduct extra iterations when a less intelligent choice is made. This trend is broken for the adaptive option; it produces only slightly more rows than the other options and takes more iterations, but does so in substantially less time, reducing the time from the slowest option to the fast

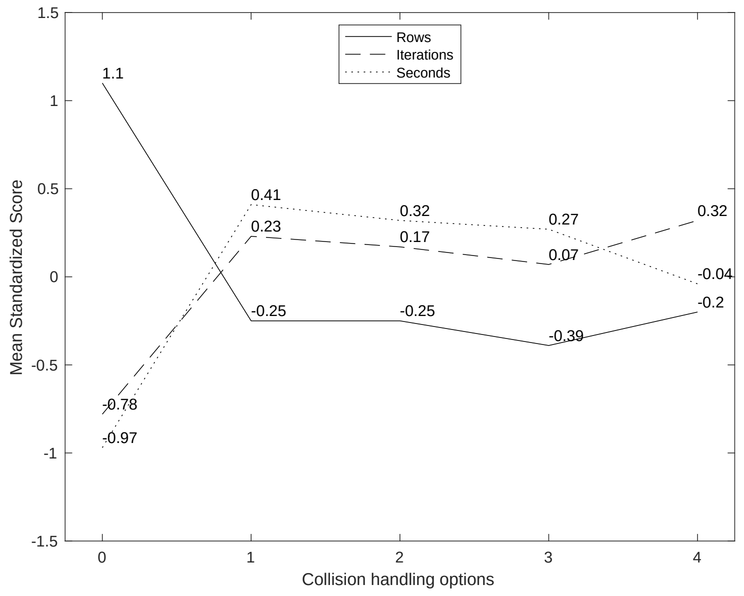


Figure 3.4: Effect of CHO on rows, iterations, and seconds

greedy option by 32.5%. This is mostly expected, but not for the reason hypothesized, which was that the adaptive CHO gets the speed from executing the random and interaction options and the accuracy from executing the interaction and column options. The most involved column surprisingly produces the fastest results of the three middle options, suggesting that the adaptive option is not faster due to using the random and interaction options, but rather the benefit is from checking less of the array before aborting to fix some collision. It might be faster as well as more accurate in the adaptive option to always use the column option while retaining the ability to exit early when the number of collisions is high relative to the proportion of array checked.

The effect of CHO on each of the response variables by block is shown in Figures A.2, A.3, and A.4 in the appendix. The results by block are mostly expected with the exception that the column option outperforms the adaptive option in seconds for

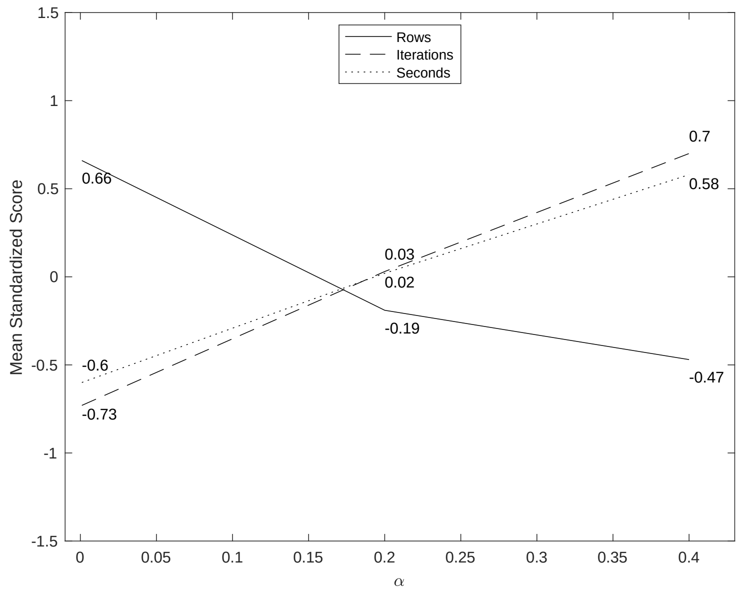


Figure 3.5: Effect of α on rows, iterations, and seconds

the $\mathsf{LA}(1,2,25,6)$ block, but this trend does not extend to the $\mathsf{LA}(1,3,8,4)$ block. The difference in seconds between the options for the $\mathsf{LA}(1,3,8,4)$ block appears to shrink, but there does not appear to be a trend indicating that as t or even d increase that the time benefit decreases for the greedy and adaptive options. In both cases, these results seem likely due to the randomness in the algorithm, and this is likely the culprit for the Blocks$\times CHO$ interaction effect.

The main effect of α on each of the response variables as the mean standardized score across all blocks is shown in Figure 3.5. Again, the trend is expected; increasing α results in fewer rows, but requires more iterations and seconds. The effect of α on the response variables by block are shown in Figures A.5, A.6, and A.7 in the appendix, and they support this conclusion. The Blocks$\times$Alpha interaction effect is significant, and the JMP Interaction Profiler for rows in Figure 3.8 suggests that the effect of α decreases as d and t increase for all response variables. The $CHO\times$Alpha

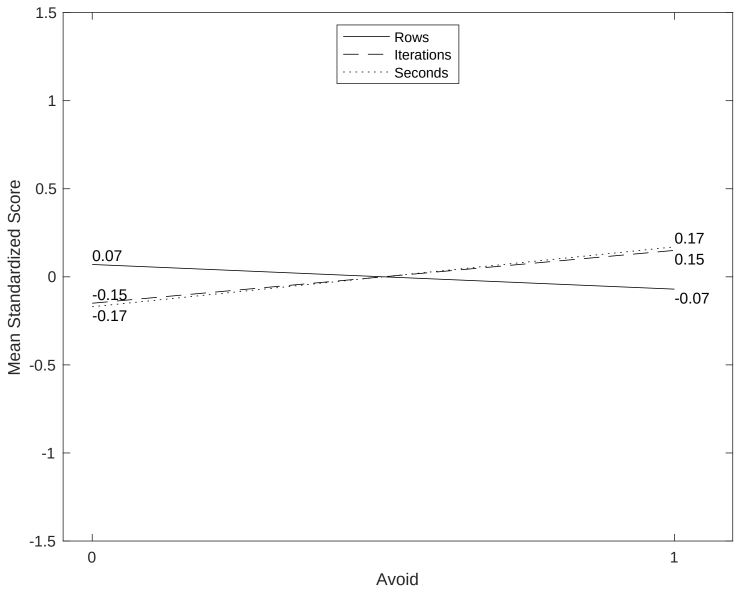


Figure 3.6: Effect of *avoid* on rows, iterations, and seconds

interaction effect is also significant for the obvious reason that the greedy option always finds the first collision and aborts to enter the collision handling phase, so the number of colliding interactions is always 2 until the program terminates with either 2 or 0 colliding interactions. As the z-score for an iteration is always 0.5, the choice of α has no effect for this CHO. The interaction results for seconds and iterations demonstrate the same effects and so are included in the appendix.

The effect of *avoid*, shown in Figure 3.6 is small though probably significant, particularly for iterations and seconds, but it increases with d and particularly t, explaining the Blocks$\times$*avoid* interaction effect. The parameter *adaptive*-α is not likely to have a significant effect (see Figure 3.7), as the average effect of *adaptive*-α is 9%, 10%, and 7% of the effect of α on rows, iterations, and seconds, respectively. Setting these parameters to true seems to result in fewer rows and more time. Both of these results support the algorithmic design choice for these parameters.

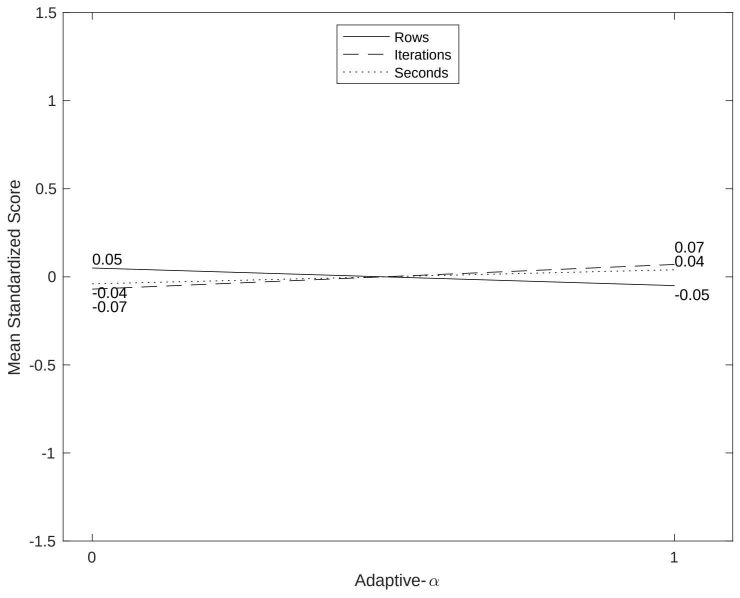


Figure 3.7: Effect of *adaptive*-α on rows, iterations, and seconds

Priority	CHO	α	*adaptive*-α	avoid
Speed	0	any	0	0
Accuracy	3	≥ 0.4	1	1
Both	4	0.2	{0,1}	{0,1}

Table 3.2: Recommended tunable parameters settings by priority

Table 3.2 illustrates the recommended algorithmic settings for each priority.

3.2.2 Comparison to Higher Strength Constructions

PSCR successfully produces arrays that are $(\bar{d}, t)$-locating with fewer rows than from a strength $(t + d)$ covering array when $v > 2$. Constructing an $\mathsf{LA}(1, 2, k, v)$ requires at least as many rows as a strength two covering array. In fact, essentially everything needs to be covered twice, so two times the number of rows in a strength two covering array provides a reasonable lower bound. A strength three covering array is a (1,2)-locating array, so this provides an upper bound on the number of

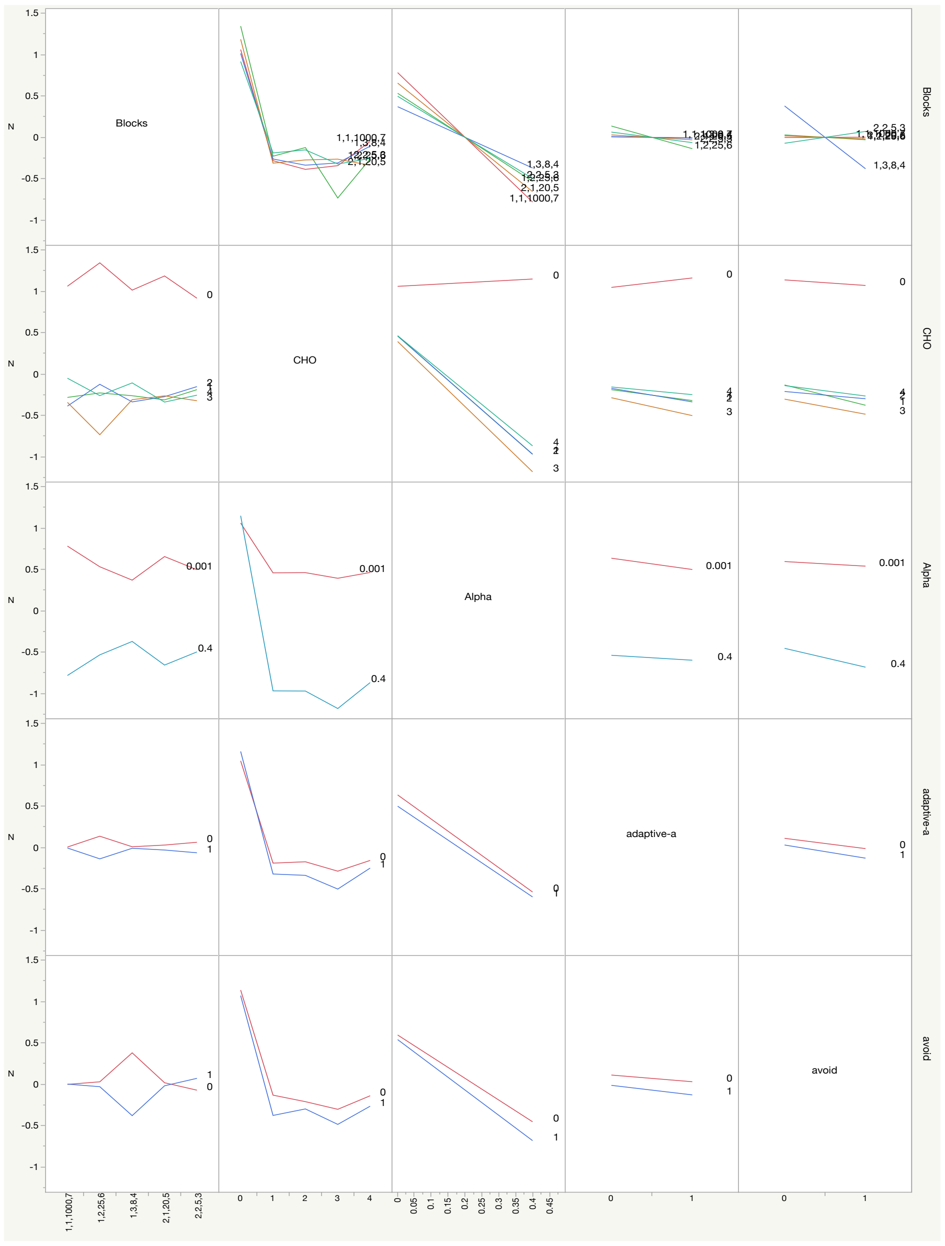


Figure 3.8: 2-way interactions of parameters on response variable rows shown by the JMP Interaction Profiler

rows. The covering array numbers are obtained from [14].

Figure 3.9 shows the rows for an $\mathsf{LA}(1, 2, 50, v)$ as the factor levels vary from $2 \leq v \leq 15$ with the rows scaled logarithmically. Two sets of results for PSCR are presented. The first set for $v \in \{2, \ldots, 10\}$ is generated with the adaptive option for collision handling ($CHO = 4$). The second set for $v \in \{10, \ldots, 15\}$ is generated with the greedy collision option ($CHO = 0$). Both sets use fixed $\alpha = 0.01$. These parameters are chosen to prioritize time efficiency for the larger v values by exiting after the first collision and a low α value, and to balance time and accuracy by choice of a low α value alongside the adaptive option for the smaller v values. The result for $v = 10$ is computed for both sets to provide an idea of how the sets might differ, though the difference could be attributed to randomness in the algorithm.

Figure 3.10 shows the number of rows for an $\mathsf{LA}(1, 2, k, 5)$ when the number of factors varies from $10 \leq k \leq 640$. Once again, the strength two covering array provides a lower bound and the strength three an upper bound for the number of rows. Figure 3.11 shows the rows for an $\mathsf{LA}(2, 1, k, 4)$ for the same number of factors. The strength one covering array is not provided as a lower bound as it has exactly v rows regardless of k. Both data sets are computed with the greedy option ($CHO = 0$) and fixed $\alpha = 0.01$.

In all cases, the algorithm was executed once for each parameter setting and that value is presented. It is likely that this algorithm constructs locating arrays with fewer rows by choosing larger values of α or setting the collision handling to resample the most involved single column. This requires more execution time. Another approach is to replicate the runs and choose the best result.

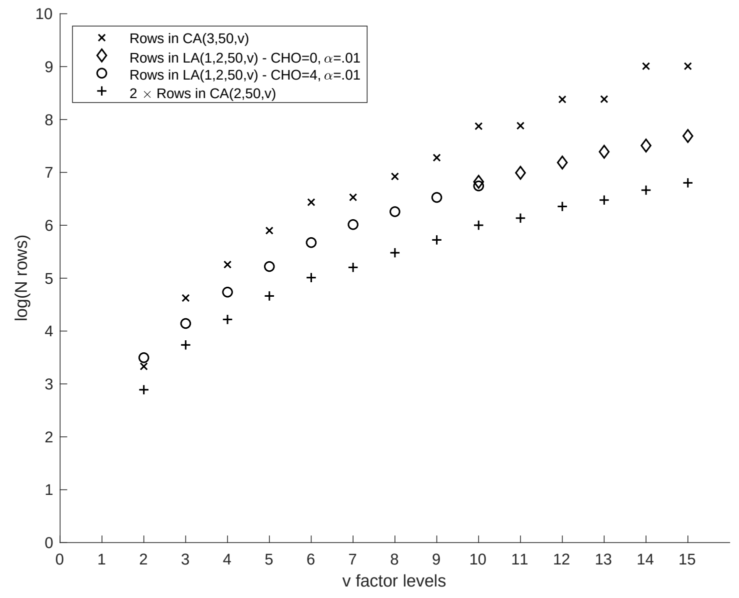


Figure 3.9: Rows in $\mathsf{LA}(1, 2, 50, v)$ with rows in $\mathsf{CA}(2, 50, v)$ and $\mathsf{CA}(3, 50, v)$ as bounds, logarithmic scale

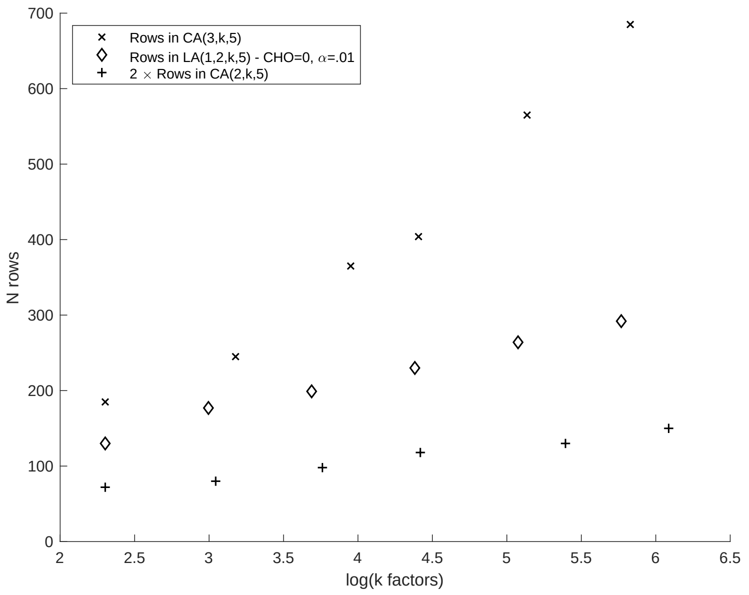


Figure 3.10: Rows in $\mathsf{LA}(1, 2, k, 5)$ with rows in $\mathsf{CA}(2, k, 5)$ and $\mathsf{CA}(3, k, 5)$ as bounds, logarithmic scale

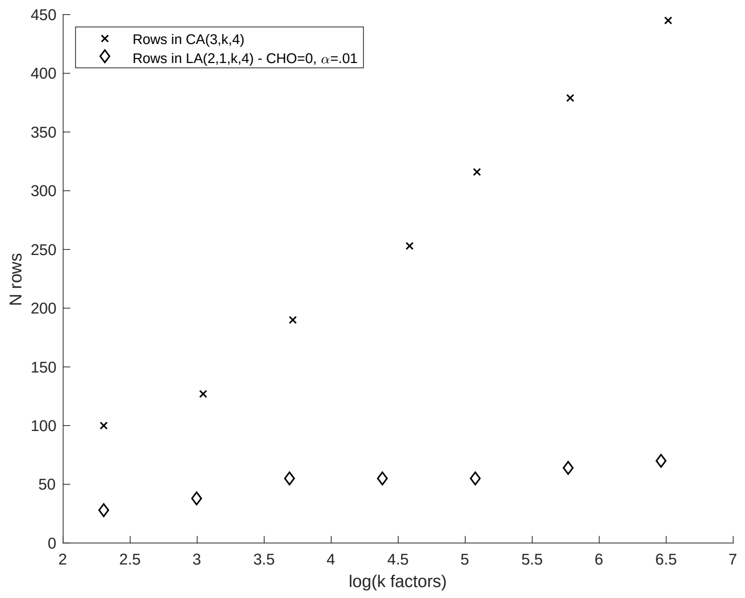


Figure 3.11: Rows in $\mathsf{LA}(2, 1, k, 4)$ with rows in $\mathsf{CA}(3, k, 4)$ as upper bound, logarithmic scale

3.2.3 Comparison to Mixed-Level Locating Array Constructions

An additional advantage of PSCR over higher strength covering array constructions is that, while many of the mathematical constructions that produce covering arrays with fewer rows require a uniform value for all of the factor levels, no modification is required for PSCR to produce mixed-level locating arrays. We compare PSCR against results produced for two systems with mixed factor levels by Seidel, Sarkar, Colbourn, and Syrotiuk [52]. Their goal is to separate interaction effects, and they use a parameter δ for distance to require that δ rows serve as witness to the difference between T_1 and T_2. When $\delta = 1$, this is a locating array by our definition. Greater distance generally requires more rows, and we are interested in creating locating arrays with few rows, so we compare to their results for $\delta = 1$.

The smaller MLA from [52], labeled "Testbed" in their paper, has 24 factors with type $5^9 4^5 3^7 2^3$ using the exponential notation. We were also interested in seeing how the number of initial randomly seeded rows and the maximum allowed iterations affects the result, so we ran a 5×3 full factorial design with 5 replicates to build a $\mathsf{LA}(1, 2, 24, (5^9 4^5 3^7 2^3))$. The *initialN* was chosen from the set $\{75, 85, 95, 105, 115\}$ and *maxIterations* from $\{100, 125, 150\}$. The collision handling option was set to resample the most involved column ($CHO = 3$) and α was set to 0.4 with *adaptive-*α and *avoid* both true. The rate of success increased from 0.27 for *initialN* $= 75$ to 1 for *initialN* $= 115$ and from 0.56 for *maxIterations* $= 100$ to 0.68 for both 125 and 150 *maxIterations*. The smallest locating array, having 110 rows, was produced both by starting with 85 and 95 rows and allowing 125 and 150 iterations, respectively, and took 12 and 11 seconds to complete. The average time of successful runs was 11.54 seconds and 11.72 overall. Analysis of the full factorial in JMP found that *initialN* was significant for the number of rows produced with $p = 0.0007$. Both *initialN* and *maxIterations* were significant for iterations, and *initialN* was significant for seconds, all with $p < 0.0001$. As *initialN* increases, N increases and seconds and iterations decrease, and as *maxIterations* increases, the number of iterations used increases, all as expected.

The larger MLA, "Simulation," has 75 factors with type $10^8 9^1 8^4 7^5 6^{10} 5^4 4^6 3^9 2^{28}$. This problem was deemed too large to run as rigorous of a testing scenario as the smaller case, and the analysis of the smaller case was unsurprising. Through trial and error, we determined that initializing the array with 250 rows produced locating arrays with a reasonable number of final rows in an acceptable amount of time. We set *maxIterations* to values from $\{75, 100, 125\}$, but chose to run 20, 15, and 10 replicates, respectively. The success rate was 0.15 for 75 maxIterations, but 0.20 for 100 and 125, and produced locating arrays with 534 to 554 rows with an average of 542.90. Successful runs used 33 to 96 iterations with an average of 69 and 3526 to

Factors	PSCR	Column Resampling	Local Optimization
$5^{9}4^{5}3^{7}2^{3}$	110	117	114
$10^{8}9^{1}8^{4}7^{5}6^{1}05^{4}4^{6}3^{9}2^{28}$	534	580	532

Table 3.3: Rows in smallest LA by algorithm

7715 seconds with an average of 5297. Allowing more iterations increased the success rate, but also increased the average size of the array and average time to completion. It seems straightforward that allowing fewer iterations weeds out locating arrays that made less progress, producing fewer but better results.

PSCR outperforms the column resampling algorithm on both MLAs and is comparable to the local optimization algorithm, both allowed 1000 resamplings [52], as shown in Table 3.3.

3.2.4 Constructing from an Ingredient Array

Recursive constructions for covering arrays often make use of good small ingredients that are expanded or composed in some way to produce larger solutions either with fewer rows than expected or in less time or, ideally, both. PCSR can build from a candidate array with an arbitrary number of rows and can allow resampling of the ingredient's rows or require that they remain untouched. For example, an $\mathsf{LA}(d-1,t,k,v)$ can be used as an ingredient to build an $\mathsf{LA}(d,t,k,v)$. Alternatively, a strength t covering array can provide the initial coverage, and a strength t' where $t < t' < d + t$ may be even more powerful.

We construct an LA(2,2,7,4) using the following ingredients (notation from [14]):

1. directly from a randomly seeded one row array as a baseline measure;

2. the strength two covering array with the fewest rows known, CA(21;2,7,4) "simulated annealing (Cohen)" [10];

3. the strength three covering array with the fewest rows known, $\mathsf{CA}(88;3,7,4)$ "Chateauneuf-Kreher NRB" [8];

4. the smallest $\mathsf{LA}(1,2,7,4)$ constructed by PSCR, $\mathsf{LA}(54;1,2,7,4)$.

As it turns out, the Chateauneuf-Kreher NRB covering array is also an $\mathsf{LA}(2,2,7,4)$ and as such requires no additional rows. This is unexpected and presents a challenge as that construction does not exist for many cases. To explore the case where an ingredient with few rows is not readily available, we add one additional ingredient:

5. a strength three covering array, $\mathsf{CA}(124;3,7,4)$ "SCPHF Conditional Expectation (CLS)" [19].

We explore both forbidding and allowing resampling of ingredient array rows, except in the direct case which does not use an ingredient and the Chateauneuf-Kreher NRB covering array which is already a locating array. All runs are given 200 iterations with the exception of $\mathsf{CA}(21;2,7,4)$ with ingredient resampling forbidden as it requires 300 iterations to complete. Parameters *adaptive*-α and *avoid* are also set to true, but when ingredient resampling is allowed for $\mathsf{CA}(21;2,7,4)$, *avoid* must be set to false to complete in any reasonable amount of time. (With so few rows, if a column is resampled and the candidate array is no longer a covering array, the algorithm gets stuck at the covering array verification step and may resample millions of times without again passing the check. By allowing the algorithm to add rows after the initial covering array verification, it avoids getting stuck, but may add more rows than in the alternative scenario.) The time reported is just the time in the PSCR algorithm as the time to build each of the ingredients varies and may not be known to us in all cases; however, we report that the ingredients $\mathsf{LA}(1,2,7,4)$ and $\mathsf{CA}(124;3,7,4)$ from an SCPHF are constructed in less than 1 second.

PSCR was executed five times for each combination of ingredient and ingredient row resampling option. The mean rows plotted against mean seconds for each ingredient and ingredient resampling option combination are shown in Figure 3.12. The ten resulting locating arrays with fewest rows are presented in Table 3.4. The three smallest are constructed from the Chateauneuf-Kreher NRB covering array, LA(54;1,2,7,4) ingredient row resampling allowed, and directly; the smallest is an exceptional construction in terms of the small number of rows generated, and the next two are likely found by good random selections. The last seven are constructed from the other covering arrays. The CA(21;2,7,4) ingredient row resampling forbidden is slow to construct, but produces good results probably because it provides the needed coverage in the ingredient rows. The CA(124;3,7,4) ingredient row resampling allowed is an order of magnitude faster and likely produces good results by providing a good estimate of the number of rows for an LA(2,2,7,4) to exist. Surprisingly, with the exception of the Chateauneuf-Kreher NRB covering array and the CA(124;3,7,4) ingredient row resampling forbidden, the row means from all ingredients are extremely close while the seconds means are not, and the fastest method produces a row mean that is only 1.002 times larger than the smallest row mean yet is built from an unremarkable covering array.

Table 3.4: LA(2,2,7,4) arrays with the fewest rows built from ingredients

Ingredient	Ingredient Resampling	Rows	Iterations	Seconds
CA(88;3,7,4)	—	88	1	20
LA(54;1,2,7,4)	Allow	118	103	1980
Direct	—	120	124	2534
CA(21;2,7,4)	Forbid	122	230	7977
CA(21;2,7,4)	Forbid	123	202	6619
CA(124;3,7,4)	Allow	124	11	265
CA(124;3,7,4)	Allow	124	9	215
CA(124;3,7,4)	Allow	124	8	185
CA(21;2,7,4)	Forbid	125	185	6574
CA(21;2,7,4)	Forbid	126	175	6069

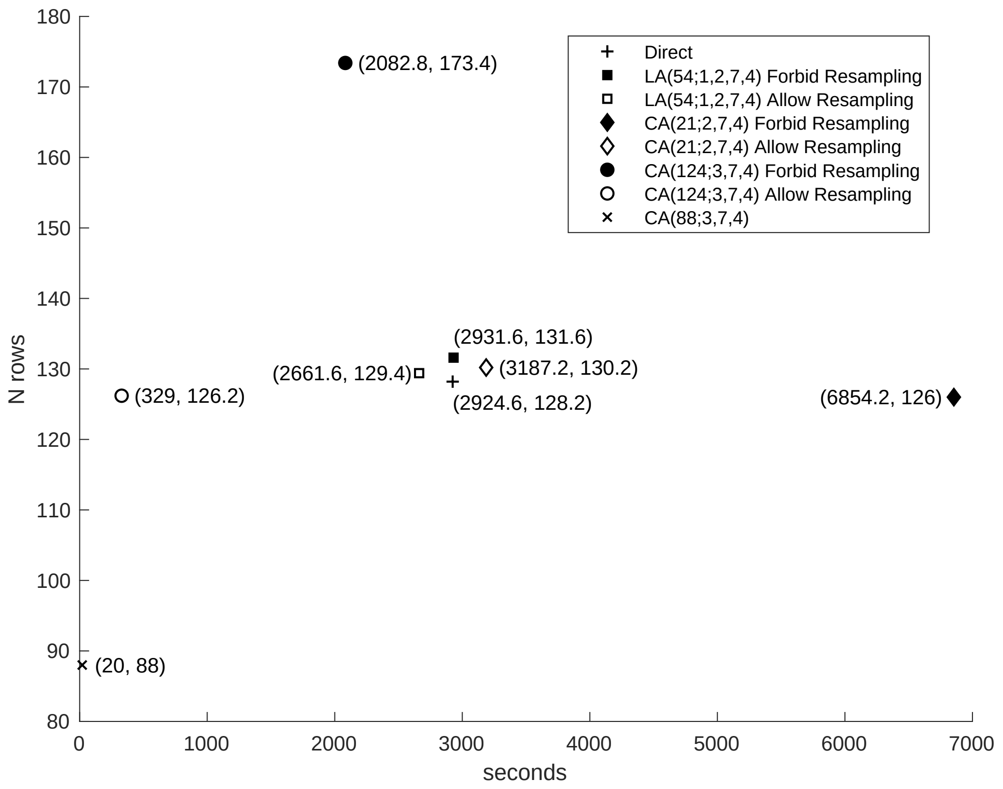


Figure 3.12: Mean rows to seconds in LAs constructed from ingredient arrays

Chapter 4

ANONYMOUS ATTRIBUTE-BASED AUTHORIZATION

The concern with claims of anonymity in attribute-based authorization scenarios based solely on lack of presenting an identity string is detailed in § 1.3 of the introduction, and necessary background for this chapter is presented there. The contribution of this work is to achieve a guaranteed degree of anonymity by requiring that certain properties of attribute distribution hold given a maximum credential size. Policies can be considered disjunctions of conjunctions of attribute values with the most restrictive policy being a single conjunction of many attribute values. Let t be the largest number of attributes in a single conjunction. We present a new combinatorial design, an *anonymizing array*, that ensures that any assignment of values to t attributes that appears in the array appears at least r times. When an anonymizing array is used for subjects registered to a system and policies contain conjunctions of at most t attributes, the system cannot identify the subject using the policy for authorization with greater than $\frac{1}{r}$ probability. This is the first attempt that we are aware of to address the problem of attribute distribution in systems that rely on the identity-less nature of attribute-based authorization for anonymity and, furthermore, to do so using a combinatorial design defined for this purpose.

When the set of subject attributes registered to a system is fixed, an anonymizing array determines the largest conjunction that can be used while achieving the anonymity guarantee r or, equivalently, the guarantee achievable for the largest conjunction. When the set of registered attributes can be appended, anonymizing arrays provide a mechanism to provide higher anonymity guarantees. A key benefit of anonymizing arrays is that they are implementation independent and can be used to enhance subject anonymity of existing systems employing ABAC or ABE access control mechanisms. As opposed to k-anonymous ABAC, our model assumes that all subjects and attributes are known to the system and the anonymity guarantee r is available immediately.

The rest of the chapter is organized as follows. The definition of our new object, an anonymizing array, its relationship to covering arrays, and how to compute the anonymity guarantee are found in § 4.1. Construction algorithms are proposed in § 4.2. An additional desirable property of anonymizing arrays and metrics for comparing two anonymizing arrays with the same parameters are presented in § 4.3. Algorithmic results are presented in § 4.4. How this problem differs from the body of work on κ-anonymity in statistical databases is explored in § 4.5.

4.1 Anonymizing Arrays

4.1.1 Definitions

Consider an array with N rows and k columns and each column i for $1 \leq i \leq k$ has entries from a set of v_i symbols. Such an array is (r,t)-*anonymous* if, when choosing an $N \times t$ subarray, $1 \leq t \leq k$, each row that appears is repeated at least r times. Given an $N \times k$ array $\mathbf{A}$, an $N' \times k$ array $\mathbf{A}'$ is (r,t)-*anonymizing* with respect to $\mathbf{A}$ if $\mathbf{A} \subseteq \mathbf{A}'$ and $\mathbf{A}'$ is (r,t)-anonymous. To write the parameters of an anonymizing array, we use the following notation. In the case of uniform number of symbols in each column, write $\mathsf{AA}(N; r, t, k, v)$. In the mixed case, list the symbols for each column in the array as $\mathsf{AA}(N; r, t, k, (v_1, \ldots, v_k))$ or use exponential notation v_i^j when j columns share the same number of symbols v_i.

Anonymizing arrays are related to covering arrays and this relationship is explored in § 4.1.4. As with covering arrays and locating arrays, the order of the columns and order of the rows is not important. The rows or columns of the array can be shuffled to obtain an equivalent array on the same parameters.

Once an anonymizing array is discussed in the context of an attribute-based system, we refer to a row as an *access profile* and columns as attributes with the symbols in the array as attribute levels (or values). Then, an access profile is an assignment of values to attributes. The *maximum credential size* is t and r is the *anonymity guarantee*. The anonymity guarantee is trivial when $r = 1$ as every credential that appears in an array appears at least once by definition; interesting cases require $r > 1$. Attributes with only one level are trivial as every row contains that attribute, so non-trivial attributes require at least two levels. A consequence of this approach is that the numbers of attribute levels must be finite. If new levels must be added, the prior anonymity guarantee may no longer hold.

Access profiles may correspond to a subject or may exist in the system as *padding*, "dummy" rows created for the purpose of reaching the anonymity guarantee. Because padding rows are added to reach a degree of anonymity not provided by the subject access profiles, care should be taken to construct rows that are not obviously padding. Access profiles need not be unique.

A user credential is defined in [53] as "sets of typed attributes relating to the same topic or structure, e.g., an employee credential may contain an age, address, and salary attribute," and users may select the subset of attributes activated in a session. The term credential typically has a stronger connotation than just a random set of attributes, including some mechanism by which the subject proves possession of those attributes. This may differ across implementations, from providing a physical card loaded with exactly that set of attributes to an application that allows the subject to decide which attributes to provide and which to withhold. Therefore, we use the term *credential* generically to mean the set of attributes that are being provided for a given request. Formally, a credential is a tuple of attribute-value pairs. Authorization decisions are made by the system based on the credential presented rather than the access profile.

Given a set of attributes, each access profile has exactly one credential that corresponds to an assignment of values to those attributes. In context, an array is (r, t)-anonymous if, for all credentials of t or fewer attributes, there are at least r identical credentials distributed among the N access profiles. This provides the anonymity guarantee that, if a subject presents a credential of t or fewer attributes in an access control scenario, there are at least $r - 1$ other access profiles corresponding to the credential. Thus, the system can identify the active subject with not greater than $\frac{1}{r}$ probability.

4.1.2 Constraints

Several types of constraints for covering arrays are treated in [6]. *Hard constraints* are interactions that cannot appear, while *soft constraints* are interactions that need not be covered, but are not illegal. These scenarios also occur in anonymizing arrays and are important to handle when appending padding rows to an anonymizing array to reach the anonymity guarantee r. Some attribute assignments may be impossible. For example, consider employee roles that only exist for a certain facility. One attribute might represent where the employee is stationed, while another is the employee's role. An employee not at facility F may not be assigned role R. This assignment should not exist in the access profiles provided, and appearance of this combination immediately identifies the containing access profile as padding. This is a hard constraint. Other assignments may not be impossible and yet do not appear in the provided anonymizing array. They can be included in padding rows, but if they are included, they must appear r times. These are soft constraints. Every credential not specified in a hard

Table 4.1: Implicit hard constraints for $\mathsf{AA}(2,2,3,2)$

Hard Constraints	Unconstrained Credentials
$\{(a_1,0),(a_2,0)\}$	$\{(a_1,1),(a_2,0)\}$
$\{(a_1,0),(a_2,1)\}$	$\{(a_1,1),(a_2,1)\}$
	$\boldsymbol{\{(a_1,0),(a_3,0)\}}$
	$\boldsymbol{\{(a_1,0),(a_3,1)\}}$
	$\{(a_1,1),(a_3,0)\}$
	$\{(a_1,1),(a_3,1)\}$
	$\{(a_2,0),(a_3,0)\}$
	$\{(a_2,0),(a_3,1)\}$
	$\{(a_2,1),(a_3,0)\}$
	$\{(a_2,1),(a_3,1)\}$

or soft constraint must appear r times.

Hard constraints may give rise to *implicit hard constraints* that cause there to be no feasible solution. For example, consider an anonymizing array $\mathsf{AA}(2,2,3,2)$ with attribute levels $\{0,1\}$. If the only constrained credentials are hard constraints $\{(a_1,0),(a_2,0)\}$ and $\{(a_1,0),(a_2,1)\}$, then any credential containing $(a_1,0)$ is implicitly constrained. There is no way to assign a level to attribute a_2 if attribute a_1 has level 0 without violating the constraints. But any credential not given as a hard or soft constraint must appear twice, so there is no solution. Table 4.1 enumerates the unconstrained credentials as those that do not appear in a constraint and shows in bold the unconstrained credentials that are implicitly constrained by the hard constraints. If the implicit constraints are defined as constraints themselves, a solution without the hard constraints exists and is given in Figure 4.1. If the hard constraints are instead soft constraints, then a solution also exists, as these credentials can appear as needed to ensure unconstrained credentials appear, provided they appear twice. (See Figure 4.4 in § 4.3 for an example).

	a_1	a_2	a_3
1	1	0	0
2	1	0	0
3	1	0	1
4	1	0	1
5	1	1	0
6	1	1	0
7	1	1	1
8	1	1	1

Figure 4.1: $\mathsf{AA}(8; 2, 3, 2)$ with hard constraints

4.1.3 Anonymizing Array Example

Consider a system at a university that has the following attributes and values.

1. *Role*=$\{$*faculty*, *graduate*, *undergraduate*$\}$

2. *Job*=$\{$*instructor*, *grader*$\}$

3. *Department*=$\{$*CS*, *EE*$\}$

4. *Semester*=$\{$*Spring*, *Fall*$\}$

The access profiles provided to the system are in array **A** (Figure 4.2). The system requires the hard constraints (*faculty*,*grader*) and (*undergraduate*, *instructor*). **A** also has the soft constraint (*graduate*,*grader*). **A** is $(1, 2)$-anonymous but not $(2, 2)$-anonymous as there is only one access profile with the credential (*CS*,*grader*).

Array **B** is an anonymizing array for **A**. **B** is $(2, 2)$-anonymous. The first six access profiles of **B** are **A** and then six padding rows have been added. Twelve rows are required, as the largest number of unconstrained credentials for a pair of attributes is six and each must appear twice. The soft constraint appears in **B** twice, but another anonymizing array for **A**, $\mathbf{B}'$, has the credential (*graduate*, *instructor*) in rows 9 and 10 instead.

	Role	Job	Department	Semester
1	faculty	instructor	CS	Spring
2	faculty	instructor	EE	Fall
3	graduate	instructor	CS	Spring
4	graduate	instructor	EE	Fall
5	undergraduate	grader	CS	Fall
6	undergraduate	grader	EE	Spring

Figure 4.2: Array **A**, an $\mathsf{AA}(6; 1, 2, 4, (3, 2^3))$

	Role	Job	Department	Semester
1	faculty	instructor	CS	Spring
2	faculty	instructor	EE	Fall
3	graduate	instructor	CS	Spring
4	graduate	instructor	EE	Fall
5	undergraduate	grader	CS	Fall
6	undergraduate	grader	EE	Spring
7	faculty	instructor	CS	Fall
8	faculty	instructor	EE	Spring
9	graduate	grader	CS	Fall
10	graduate	grader	EE	Spring
11	undergraduate	grader	CS	Fall
12	undergraduate	grader	EE	Spring

Figure 4.3: Array **B**, an $\mathsf{AA}(12; 2, 2, 4, (3, 2^3))$ that is $(2, 2)$-anonymizing for **A**

B is not $(3, 2)$-anonymous, as the credential $(faculty, CS)$ does not appear at least three times. It is not $(2, 3)$-anonymous, as the credential $(graduate, grader, Fall)$ appears only once. If $graduate \land grader \land Fall$ is a policy used for an authorization decision, the access profile in row 9 is uniquely identified.

4.1.4 Relationship to Covering Arrays

Anonymizing arrays are similar to covering arrays with constraints and higher coverage. The primary difference due to application is in the desired homogeneity property (see § 4.3), but also in how constraints are treated. For covering arrays, the norm is to define the interactions that must not appear (hard constraints), then to define the interactions that might appear (soft constraints, possibly further divided into “don't care” and “avoid”), and then to derive the interactions that must appear. For anonymizing arrays, the access profiles provided define the credentials that must appear (unconstrained credentials). The system specification defines the credentials that must not appear (hard constraints), and the soft constraints are defined to be the remaining credentials that are in neither set. Given an anonymizing array without a supplied set of constraints, it may be impossible to deduce which of the non-appearing credentials are soft and which are hard constraints. The same difficulty arises differentiating which of the appearing credentials are soft constraints and which are unconstrained. Care must be taken when converting between covering arrays and anonymizing arrays that constraints are categorized correctly.

Many construction algorithms exist for building covering arrays, though few explicitly include constraint handling or higher coverage requirements. The following non-exhaustive list of relationships elucidate how to use covering array constructions to build anonymizing arrays.

Theorem 2. *A mixed-level covering array $\mathsf{MCA}_\lambda(t, k, (v_1, \ldots, v_k))$ with hard constraint set $\mathcal{H}$ is an $\mathsf{AA}(\lambda, t, k, (v_1, \ldots, v_k))$ with hard constraint set $\mathcal{H}$ and all other credentials appearing.*

Proof. Covering arrays are usually described without explicitly specifying λ, the number of times a t-way interaction is covered, in which case $\lambda = 1$ is implied. Every t-way interaction that appears in the covering array λ times is a credential that appears λ times in the corresponding anonymizing array. The interactions in $\mathcal{H}$ never appear in the covering array so they never appear in the anonymizing array, as required. $\square$

Extending the definition of soft constraints to higher λ for covering arrays does not appear to be addressed in the literature. It seems counterintuitive to require that if a “don't care” or even an “avoid” interaction appears once in the covering array that it must then appear λ times, as this might add more rows to the covering array than are needed. If soft constraints can appear zero or more times but are not required to appear $\lambda > 1$ times in the covering array, then soft constraints must not be present in a covering array used as an anonymizing array by Theorem 2. There must also exist a mapping of soft constraints in the anonymizing array onto either hard constraints in the covering array if they do not appear or onto unconstrained interactions if they do.

Theorem 3. *If there exists an $\mathsf{MCA}(t, k, (v_1, \ldots, v_k))$ with hard constraint set $\mathcal{H}$ and soft constraint set $\mathcal{S}$, then there exists an $\mathsf{AA}(r, t, k, (v_1, \ldots, v_k))$ with $\mathcal{H}$ and $\mathcal{S}$.*

Proof. Make r vertical copies of the covering array. No interaction of $\mathcal{H}$ appears in the covering array, so none of these credentials appear in the anonymizing array. Any interaction of $\mathcal{S}$ that appears in the covering array at least once appears in the anonymizing array at least r times, and the rest never appear. All unconstrained credentials, $\mathcal{C} \setminus \{\mathcal{H} \bigcup \mathcal{S}\}$, appear at least once in the covering array and at least r times in the anonymizing array. $\square$

Theorem 4. *An $\mathsf{MCA}(t, k, (v_1, \ldots, v_k)), v = \min_{i=1}^{k}(v_i)$ with no constrained interactions is an $\mathsf{AA}(v, t-1, k, (v_1, \ldots, v_k))$ with no constrained credentials.*

Proof. In the mixed-level covering array without constraints, a $(t-1)$-way interaction appears at least v_i times, once with each of the v_i symbols in the t-th column of the t-way interaction including those $t-1$ columns. Then every $(t-1)$-way interaction appears at least v times where v is the smallest number of symbols for a column in the covering array. This covering array is then an anonymizing array with $r = v$. $\square$

Theorem 5. *If there exists a covering array $\mathsf{CA}(t, k, v)$ with a set of hard constraints $\{(c_1, \sigma_1), \ldots, (c_{t-1}, \sigma_{t-1}), (c_x, \sigma_y)\}$ for each column symbol pair (c_x, σ_y) with column $c_x \in \mathcal{K} \setminus \{c_1, \ldots, c_{t-1}\}$ and $\sigma_y \in \Sigma_x$, the symbol set of c_x, then there exists an anonymizing array $\mathsf{AA}(v, t-1, k, v)$ with $\{(c_1, \sigma_1), \ldots, (c_{t-1}, \sigma_{t-1})\}$ as a hard constraint.*

Proof. To guarantee that the constrained credential with $t-1$ attributes never appears in the anonymizing array, it must be the case that no t-way interactions of which it is a subset appeared in the covering array. The coverage for all unconstrained credentials follows from Theorem 4. To extend the proof to soft constraints, soft constraints in the anonymizing array must be mapped to unconstrained interactions or to hard constraints in the covering array. $\square$

4.1.5 Computing the Anonymity Guarantee

An inverse relationship exists between t and r so that as the credential size increases, the number of repetitions of the credential in an array must either stay the same or decrease.

Theorem 6. *Given an array $\mathbf{A}$ that is (r, t)-anonymous and not $(r+1, t)$-anonymous, for every $t \leq t' \leq k$ for which $\mathbf{A}$ is (r', t')-anonymous, it must be the case that $r' \leq r$.*

Proof. Pick the credential c that appears the fewest number of times in $\mathbf{A}$ and let r be the number of times c appears. $\mathbf{A}$ is (r, t)-anonymous by definition and is not $(r+1, t)$-anonymous. Look at the set of rows in which c appears. Choose any credential c' that contains c. The rows in which c' appears must be a subset of the rows in which c appeared. Then for $t' \geq t$, if $\mathbf{A}$ is (r', t')-anonymous, then $r' \leq r$. $\square$

Corollary 6.1. *An array that is (r, t)-anonymous is also (r, t')-anonymous for every $t' < t$.*

Proof. Pick any credential, c, of size t. It appears in at least r rows. Pick any t'-subset of c. It appears in at least these rows. □

Given an anonymizing array $\mathbf{A}$ and a maximum credential size t, Algorithm 5 computes the maximum r for which $\mathbf{A}$ is (r, t)-anonymous. It also serves to check an anonymizing array. If a hard constraint is found, it returns 0. Recall that soft constraints may appear zero or at least r times. To compute all pairs r and t for $\mathbf{A}$, Algorithm 5 can be executed for increasing values of t until it returns $r = 1$.

Algorithm 5: Compute Anonymity Guarantee

```
input  : $\mathbf{A}$ an $N \times k$ array, $t$
output: $r$
begin
    for each of the $\binom{k}{t}$ sets of columns of $\mathbf{A}$ do
        Scan the $N \times t$ subarray of $\mathbf{A}$ and store the count of each credential
        if a hard constraint has count $> 0$ then
            Return 0
        else
            Set $r$ to be the smallest non-zero count so far
    Return $r$
```

4.2 Anonymizing Array Construction Algorithms

In situations where attributes can be assigned to access profiles for the purpose of anonymous authorization rather than existing as real-world attributes of subjects, as in key distribution, an anonymizing array is built from scratch. In other situations, the attributes correspond to immutable properties of the subjects themselves, such as name or birth year, and so we might be presented with a fixed set of access profiles and asked to compute the anonymity guarantee or to provide an anonymizing array to anonymize a set of access profiles to meet a particular guarantee.

We hypothesize that mathematical direct constructions for covering arrays are not particularly useful in building anonymizing arrays due to constraints. Random constructions that generate an array that has high probability of satisfying the desired properties do not work well for covering arrays, and they could be particularly disastrous in access control scenarios where mistakes are expensive. An approach that generates an anonymizing array with high probability still has a non-zero probability of failing to have the desired property. Adapting covering array search techniques that can accommodate constraints to build anonymizing arrays seems more promising.

4.2.1 Moser-Tardos-style Column Resampling (MTCR) Algorithm

A Moser-Tardos-style column resampling algorithm (MTCR) can be used for anonymizing array construction [42]. Two implementation issues concern how to handle constraints and how to determine when to add a row if building up from fewer rows than required, as done in Chapter 3. The column resampling algorithm hits the iteration limit if hard constraints cause implicit constraints that result in no possible solution. This is not ideal as the case of no solution cannot be distinguished from not allowing enough iterations. A better approach is to conduct a feasibility check first, such as by encoding constraints as a satisfiability problem and utilizing a SAT solver such as described in [9], and only running the algorithm if a solution exists. However, determining if a solution exists in the presence of hard constraints is known in general to be NP-complete [6].

The fastest way to check against constraints, as usual, comes with a storage tradeoff. If $\mathcal{T}$ is the set of $\binom{k}{t}$ t-subsets of columns, create a two-dimensional array of possible credentials $\sum_{T \in \mathcal{T}} \prod_{i \in T} v_i$. The position of T in colexicographic ordering of the sets is the *rank*. Indexing is straightforward by computing the rank and credential value, and checking each credential against the constraint array is done in constant time. Unconstrained credentials are indicated by 1, soft constraints by 0, and hard constraints by -1. An alternative is to form a set of lists, one for each rank, and store each constrained credential in the appropriate list. Credentials have a natural ordering, so comparing each item in the list against the credential being checked not just for equality but also for relative ordering allows an exit once the list item exceeds the credential. When the number of constraints is small, the list provides storage savings and checking might not incur much more computation time than the array method. However, as constraints need to be checked for every row of a rank, it is likely that constant time lookup of the array is worth the extra storage. MTCR is implemented with the array.

Estimating the number of rows needed is not necessarily straightforward, so one option is to build up rows until the coverage requirement is met or an iteration limit is reached. When a set of real access profiles are provided and the algorithm is executed to add padding rows to reach the anonymity guarantee, the algorithm forbids resampling of the provided rows. When starting from scratch, the candidate array can begin with zero rows or a starting number can be computed by taking $r \max_{rank=1}^{\binom{k}{t}}(\text{credentials for } rank - \text{constraints for } rank)$, and these rows can be randomly populated.

Determining how often to add a row inside the algorithm is challenging. Adding too quickly may produce more rows than needed while adding too conservatively may cause the iteration limit to be reached. The number of resamplings conducted since the last time a row was added can be used to estimate progress. The distinction between a bad event due to the presence of a constraint and a bad event due to lack of r coverage of non-constrained credentials needs to be made. Having too few rows can be a contributing factor to lack of r coverage, but not to presence of a constraint. In fact, the more rows, the more likely a constraint is to appear somewhere. If the number of resamplings is only incremented when the resampling is due to lack of coverage, it can be better used to gauge that not enough rows exist to meet the coverage requirements. A fixed ordering is used by going rank by rank, so it is expected, though not guaranteed, that if the rank being checked is higher, fewer bad events still exist in the array than if the rank is lower. The number of resamplings needed to add a row can then be proportional to the amount of array still left to check, as $rank * threshold$, so that rows are added more readily when bad events are encountered early in checking. The *threshold* can be chosen as a fixed value or proportional to the number of subsets of columns or based on the size of the problem.

MTCR is given in Algorithm 6.

4.2.2 Conditional Expectation Heuristic Search (CEHS) Algorithm

An approach that combines ideas from conditional expectation and a heuristic to avoid constraints can also be used [5, 19]. This greedy algorithm, Conditional Expectation Heuristic Search (CEHS), builds an anonymizing array row by row. As with the column resampling approach, CEHS can be used to append padding rows to a set of access profiles that do not yet meet the anonymity guarantee r. When the algorithm begins, hard constraints, soft constraints, and unconstrained credentials are categorized and marked, but once a soft constraint appears in a row, it gets moved to the unconstrained credential category. This allows the algorithm to not care about placing soft constraints until they appear, after which they must be covered just like an unconstrained credential.

Call a credential not yet covered if it is unconstrained and appears fewer than r times. The expectation for a row is the number of not yet covered credentials that are covered if the symbols are assigned to columns randomly. A conditional expectation algorithm operates as follows. Given a partial row with some columns fixed to symbols and some columns free, randomly select a free column. For the

Algorithm 6: Moser-Tardos-style Column Resampling (MTCR)

```
input : A an $N \times k$ array, $r$, $t$, $V$ a $1 \times k$ array of attribute levels,
        $numCreds = \prod_{i \in T} V[i]$, $Constraint$ a $1 \times numCreds$ array with
        constraint status of all credentials, $maxIterations$
output: A or $\emptyset$
begin
    iteration = 0
    numResamplings = 0
    repeat
        badEventConstraint = badEventCoverage = false
        for each of the $\binom{k}{t}$ subsets of $t$ columns, $T$, AND while no bad events do
            Create $Count$, a $1 \times numCreds$ array
            for each credential, c,in the $N \times t$ subarray of A with columns from $T$ do
                if $Constraint[c] \geq 0$ then
                    Increment Count[c]
                else
                    badEventConstraint = true
            for $1 \leq c \leq numCreds$ do
                if $Count[c] < r$ then
                    if $Constraint[c]$ is 1 for any $c$ then
                        badEventCoverage = true
                        Increment numResamplings
                        if $numResamplings \geq rank * threshold$ then
                            Add a row to A and increment $N$
                            numResamplings = 0
                    else if $0 < Count[c]$ and $Constraint[c]$ is 0 then
                        badEventConstraint = true
            if any bad event then
                Resample all $t$ columns of $T$
        if no bad events then
            return A
        Increment iteration
    until iteration is equal to maxIterations
    Iteration limit reached without no bad events, so return $\emptyset$
```

symbols of that column, choose one that does not reduce the expectation for the row. That is, if the symbol is placed, at least as many credentials are covered for the row with the remaining symbols in free columns chosen randomly. Computing the best symbol for a column based on what has been chosen already, rather than just one that is at least as good as the expectation, may improve the performance in terms of coverage for the row.

Given a row with $i-1$ columns fixed to symbols and the rest free, choose a column i randomly and consider the v_i symbols to place in column i. The best symbol covers the most credentials that have not yet been covered r times without violating a hard constraint. Let $\mathcal{T}_i$ be the set of $\binom{k-1}{t-1}$ sets of t columns involving i, and $\mathcal{C}_T$ the set of possible credentials for a t-set of columns, T. Suppose column i is fixed to symbol σ. Define

$$value(i,\sigma) = \sum_{T\in\mathcal{T}_i} \sum_{c\in\mathcal{C}_T} \lambda(c)P(c)$$

where $P(c)$ is the probability of credential c appearing

$$P(c) = \frac{\text{ways to cover c}}{\text{ways to fix free columns of T}}$$

and $\lambda(c)$ is related to the coverage status of c. Temporarily ignoring hard constraints, consider the definition

$$\lambda(c) = \begin{cases} 1 \text{ if c covered fewer than } r \text{ times,} \\ 0 \text{ if c covered at least } r \text{ times or c is a soft constraint.} \end{cases}$$

The expected number of not yet covered credentials newly covered by placing σ in i is $value(i,\sigma)$. The best symbol is one that maximizes $value(i,\sigma)$. Ties can be broken randomly.

Another option is to prioritize the credentials that have been covered fewer times over those that have been covered more often. Refine the prior definition of $\lambda(c)$ to

$$\lambda(c) = \begin{cases} \frac{r-\text{times } c \text{ covered}}{r} \text{ if } c \text{ is unconstrained,} \\ \\ 0 \text{ if } c \text{ is a soft constraint.} \end{cases}$$

An opportunity to avoid hard constraints is available in two places within CEHS. The last chance to ensure that a hard constraint is not placed in the array is when considering t columns with $t-1$ columns fixed and one is free. If fixing a symbol σ in column i causes a violation of a hard constraint, the cost of placing symbol σ in column i for that credential should be sufficiently high as to outweigh the benefit gained by covering other credentials that have symbol σ in column i. There are $\binom{k-1}{t-1} - 1$ other t sets involving column i, each with some number of credentials. At most, a t-set contributes a benefit of 1, because at best only one credential is newly covered in that row for that t-set of columns. Then the most positive value a symbol receives from the other credentials is $\binom{k-1}{t-1} - 1$. Assign $\lambda = -\binom{k-1}{t-1}$ to forbid placing a symbol that violates a hard constraint. A positive $value(i, \sigma)$ is a *benefit* while a negative $value(i, \sigma)$ is a *cost*.

An algorithm that does not wait until the last chance to avoid the placement of the hard constraint, but also handles hard constraints when considering symbols for a t-set in which fewer than $t-1$ columns are fixed may have better performance. If hard constraints simply do not provide benefit, that is $\lambda = 0$, there is a risk of placing symbols in columns fixed earlier that result in eventually denying a symbol σ in column i while some later unconstrained credential requires σ in i. For example, consider the case that there is one unconstrained credential left to cover, $\{(c_2, 2), (c_3, 3)\}$, but $\{(c_1, 1), (c_2, 2)\}$ is a hard constraint. (The $\star$ symbol indicates a "don't care" position.)

c_1	c_2	c_3	status
1	2	$\star$	hard constraint
$\star$	2	3	not yet covered

If hard constraints do not incur a cost until they are guaranteed to be placed, symbol 1 may be placed in column c_1 when some other symbol, 0, may provide no benefit, but may be a better choice because it is not involved in a hard constraint. When considering c_2, symbol 2 is now forbidden and then $\{(c_2, 2), (c_3, 3)\}$ is not covered by the row. If columns are chosen in a fixed order and ties are handled deterministically, the algorithm runs forever on this sequence of choices. One reason that ties are broken randomly and the order of columns is chosen randomly is so that $(c_1, 0)$ is eventually chosen with non-zero probability and the search terminates. However, this might cause more rows to be added than are needed due to relying on chance. Assigning a low cost to future hard constraint violations drives the search so that if there is a choice of symbol that costs nothing, it is selected instead and the future hard constraint is avoided earlier.

Choosing the right cost is important, as many hard constraints acting together can outweigh the benefit of an unconstrained credential. If $\{(c_1, 1), (c_2, 2)\}$ and $\{(c_1, 1), (c_3, 3)\}$ are hard constraints and $\{(c_1, 1), (c_4, 4)\}$ is an unconstrained credential, 1 must be selected in c_1 so that $\{(c_1, 1), (c_4, 4)\}$ can be covered. The cost of potential hard constraints must be low enough for 1 to be selected at least r times.

c_1	c_2	c_3	c_4	status
1	2	$\star$	$\star$	hard constraint
1	$\star$	3	$\star$	hard constraint
1	$\star$	$\star$	4	not yet covered

The cost of a potential hard constraint should drive the algorithm away from placing symbols that eventually result in a hard constraint when other options are better, but not to the extent that a required credential cannot be covered. The absolute value of the highest total cost must be less than the absolute value of the lowest benefit. The lowest benefit of placing symbol σ in column j occurs when there is one credential to be covered one remaining time with the highest number of symbols, $v = \max_{i=1}^{k}(v_i)$. The probability of being placed is lowest when all other columns in the t-set are still free, assuming j is fixed to σ. Then $P(c) = \frac{1}{v^{t-1}}$ and $\lambda(c) = \frac{1}{r}$, so the benefit is $\frac{1}{rv^{t-1}}$.

We now compute the highest possible cost. Again, suppose that j is fixed to σ. As we are considering the case of many potential hard constraints acting against an uncovered unconstrained credential with lowest benefit, there is one t-set with an uncovered unconstrained credential. For the highest cost, the other t-sets involving the column being decided, j, have $\binom{k-1}{t-1} - 1$ potential hard constraints and one free column. (If all columns are fixed when considering placing σ in j, any hard constraint violations are not potential but rather immediate.) For each t-set, let w be the number of symbols for the free column. There are w credentials with symbols matching the $t-1$ fixed columns, and each is chosen with probability $P(c) = \frac{1}{w}$. The highest cost occurs when all are hard constraints, ignoring for a moment that this leads to an infeasible solution, so each t-set contributes at most $w\frac{1}{w}\lambda$, so the total cost is $(\binom{k-1}{t-1} - 1)\lambda$.

The value of λ must ensure that $|(\binom{k-1}{t-1} - 1)\lambda| < \frac{1}{rv^{t-1}}$. Set $\lambda = \frac{-1}{(\binom{k-1}{t-1}-1)ry^t}$, $y = max_{i=1}^{k}(v_i)$. When $y \geq v$,

$$\left|\left(\binom{k-1}{t-1} - 1\right)\frac{-1}{(\binom{k-1}{t-1} - 1)ry^t}\right| = \left|\frac{-1}{ry^t}\right| < \left|\frac{1}{rv^{t-1}}\right|.$$

The final definition for $\lambda(c)$ is

$$\lambda(c) = \begin{cases} \frac{r\text{−times } c \text{ covered}}{r} \text{ if } c \text{ is unconstrained,} \\ \\ 0 \text{ if } c \text{ is a soft constraint,} \\ \\ \frac{-1}{(\binom{k-1}{t-1})-1)ry^t} : y = \max_{i=1}^{k}(v_i) \text{ if } c \text{ is a hard constraint with at least 1 free column,} \\ \\ -\binom{k-1}{t-1} \text{ if } c \text{ is a hard constraint with no free columns.} \end{cases}$$

As with the Moser-Tardos-style algorithm, a feasibility check should be conducted beforehand, as some scenarios can still result in infinite looping. For example, consider this set of credentials where the number of levels for c_2 is 2.

c_1	c_2	c_3	status
1	1	$\star$	hard constraint
$\star$	2	3	hard constraint
1	$\star$	3	not yet covered

The cost of placing 1 in c_1 does not outweigh the benefit as an uncovered credential, $\{(c_1, 1), (c_3, 3)\}$, still exists. The algorithm is forced to place 2 in c_2 so it cannot place 3 in c_3. In this case, the order of columns does not matter; it cannot violate a hard constraint if there is another option so it repeatedly avoids one or the other hard constraint, never being able to cover the final credential.

Despite all efforts to make good local decisions, the algorithm does not have complete lookahead. A series of local decisions based on the ordering of columns in an execution can lead to every symbol option for a column resulting in the placement of some hard constraint even if there is a solution for the anonymizing array problem. In this case, the algorithm aborts and can be run again.

The complete CEHS algorithm is given in Algorithm 7.

4.2.3 Post-optimization for Row Reduction

MTCR and CEHS are employed to add padding rows to reach an anonymity guarantee, but they may add more rows than necessary. The post-optimization strategy using necessity analysis [44] naturally extends to attempt to reduce the number of padding rows by treating the anonymizing array as a covering array of higher λ with constraints. Though we do not develop the extension fully here, we suggest how it can be done. Consider the padding rows. Each time a credential appears that is not yet covered r times, mark all t positions of the credential in the row as necessary. After all $\binom{k}{t}$ credentials are checked, all positions that are not necessary can be

Algorithm 7: Conditional Expectation Heuristic Search (CEHS)

```
input  : A an $N \times k$ array, $V$ a $1 \times k$ array of attribute levels, $r$, $t$,
         $numCreds = \prod_{i \in T} V[i]$, $Constraint$ a $1 \times numCreds$ array with
         constraint status of all credentials
output: A, $N$
begin
    Construct a $1 \times numCreds$ array R to hold the remaining coverage of
     each credential, $c$
    Set R[c] to the number of remaining appearances required for $c$
    while $R[c] > 0$ for some $c$ do
        Add a row with all columns free
        Increment $N$
        while some column is free do
            Randomly select a column index $i$
            for each symbol $\sigma \in [V[i]]$ do
                for each of the $\binom{k-1}{t-1}$ subsets of columns, $T$, involving $i$ do
                    for each of the $\prod_{j \in T} V[j]$ credentials, c, of $T$ do
                        $P(c) = \frac{\text{ways to cover c}}{\text{ways to fix free columns of T}}$
                        if $P(c) = 1$ and c is a hard constraint then
                            $\lambda(c) = -\binom{k-1}{t-1}$
                        else if c is a hard constraint then
                            $\lambda(c) = \frac{-1}{(\binom{k-1}{t-1}-1)ry^t} : y = \max_{i=1}^{k}(v_i)$
                        else
                            $\lambda(c) = \frac{R[c]}{r}$
                        Add $\lambda(c)P(c)$ to value$(i, \sigma)$
            Place symbol $\sigma$ in column $i$ that maximizes $value(i, \sigma)$
        for each of the credentials, c, appearing in row $N$ do
            Update R[c]
            if c is a soft constraint but c appears in the row then
                Change $Constraint[c]$ to unconstrained
                Set $R[c] = r - 1$
            else if c is a hard constraint then
                Return $\emptyset$
    Return A, $N$
```

marked $\star$. If a padding row is composed only of $\star$ symbols, the row can be removed without reducing the anonymity guarantee. If the randomized algorithm from [44] is used and entries are chosen at random to replace the $\star$ entries, a check must be conducted to ensure that hard constraints are not violated in the array, or that if new soft constraints appear, they appear r times.

4.3 Homogeneity in Anonymizing Arrays

4.3.1 Designing Metrics

In addition to requiring that all credentials appear either zero or at least r times, we prefer anonymizing arrays that do not contain groups of highly similar access profiles. If an access profile shares all of its $\binom{k}{t}$ credentials with a group of $r-1$ other access profiles, when any of these credentials is used, the system can identify someone in the group as the subject. Even if no one subject can be identified with greater than $\frac{1}{r}$ probability, this is likely undesirable as it allows access requests using these credentials to be linked and behavior of the group to be tracked. An anonymizing array in which access profiles share different credentials with different groups is preferable to another anonymizing array with the same anonymity guarantee in which access profiles share credentials with the same groups frequently. Lastly, the size of the groups matters as well. Tracking the behavior of a group of $2r$ access profiles that share all credentials is less targeted to one access profile than tracking the behavior of a group of r access profiles. An ideal solution captures the interplay of these two concerns in a single metric.

Consider a multi-hypergraph representation of an anonymizing array. There are N vertices, one for each of the access profiles. Each vertex has degree $\binom{k}{t}$, and a hyperedge represents the credential possessed by the access profile for this t set of attributes connecting all of the access profiles that share this credential. When all attributes have v levels, there are up to $\binom{k}{t}v^t$ edges. (When attributes have mixed levels, replace v^t with $\prod_{j=1}^{t} v_j$ for each of the $\binom{k}{t}$ sets of attributes. We use constant v for simplicity in the counting throughout this section, but all of the results extend naturally.) It is a multigraph as the same vertices can be connected by multiple hyperedges if the same group shares more than one credential. Our metric, then, measures the number of edges that connect a vertex to the same set of vertices and the sizes of those sets.

A clustering coefficient can be described as the measure of neighbors of a vertex that know each other; if vertex v has m neighbors, the clustering coeffcient is the fraction of the $m(m-1)/2$ possible edges that actually exist between those vertices [61]. The clustering coefficient describes the tendency of vertices to form cliques. Consider, however, a scenario where all N access profiles share a universal credential. This measure masks any other behavior exhibited in the rest of the graph. Additionally, work to extend this concept to hypergraphs ignores multi-edges, considering them as a simple edge [23]. The metric must measure how many times a vertex is connected to a group of vertices, not just how many vertices it is connected to, so the multi-edges cannot be ignored.

Structural similarity has the same issue; it measures how many neighbors two vertices have in common, but not how often they have similar neighbors. We can also think of an anonymizing array as a set of $\binom{k}{t}$ hypergraphs each having copies of the same N vertices and each graph having up to v^t edges. To ask how similar the neighborhood of a vertex u is across all of the graphs, a similarity score for neighborhoods is required. This might require counting the number of vertices in common and making $\binom{\binom{k}{t}}{2}$ comparisons of the neighborhoods between graphs for each of the N vertices. Defining the similarity score still presents difficulty, as the sets $\{u_i, u_j\} \bigcap \{u_i, u_j\}$ and $\{u_i, u_j, u_l, u_m\} \bigcap \{u_i, u_j, u_p, u_q, u_r, u_s\}$ have the same score, 2, under similarity by intersection. For our purposes, the first two neighborhoods are much more similar than the last two as every vertex that appears in one neighborhood appears in the other. No well-known graph measure accurately describes the metric required here.

Diversity is a metric for covering arrays [46] defined as

$$\frac{\text{number of distinct } t\text{-way interactions covered}}{\text{number of opportunities to cover } t\text{-way interactions}}.$$

To be a covering array without constraints, the number of t-way interactions covered must be $\binom{k}{t}$. Diversity is a measure of how efficiently interactions are covered. An array that covers all t-way interactions in fewer rows has a higher diversity score. Then the number of times a particular t-way interaction is covered is ideally close to one.

For anonymizing arrays, the number of times a credential appears is ideally close to r. If some credentials appear many more times than r, it is possible that the anonymity guarantee can be increased if some of the repetitions of those credentials were replaced with less frequently appearing credentials. Superficially, it seems that diversity can be extended to anonymizing arrays as

$$\frac{\text{number of distinct } t\text{-sized credentials covered at least } r \text{ times}}{\text{number of opportunities to cover } t\text{-sized credentials}}.$$

Dependence between rows does not matter for covering arrays as these are tests in a test suite that are executed independently, but dependence between access profiles does matter for anonymizing arrays. This metric does not capture the preference for two or more access profiles to not contain many of the same credentials.

Call the *neighborhood* of a credential the access profiles that share that credential; this is equivalent to the set of vertices connected by a hyperedge, maintaining multi-edges as unique edges. Computing the neighborhood size for each credential measures how inclusive or exclusive the credential is, but does not tell us anything about which access profiles appear together in neighborhoods. Computing the total number of neighbors for an access profile, all neighbors of the vertex, says something about how many other access profiles it shares a credential with, but does not detect when a vertex has many small neighborhoods of the same neighbors and one large neighborhood shared with everyone versus many diverse neighborhoods. Computing the number of times a pair of access profiles appear together is also useful, but again ignores the effect of large versus small neighborhoods. To reiterate, a measure that includes the number of times access profiles are neighbors and the size of the neighborhoods is needed, and we propose a solution in the next section.

4.3.2 Homogeneity Definitions

Local homogeneity describes how often an access profile appears in small groups of similar access profiles. *Global homogeneity* describes how homogeneous the access profiles of an anonymizing array are on average. The *neighborhood* of a credential is the set of access profiles that possess the credential. The *closeness* of a pair of access profiles is a sum of their *weight* over all credentials, and the weight of a pair of access profiles on a credential is inversely proportional to the size of the neighborhood of the credential if the access profiles are in the neighborhood. In other words, two access profiles that appear together frequently in small neighborhoods are the closest, those that appear together frequently in large neighborhoods or less frequently in small neighborhoods are moderately close, and those that never appear together are the least close. Local homogeneity measures how alike an access profile is to its neighbors as the average of its non-zero closeness scores. Homogeneity is the inverse of what we might have called “diversity.” We are interested in having low global homogeneity for the anonymizing array on average, but keeping the maximum local homogeneity below an acceptable value is important for privacy preservation for individual subjects.

Formally, let $\mathcal{U}$ be a set of N access profiles and let $\mathcal{C}$ be the set of all credentials. Define the neighborhood of a credential $c \in \mathcal{C}$ as $\rho(c) = \{u_i : u_i \text{ possesses } c, u_i \in \mathcal{U}\}$. Define the weight of access profiles u_i and u_j on credential c as

$$weight(u_i, u_j, c) = \begin{cases} \frac{1}{|\rho(c)|} \iff \{u_i, u_j\} \subseteq \rho(c) \\ 0 \text{ otherwise} \end{cases}$$

and

$$closeness(u_i, u_j) = \sum_{c \in \mathcal{C}} weight(u_i, u_j, c).$$

The neighbors of an access profile u_i are the access profiles that are in a neighborhood of u_i's credentials, so define

$$neighbors(u_i) = \left\{ \bigcup_{c \in \mathcal{C}} \rho(c) : u_i \in \rho(c) \right\}.$$

Then define the local homogeneity metric as

$$homogeneity(u_i) = \frac{1}{|neighbors(u_i)|} \sum_{u_j \in \mathcal{U}, u_j \neq u_i} closeness(u_i, u_j)$$

and global homogeneity metric as the average local homogeneity,

$$globalHomogeneity = \frac{1}{N} \sum_{u_i \in \mathcal{U}} homogeneity(u_i).$$

Considering just the sum of closeness scores without averaging over neighbors can be misleading. A high homogeneity access profile may have a few high closeness scores and many zeros, while a low homogeneity access profile may have many low closeness scores, with the result that the closeness scores of the two access profiles may be indistinguishable. By taking the average over the number of neighbors, the average closeness score of each access profile is retrieved and can be compared.

An alternative metric to closeness, $closeness'$, is the count of shared neighborhoods divided by the average neighborhood size, but it is less descriptive. Consider $closeness(u_i, u_j) = \frac{1}{2} + \frac{1}{6} = \frac{4}{6}$. The alternative metric produces $closeness'(u_i, u_j) = \frac{2}{4} = \frac{3}{6}$. In the latter, all shared neighborhoods weigh equally, whereas in the former, the size of the neighborhood determines the weight of that neighborhood. The better metric gives a larger weight when access profiles appear together in small neighborhoods.

Division by $|neighbors(u_i)|$ requires every access profile to have at least one neighbor. This is not the same as requiring $r > 1$ which indicates that every credential has a neighborhood size of at least 2. Computation of homogeneity scores may still be useful as an interim step in an algorithm when the ideal anonymity guarantee has not yet been met. An appropriately large value for $homogeneity(u_i)$ when $|neighbors(u_i)| = 0$ needs to be determined. Choosing ∞ obscures $globalHomogeneity$ and prevents comparison of the array on this metric. No score with a valid number of neighbors can exceed $\binom{k}{t}$ so this is a reasonable value.

The intermediary metric, $closeness(u_i, u_j)$ is useful for finer granularity of analysis. Specifically, an access profile with the highest homogeneity possible has $N - r$ access profiles u_j such that $closeness(u_i, u_j) = 0$ and $r - 1$ access profiles u_ℓ with $closeness(u_i, u_\ell) = \frac{\binom{k}{t}}{r}$. The histogram of closeness scores for a high homogeneity access profile is bimodal. Conversely, a low homogeneity access profile has a histogram of closeness scores with a mode around $\frac{\binom{k}{t}}{N}$. It is likely the case that the variance of high homogeneity is higher than the variance of low homogeneity. This level of granularity may not be necessary, but retaining closeness scores might be useful when attributes can be resampled to break high homogeneity by identifying specific access profiles that are too similar. We use this idea in § 4.3.6.

This level of granularity may not be required, and in this case, computation of the closeness scores can be bypassed by noting that for every credential c of u_i with neighborhood $|\rho(c)|$, $|\rho(c)|-1$ neighbors contribute $\frac{1}{|\rho(c)|}$ to u_i's homogeneity score. This fact explains why it is important to not include $closeness(u_i,u_i)$, as doing so reduces the homogeneity metric to $\frac{\binom{k}{t}}{|neighbors(u_i)|}$ for all u_i. This is essentially just a measure of the total number of neighbors for each u_i since $\binom{k}{t}$ is fixed for all. A measure of the number of neighbors cannot indicate whether u_i shares every neighborhood with the same set of neighbors or if it shares different neighborhoods with subsets of its neighbors. As an example, suppose $\binom{k}{t}=4$ and suppose access profile u_i shares all credentials with 8 other access profiles. Then $homogeneity(u_i)=\frac{4(8)\frac{1}{9}}{8}=\frac{4}{9}$. Suppose another access profile u_j shares each credential with two other access profiles for a total of 8 neighbors. Then $homogeneity(u_j)=\frac{4(2)\frac{1}{3}}{8}=\frac{1}{3}$. As desired, u_i has a higher homogeneity score as it always appears with the same profiles while u_j has a slightly lower homogeneity score. If $closeness(u_i,u_i)$ is included, both scores are $\frac{4}{8}$.

4.3.3 Bounds on Homogeneity

An access profile with the largest homogeneity score possible always appears with the same neighbors in the smallest neighborhoods allowed. Let u_i be such an access profile appearing with the same $r-1$ neighbors for each of its credentials. Then $weight(u_i,u_j,c)=\frac{1}{r}$ for the $\binom{k}{t}$ credentials of u_i, and $closeness(u_i,u_j)=\binom{k}{t}\frac{1}{r}$ for $r-1$ access profiles u_j. Then

$$homogeneity(u_i)\le\frac{(r-1)\binom{k}{t}\frac{1}{r}}{r-1}=\frac{\binom{k}{t}}{r}$$

as the average closeness score of u_i with each of its $r-1$ neighbors, as expected.

A lower bound on the global homogeneity can be computed for the restricted case where the number of attribute levels is constant, all credentials appear, and $N\le v^k$. The array with the lowest homogeneity in this case has the largest possible neighborhood for every credential and an access profile shares each of its credentials with a different group of neighbors. If every credential has the largest neighborhood possible, then $\forall c \ |\rho(c)| = r = \frac{N}{v^t}$. Each of u_i's credentials has $\frac{N}{v^t} - 1$ neighbors contributing $\frac{v^t}{N}$, so the sum of u_i's closeness scores is $\binom{k}{t}(\frac{N}{v^t} - 1)\frac{v^t}{N}$. For u_i to always appear in neighborhoods with different neighbors, $u_i \in \rho(c_a) \wedge u_i \in \rho(c_b) \implies \rho(c_a) \bigcup \rho(c_b) = \{u_i\}$. Then u_i has $\binom{k}{t}(\frac{N}{v^t} - 1)$ neighbors. Then the lower bound is

$$homogeneity(u_i) \geq \frac{\binom{k}{t}(\frac{N}{v^t} - 1)\frac{v^t}{N}}{\binom{k}{t}(\frac{N}{v^t} - 1)} = \frac{v^t}{N}$$

and this is true for every access profile, so it is also the lower bound on the global homogeneity score. This explains why the global homogeneity score tends to decrease as N increases. If $N \geq v^k$, there must be two identical access profiles and then it cannot be the case that a pair of credentials intersects in only one access profile. The restrictions for this bound limit its usefulness.

The expected homogeneity score when attributes are assigned uniformly at random and there are no constraints can also be computed. The probability of a particular credential being placed in a t-set of columns is $\frac{1}{v^t}$ for each access profile, so the expected neighborhood size of a credential is $|\rho(c)| = \frac{N}{v^t}$. Each of u_i's credentials c have $|\rho(c)| - 1$ neighbors that contribute weight $\frac{1}{|\rho(c)|}$, so the expectation of the sum of closeness score for u_i is $\binom{k}{t}\frac{|\rho(c)|-1}{|\rho(c)|} = \binom{k}{t}\frac{\frac{N}{v^t}-1}{\frac{N}{v^t}}$.

To compute the expected number of neighbors, when chosen uniformly at random, an access profile u_j is a neighbor of u_i if and only if u_j has the same symbols as u_i in at least t columns. The probability that u_j has the same symbol as u_i in exactly one column with all others different is $\binom{k}{1}(\frac{1}{v})(\frac{v-1}{v})^{(k-1)} = \binom{k}{1}\frac{(v-1)^{(k-1)}}{v^k}$. The probability that u_j has the same symbols as u_i in no more than $t-1$ columns is

$$P = \sum_{a=0}^{(t-1)} \binom{k}{a} \frac{(v-1)^{(k-a)}}{v^k}.$$

Then the probability that u_j is a neighbor of u_i is $1-P$. The probability that another row u_ℓ is a neighbor of u_i is independent of the probability for u_j, so the expected number of neighbors is

$$|neighbors(u_i)| = (N-1)\left(1 - \sum_{a=0}^{(t-1)} \binom{k}{a} \frac{(v-1)^{(k-a)}}{v^k}\right).$$

Then the expected local homogeneity score is

$$E[homogeneity(u_i)] = \frac{\binom{k}{t}\frac{\frac{N}{v^t}-1}{\frac{N}{v^t}}}{(N-1)\left(1 - \sum_{a=0}^{(t-1)} \binom{k}{a} \frac{(v-1)^{(k-a)}}{v^k}\right)}.$$

When the number of levels is not constant, the same computation is done by enumerating the probabilities of matching the symbols in a set of a columns with all other column symbols different. This method for computing the expectation does not appear to be useful when the array contains hard constraints, as a constraint involving t symbols changes the probabilities of subsets of those symbols.

Parameter values relative to other parameters affect the homogeneity. As mentioned before, the global homogeneity tends to decrease as N increases. As credentials grow in size – as t approaches k for fixed N – the number of neighbors expected for an access profile decreases and therefore the expected homogeneity score increases. That is, the closer a credential becomes to being a full access profile, the less other access profiles can be used to obscure the identity of the subject. If an access profile has a large number of neighbors as t approaches k, this suggests duplication. Specifically, if an access profile has a neighbor for $t = k$, then there must be two identical access profiles. If $r > \frac{N}{v^t}$, then some credential does not appear, increasing the number of times the remaining credentials for that set of t attributes appear. When $r = N$, every row is a duplicate. Therefore, the homogeneity score makes the most sense when r is chosen reasonably and is useful to compare two anonymizing arrays with the same parameters.

4.3.4 Homogeneity Computation

Algorithm 8 computes the local homogeneity of an anonymizing array, **A**.

Algorithm 8: Compute Homogeneity

```
input  : $\mathbf{A}$ an $N \times k$ array, $v$, $t$
output: $Homogeneity$ a $1 \times N$ array
begin
    for each of the $\binom{k}{t}$ subsets of $t$ columns do
        Create a list $\rho(c)$ for each of the $v^t$ credentials, $c$
        for each $u_i \in \rho(c)$ do
            Add $\frac{|\rho(c)|-1}{|\rho(c)|}$ to $Homogeneity[u_i]$
            for each $u_j \neq u_i \in \rho(c)$ do
                Set $neighbors[u_i][u_j] = true$
    Count the true values of $neighbors[u_i]$ as $numNeighbors[u_i]$
    Set $Homogeneity[u_i] = \frac{Homogeneity[u_i]}{numNeighbors[u_i]}$
```

The algorithm employs the following storage:

- v^t lists with average size $\frac{N}{v^t}$ for a total space of N where the space can be reused for each of the $\binom{k}{t}$ iterations;
- a $1 \times N$ array of doubles to store the homogeneity scores;
- a $N \times N$ array of booleans to store the neighbor status.

The running time is as follows:

- computing the lists for each set of t columns requires one scan through the $N \times t$ subarray, $\mathcal{O}(N\binom{k}{t})$;
- for each of the v^t lists, update the value in homogeneity, $\mathcal{O}(N)$, and update the neighbors boolean array, $\mathcal{O}(N^2)$;

- compute the count of neighbors for each of the N access profiles, $\mathcal{O}(N^2)$.

The entire algorithm is $\mathcal{O}(N^2 + N\binom{k}{t})$. It is preferred to eliminate the need to count the neighbors of u_i, but as explained, it is required for our metric, and there does not appear to be a way to count the neighbors of an access profile without comparing it to all other access profiles.

4.3.5 Homogeneity Examples

The three anonymizing arrays in the first example (Figures 4.4, 4.6, and 4.8) have the same parameters N, r, t, k, v. They do not have the same credentials appearing. The minimum, global, and maximum homogeneity scores for each array are in Table 4.2. The lower bound, expectation, and upper bound are 0.5, 0.75, and 1.5, respectively. The first, built from a 2^3 full factorial design, is as diverse as possible with a different access profile in every row (Figure 4.4). The second is built from a 2^{3-1} fractional factorial with two replicates where the last four rows are copies of the first four (Figure 4.6). The third has two replicates of one access profile and six replicates of another (Figure 4.8).

The first two arrays have a uniform local homogeneity score for every access profile. The small group in the third array has a much higher homogeneity score than the large group. Even though the large group contains identical access profiles, it is not considered to be highly homogeneous, relative to the small group. While the large group can be tracked, it is anticipated that the impact of tracking to individual subjects is smaller due to the obscurity provided by the size of the group.

Figures 4.5, 4.7, and 4.9 show the multi-hypergraph representations of these arrays. Access profiles are vertices labeled by the corresponding row index in the array, and an edge surrounds a set of vertices if the set of access profiles share a credential. Each vertex has degree three. The number of vertices contained in an edge varies.

	a_1	a_2	a_3
1	0	0	0
2	1	0	0
3	0	1	0
4	1	1	0
5	0	0	1
6	1	0	1
7	0	1	1
8	1	1	1

Figure 4.4: Low homogeneity $\mathsf{AA}(8; 2, 2, 3, 2)$ from 2^3 full factorial

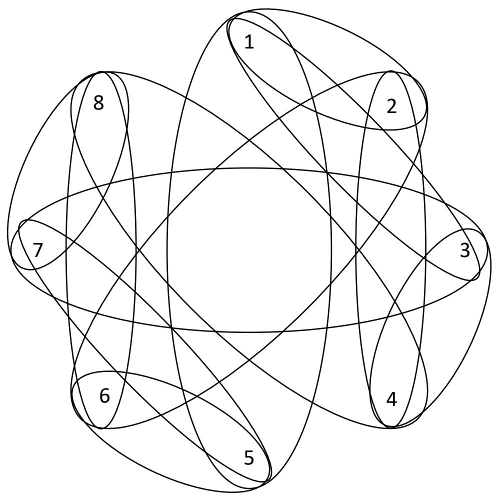


Figure 4.5: Multi-hypergraph representation of the low homogeneity array

This representation demonstrates the clustering behavior that the homogeneity metrics are designed to measure.

Now consider the low homogeneity array from Figure 4.4 and two anonymizing arrays that have the same credentials appearing and the same parameters r, t, k, v, but six more rows. The array in Figure 4.10 is constructed by repeating the first six rows of Figure 4.4. The additional rows provide more neighbors, so each access profile appears in larger groups, even if they are not uniformly distributed. This reduces the

	a_1	a_2	a_3
1	1	0	0
2	0	0	1
3	1	1	0
4	0	1	1
5	1	0	0
6	0	0	1
7	1	1	0
8	0	1	1

Figure 4.6: Medium homogeneity $\mathsf{AA}(8; 2, 2, 3, 2)$ from 2^{3-1} fractional factorial

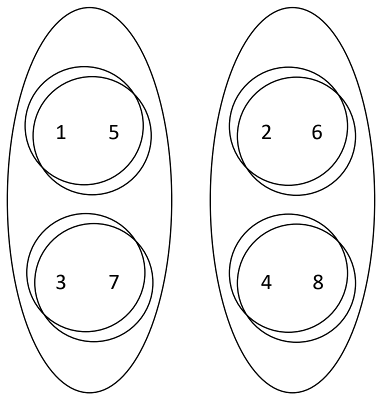


Figure 4.7: Multi-hypergraph representation of the medium homogeneity array

global homogeneity score overall and decreases the local homogeneity scores.

The third anonymizing array, given in Figure 4.11, is created by repeating the last row of the original array. This provides additional neighbors to some access profiles, reducing their homogeneity scores, but does not affect others. While the global homogeneity score is lower, the local homogeneity scores in this array have a larger range, and some have no improvement over the original array. This suggests that global homogeneity alone is less useful as a metric than when minimum local

	a_1	a_2	a_3
1	0	0	0
2	0	0	0
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	1	1	1
8	1	1	1

Figure 4.8: High homogeneity $\mathsf{AA}(8; 2, 2, 3, 2)$ with two unique rows and many replicates

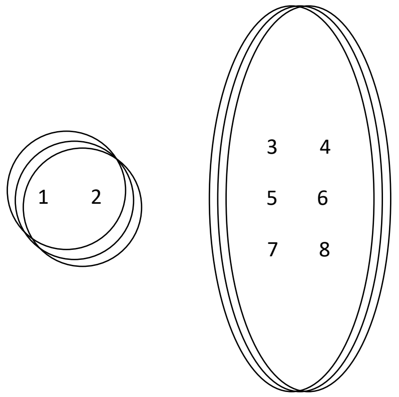


Figure 4.9: Multi-hypergraph representation of the high homogeneity array

Table 4.2: Homogeneity scores for the low, medium, and high homogeneity arrays

Array	min	global	max
Low homogeneity	0.500	0.500	0.500
Medium homogeneity	0.583	0.583	0.583
High homogeneity	0.500	0.750	1.500

Table 4.3: Low homogeneity array homogeneity scores

row	$\|neighbors\|$	local homogeneity
1	3	0.500
2	3	0.500
3	3	0.500
4	3	0.500
5	3	0.500
6	3	0.500
7	3	0.500
8	3	0.500

homogeneity and maximum local homogeneity are used alongside the average (global) homogeneity.

The number of neighbors and local homogeneity scores of each access profile are given for the arrays in Tables 4.3, 4.4, and 4.5, and the minimum, average, and maximum local homogeneity scores for each array are given in Table 4.6.

4.3.6 Homogeneity Post-optimization (HP) Algorithm

An anonymizing array instance may not be the least homogeneous instance for a given set of parameters. The homogeneity of an anonymizing array is described by the homogeneity scores of all access profiles and, with a lower level of granularity, the global homogeneity score, possibly supplemented by the minimum and maximum local homogeneity scores. In this section, we develop a post-optimization strategy that takes an anonymizing array and attempts to improve its homogeneity.

Necessity analysis works for row reduction when homogeneity is ignored because

	a_1	a_2	a_3
1	0	0	0
2	1	0	0
3	0	1	0
4	1	1	0
5	0	0	1
6	1	0	1
7	0	1	1
8	1	1	1
9	0	0	0
10	1	0	0
11	0	1	0
12	1	1	0
13	0	0	1
14	1	0	1

Figure 4.10: $\mathsf{AA}(14; 2, 2, 3, 2)$ from a 2^3 full factorial with the last six rows replicated once

Table 4.4: Homogeneity scores for the $\mathsf{AA}(14; 2, 2, 3, 2)$ from a 2^3 full factorial with the last six rows replicated once

row	$\lvert neighbors \rvert$	local homogeneity
1	7	0.321
2	7	0.321
3	6	0.361
4	6	0.361
5	6	0.361
6	6	0.361
7	5	0.367
8	5	0.367
9	7	0.321
10	7	0.321
11	6	0.361
12	6	0.361
13	6	0.361
14	6	0.361

	a_1	a_2	a_3
1	0	0	0
2	1	0	0
3	0	1	0
4	1	1	0
5	0	0	1
6	1	0	1
7	0	1	1
8	1	1	1
9	1	1	1
10	1	1	1
11	1	1	1
12	1	1	1
13	1	1	1
14	1	1	1

Figure 4.11: $\mathsf{AA}(14; 2, 2, 3, 2)$ from a 2^3 full factorial with the last row replicated six times

Table 4.5: Homogeneity scores for the $\mathsf{AA}(14; 2, 2, 3, 2)$ from a 2^3 full factorial with the last row replicated six times

row	$\|neighbors\|$	local homogeneity
1	3	0.500
2	3	0.500
3	3	0.500
4	9	0.208
5	3	0.500
6	9	0.208
7	9	0.208
8	9	0.292
9	9	0.292
10	9	0.292
11	9	0.292
12	9	0.292
13	9	0.292
14	9	0.292

Table 4.6: Comparison of homogeneity scores for adding rows to the low homogeneity array from a 2^3 full factorial

Array	min	global	max
Low homogeneity	0.500	0.500	0.500
Replication of six rows	0.321	0.351	0.367
Replication of one row	0.208	0.333	0.500

credentials are only required to appear r times, thus it operates in a vertical manner on the array. That is, begin checking at the top of an $N \times t$ sub-array and mark the positions of a row as necessary for any credential appearing in the row that has not yet appeared r times. Homogeneity has simultaneous vertical and horizontal implications. That is, it is a measure of in how many credentials (horizontal) access profiles appear together and how many access profiles (vertical) are in the neighborhoods of the credentials. It seems unlikely that any strictly top-down or left-right approach is useful.

High homogeneity occurs when some access profile, u, either shares many of its credentials with the same access profiles or when the neighborhood sizes of its credentials are small, or a combination of the two. In the multi-hypergraph representation of an anonymizing array, u has many edges linking it to its neighbors and the edge degree is small. A post-optimization strategy is to remove u from an edge that it shares with an access profile v such that $closeness(u, v)$ is large and add an edge with an access profile w such that $closeness(u, w)$ is small by changing some of u's attributes. A purely random approach simply resamples all of u's attributes, but this may cause the array to no longer meet the anonymity guarantee.

A targeted approach is to change the attributes so that some of the credentials are preserved in the modified array, even if they appear in new rows. This is accomplished by *crossover*, or swapping credentials between two access profiles, an idea inspired by genetic algorithms. An algorithm for this approach works as follows. Identify a high homogeneity access profile, u. Scan the closeness scores of u and identify v, the access profile with the highest closeness score. Scan the weights for each credential and select the credential c with the largest $weight(u, v, c)$. For this credential, identify the access profile w for which $weight(u, w, c)$ is smallest, which is zero for any access profile w that does not have that credential. Swap the attributes of u and w for the columns of credential c.

This approach has two issues. First, it requires the weight array to be stored whereas closeness can be computed as sums without the intermediary weight array. Second, the view at the granularity level of weight does not inform how close u and w are on other credentials. Suppose u and w are identical in all columns except some columns of c. Then while u is close to v, w is also close to v. Swapping the attributes of c in this case is equivalent to swapping the rows of u and w so that u itself may be less close to v, but the overall homogeneity of the array has not changed. The key intuition is that crossover should be used to "decouple" u from a clique-like group on one credential and create a link between that group and an access profile outside the group while forming a link between u and the group of w.

A simplified example is in Figure 4.12. Suppose $c_{x,i}$ is a credential for the attributes for the set of columns at rank x having value i. Access profiles u and v share two credentials and are tightly coupled as are w and z. By swapping the values on credential c_x between u and w, u now shares a credential with z and w shares a credential with v, decoupling the groups.

Suppose then that we identify u and w such that $homogeneity(u)$ is highest and $closeness(u, w)$ is lowest. Scanning the weights gives information about shared credentials so we could choose to swap any credentials c where $weight(u, w, c,) = 0$. Too many swaps results in swapping the entire row, and as u and w are chosen to have

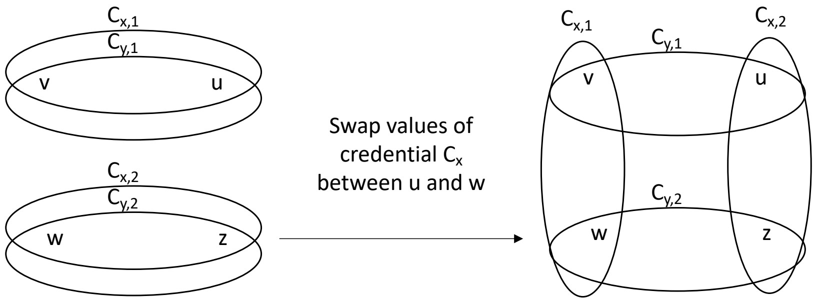


Figure 4.12: Crossover of credentials between access profiles u and w to decrease homogeneity

the smallest *closeness* score, they may have no credentials in common. Swapping a single credential changes up to $\binom{k}{t} - \binom{k-t}{t}$ other credentials, so it seems unlikely that there is an approach that makes the best decision without considering all possibilities. Instead, we take an approach that is the middle path between random row resampling and computationally intensive search.

The strategy is to generate a set of child arrays and choose the most fit child to become the parent for the next generation. The fitness of an array is its global homogeneity. For each child of a generation, an access profile u with the highest homogeneity score is identified and a set S of the s user profiles with the lowest $closeness(u, w)$ scores are selected with ties broken randomly. A $blocksize \leq t$ is deterministically chosen that decreases relative to time, $\frac{generation}{maxGenerations}$, and a starting attribute $1 \leq start \leq k$ is randomly selected. For each of the $\lfloor \frac{k}{blocksize} \rfloor$ blocks of attributes starting at $start$, the decision to swap is made with probability that decreases in proportion to time and $w \in S$ is selected with probability $\frac{1}{s}$. In this implementation, $s = r$ if there are at least r padding rows; otherwise s is set to the number of padding rows minus one. Only padding rows are affected by crossover and mutation; user access profiles are considered immutable. After crossover is completed, the child is checked to ensure it still meets the anonymity guarantee and no hard constraints appear. If not, it's deleted. Initially the parent is labeled $mostFit$, and a child becomes $mostFit$ if $globalHomogeneity(child) \leq globalHomogeneity(mostFit)$. Once all children have been generated and compared, $mostFit$ becomes the parent for the next generation. When the parent and a child have the same $globalHomogeneity$, the child is preferred to avoid multiple generations without a change to the array. When no child is more fit than the parent, the next generation begins with the last parent.

As mentioned, swapping one credential changes up to $\binom{k}{t} - \binom{k-t}{t}$ other credentials in the same access profile. If one of the affected credentials appears very few times in the array, it might be the case that swapping reduces the appearance of a credential below r, and it may be the case that a solution cannot be found in the number of random children allowed in a generation. In this case, the parent is retained for the next generation. This triggers a mutation round. The intuition behind mutation is to allow additional appearances of the credential that is eliminated by resampling to occur elsewhere in the array so that the array is still (r, t)-anonymous. In a mutation round, an access profile is selected for mutation and each of its symbols are randomly resampled with probability relative to the number of generations without progress. The number of rows to mutate and how often to mutate is tunable. In this implementation, one row is chosen for mutation and the $mutationRate$ increases by 0.1 for every generation without progress, stopping when $mutationRate = 1.0$. Mutation continues until the array again reaches the anonymity guarantee.

It seems likely that the number of rows to mutate should be proportional to the size of the problem with larger arrays allowed more mutated rows. How to choose the row to mutate is also an open problem. It seems that the rows that are being affected by crossover should be avoided as successful mutation increases the appearance elsewhere of a credential that has been eliminated from those rows. In this implementation, a row that has a low closeness score with the row with the largest homogeneity score is chosen. A randomly selected row might be more effective at avoiding getting stuck or there might even be an intelligent choice. Mutating after crossover in the event that the child does not meet the anonymity guarantee is an option. It is not chosen at this time for the reason that this increases the chances of getting stuck in a loop. That is, as implemented, if a child ceases to meet the anonymity guarantee due to a mutation, the row that has been mutated is the only row modified at this time and so resampling occurs on this row until a solution is found. As implemented, crossover may cause coverage lower than r and the worst case is that the child is not selected for the next generation. If mutation follows crossover, $s+1$ rows have been modified. It might be the case that mutating one row can not solve the problem and the algorithm may loop forever. This supports the claim that a "good" way to choose mutation is still needed.

A number of stopping conditions are possible. The algorithm can be designed to stop after a given number of iterations, *maxGenerations*, which is used here. Alternatively, it can be stopped early if a number of generations have passed without improvement, if it reaches some given value such as the expectation, or when the minimum local homogeneity score is equal to the maximum local homogeneity score. A high-level view of the Homogeneity Post-optimization (HP) algorithm is given in Algorithm 9.

4.4 Results

4.4.1 Comparison of MTCR and CEHS Algorithms

The most common use expected for an anonymizing array is the case where a set of access profiles is provided that does not yet meet the desired anonymity guarantee and an anonymizing array is created with respect to the set. In this case, the CEHS

Algorithm 9: Homogeneity Post-optimization (HP)

```
input  : A an $N \times k$ array, $r, t, levels$, $maxGenerations$, $numChildren$, $s$
output: A
begin
    Set $parent = \mathbf{A}$
    Compute $Homogeneity$ for $parent$
    $timeSinceProgress = 0$
    repeat
        $mostFit = parent$
        for $0 \le i < numChildren$ do
            Create $child$ a copy of $\mathbf{A}$
            Identify $u = \max_i^N(homogeneity(i))$
            if $timeSinceProgress > 0$ then
                Select a user profile $w$ with the $s + 1$ smallest $closness(u, w)$
                Set $mutationRate = 0.1 timeSinceProgress$
                repeat
                    Resample each symbol in $w$ with probability $mutationRate$
                until $child$ has anonymity guarantee $r$
            Identify $S$, a set of user profiles with the $s$ smallest scores
             $closeness(u, w), w \in S$
            Randomly select $0 \le start < k$
            Set $blockSize = \lceil (1 - \frac{generation}{maxGenerations}) t \rceil$
            for each of $\lfloor \frac{k}{blocksize} \rfloor$ columns starting at position start do
                Randomly select $w$ with $\frac{1}{s}$ probability
                Swap $w$'s symbols in those columns with $u$'s
            Compute $r$ and $Homogeneity$ for $child$
            if $child$ has anonymity guarantee $r$ AND
             $globalHomogeneity(child) \le globalHomogeneity(mostFit)$ then
                Set $mostFit = child$
        if $mostFit$ is $parent$ then
            Increment $timeSinceProgress$
        else
            Set $timeSinceProgress = 0$
        Set $parent = mostFit$
    until $generation < maxGenerations$
    Set $\mathbf{A} = parent$
    Return A
```

	c_1	c_2	c_3	c_4
1	0	0	0	0
2	0	0	1	1
3	1	0	0	0
4	1	0	1	1
5	2	1	0	1
6	2	1	1	0

Figure 4.13: **A** in numerical levels, $\mathsf{AA}(6; 1, 2, 4, (3, 2^3))$

Table 4.7: Hard and soft constraints in numerical levels for **A**

Hard constraints	Soft constraints
$\{(c_0, 0), (c_1, 1)\}$	$\{(c_0, 1), (c_1, 1)\}$
$\{(c_0, 2), (c_1, 0)\}$	

and MTCR algorithms are used to append padding access profiles.

The anonymizing array example **A** from section § 4.1.3 represented by a mapping of attribute values to numerical levels is in Figure 4.13. It is $(1, 2)$-anonymous. The $(2, 2)$-anonymizing arrays, **C** and **D**, for the provided set of access profiles constructed by CEHS and MTCR are in Figure 4.14 and Figure 4.15, respectively. The constraints are in Table 4.7. MTCR produces an array with two more rows while CEHS finds one with the minimum number of rows, though randomness in both algorithms produces arrays with varying numbers of rows in different executions.

In tests, MTCR produces anonymizing arrays with the same number of rows as CEHS when $t = 1$ without constraints and the algorithm is restricted to use only the number of rows produced by CEHS. When it is allowed to add more rows than a provided bound, it often adds more than needed. This is likely due to implementation choices regarding when to add rows. The algorithm is run to produce instances of $\mathsf{AA}(r, 1, 10, (5^1 4^2 3^3 2^4))$ when either N is restricted or unlimited with results in Table 4.8.

	c_1	c_2	c_3	c_4
1	0	0	0	0
2	0	0	1	1
3	1	0	0	0
4	1	0	1	1
5	2	1	0	1
6	2	1	1	0
7	1	1	0	1
8	1	1	1	0
9	0	0	1	0
10	2	1	1	1
11	0	0	0	1
12	2	1	0	0

Figure 4.14: **C**, a $(2, 2)$-anonymizing array for **A** built by CEHS, $\mathsf{AA}(12; 2, 2, 4, (3, 2^3))$

	c_1	c_2	c_3	c_4
1	0	0	0	0
2	0	0	1	1
3	1	0	0	0
4	1	0	1	1
5	2	1	0	1
6	2	1	1	0
7	0	0	1	1
8	0	0	0	1
9	1	0	1	1
10	2	1	1	0
11	2	1	0	1
12	1	0	0	0
13	0	0	1	0
14	2	1	0	1

Figure 4.15: **D**, a $(2, 2)$-anonymizing array for **A** built by MTCR, $\mathsf{AA}(14; 1, 2, 4, (3, 2^3))$

Table 4.8: Number of rows produced by MTCR with restricted rows versus MTCR with unlimited rows for $t = 1$

r	Restricted	Unlimited
1	5	9
2	10	13
3	15	17
4	20	25
5	25	29

When $t = 2$ and MTCR is allowed 10^6 iterations, in general it requires more rows than CEHS to find a solution. When $t = 2$ and one hard constraint is introduced, MTCR requires approximately twice as many rows. When the number of hard constraints is increased to two, MTCR does not complete in 10^6 iterations for any fixed number of rows or allowed unlimited rows. When the two constraints are instead soft constraints, MTCR performs in fewer iterations and rows than for one hard constraint. The number of rows produced and iterations required to construct instances of $\mathsf{AA}(r, 2, 10, (5^1 4^2 3^3 2^4))$ by MTCR and CEHS and with the different numbers and types of constraints are given in Table 4.9. An execution failing to complete in the number of iterations allowed is denoted by $N = \infty$.

4.4.2 Comparison to Replicated Mixed-Level Covering Arrays with Constraints

A less frequent scenario for anonymizing array use is when attributes can be assigned arbitrarily to subjects, similar to key distribution. In this case, an anonymizing array may be created "from scratch." Given the relationship to covering arrays, in this section, we compare the performance of constructing anonymizing arrays for $r > 1$ by CEHS against the copy construction of covering arrays in Theorem 3, and we compare constrained and unconstrained arrays.

To create a constrained anonymizing array, CEHS is executed for $1 \le r \le 5$, $1 \le t \le 4$, and the set of hard constraints $\mathcal{H}_t$ (Table 4.10) to construct an

Table 4.9: Number of rows produced by MTCR versus CEHS for $t = 2$

r	Algorithm	Number constraints	Constraint type	Rows allowed	Iterations	N
1	CEHS	0	none	–	–	24
1	MTCR	0	none	24	10^6	∞
1	MTCR	0	none	36	10^6	∞
1	MTCR	0	none	48	735	48
1	MTCR	0	none	∞	126	59
2	CEHS	0	none	–	–	43
2	MTCR	0	none	43	10^6	∞
2	MTCR	0	none	64	242199	64
2	MTCR	0	none	86	26	86
2	MTCR	0	none	∞	411	100
1	CEHS	1	hard	–	–	26
1	MTCR	1	hard	36	10^6	∞
1	MTCR	1	hard	48	4441	48
1	MTCR	1	hard	∞	1228	64
2	CEHS	1	hard	–	–	41
2	MTCR	1	hard	64	10^6	∞
2	MTCR	1	hard	86	6500	86
2	MTCR	1	hard	∞	10088	93
1	CEHS	2	hard	–	–	26
1	MTCR	2	hard	36	10^6	∞
1	MTCR	2	hard	48	10^6	∞
1	MTCR	2	hard	∞	10^6	∞
2	CEHS	2	hard	–	–	44
2	MTCR	2	hard	64	10^6	∞
2	MTCR	2	hard	86	10^6	∞
2	MTCR	2	hard	∞	10^6	∞
1	CEHS	2	soft	–	–	27
1	MTCR	2	soft	48	247	48
1	MTCR	2	soft	∞	123	57
2	CEHS	2	soft	–	–	43
2	MTCR	2	soft	86	434	86
2	MTCR	2	soft	∞	468	90

Table 4.10: Sets of hard constraints H_t chosen for $2 \le t \le 4$

$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{H}_4$
$\{(a_1,0),(a_2,0)\}$	$\{(a_1,0),(a_4,0),(a_7,0)\}$	$\{(a_1,0),(a_2,0),(a_3,0),(a_4,0)\}$
$\{(a_1,1),(a_3,1)\}$	$\{(a_2,0),(a_5,0),(a_8,0)\}$	$\{(a_5,0),(a_6,0),(a_7,0),(a_8,0)\}$
$\{(a_4,0),(a_7,0\}$	$\{(a_3,0),(a_6,0),(a_9,0)\}$	$\{(a_1,1),(a_2,1),(a_9,1),(a_{10},1)\}$
$\{(a_5,1),(a_8,1)\}$	$\{(a_1,1),(a_2,1),(a_{10},1)\}$	
$\{(a_6,2),(a_9,0)\}$		
$\{(a_1,2),(a_{10},1)\}$		

$\mathsf{AA}(r,t,10,(5^14^23^32^4))$. No constraints are given for $t=1$ as this simply reduces the number of levels by one for that row. The number of rows for this construction are plotted in Figure 4.22 with closed scatterplot markers and labels indicating t and "CEHS(r)."

An anonymizing array with $r=1$ is equivalent to a mixed-level covering array with constraints. Each $\mathsf{AA}(1,t,10,(5^14^23^32^4))$ is a $\mathsf{CA}(t,10,(5^14^23^32^4))$ with hard constraints $\mathcal{H}_t$. To build an anonymizing array with anonymity guarantee r, $2 \le r \le 5$ copies are made of each covering array. The number of rows for this construction are plotted in Figure 4.22 with the open scatterplot markers and labels indicating the value of t and "CEHS(1)$\times r$."

When $t=1$, the number of rows needed is always r times the maximum number of levels, and both constructions produce the same number of rows. For $t>1$, copying the covering array $r>1$ times always produces more rows than constructing the anonymizing array by CEHS.

When parameters N, r, t, k, and v are controlled, CEHS likely produces anonymizing arrays that are less homogeneous than the copy construction method. A challenge in comparing these methods is that an anonymizing array with more rows is less homogeneous than one with fewer rows, in general. Additional rows increase the likelihood that access profiles have larger credential neighborhoods. When $t=1$, the anonymizing arrays produced by both methods have the same number of rows for all values of r

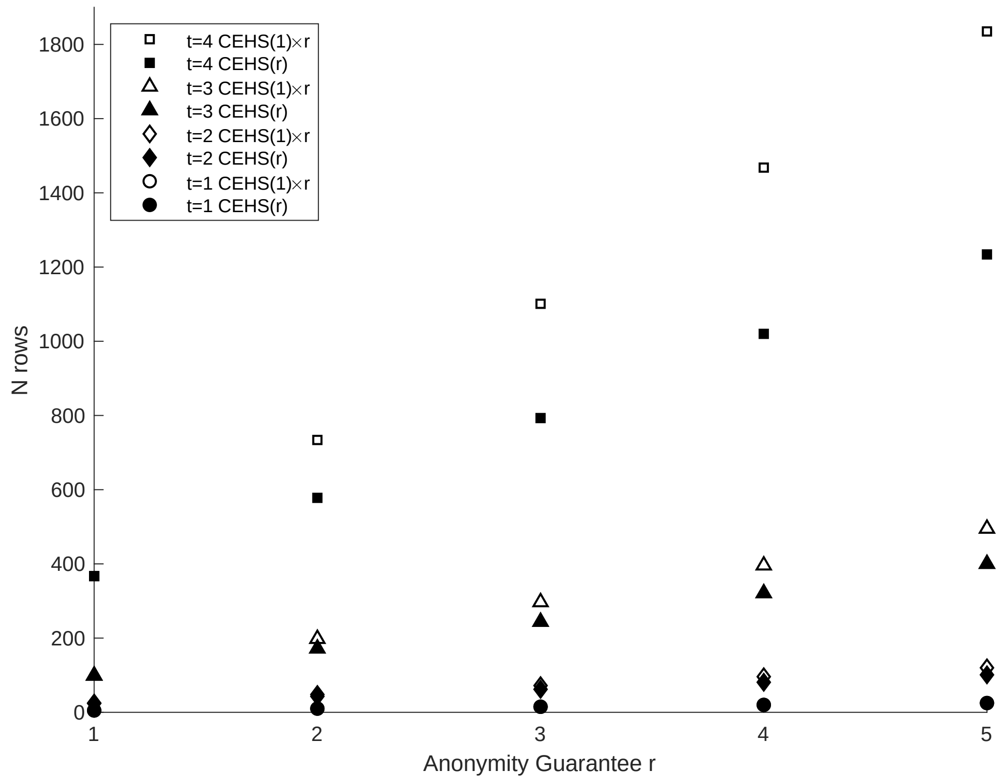


Figure 4.16: Comparison of CEHS versus copy construction of covering arrays to build anonymizing arrays with hard constraints

and so provide a good opportunity for comparison. Figure 4.17 depicts the minimum, average, and maximum local homogeneity as low, middle, and high bars, respectively, for the two series of anonymizing arrays, $\mathsf{AA}(r, 1, 10, (5^1 4^2 3^3 2^4))$ constructed as described. The lines are horizontally offset for readability. The anonymizing arrays created by CEHS for each value of r always have lower homogeneity scores than the arrays created by making r copies of the covering array.

To form a second basis for comparison, we add or delete rows from two $(2, 2)$-anonymizing arrays constructed in this section. We randomly select 5 rows from another of the $(r, 2)$-anonymizing arrays constructed by CEHS and append them to the $\mathsf{AA}(43; 2, 2, 10, (5^1 4^2 3^3 2^4))$. We then compare the homogeneity scores of the re-

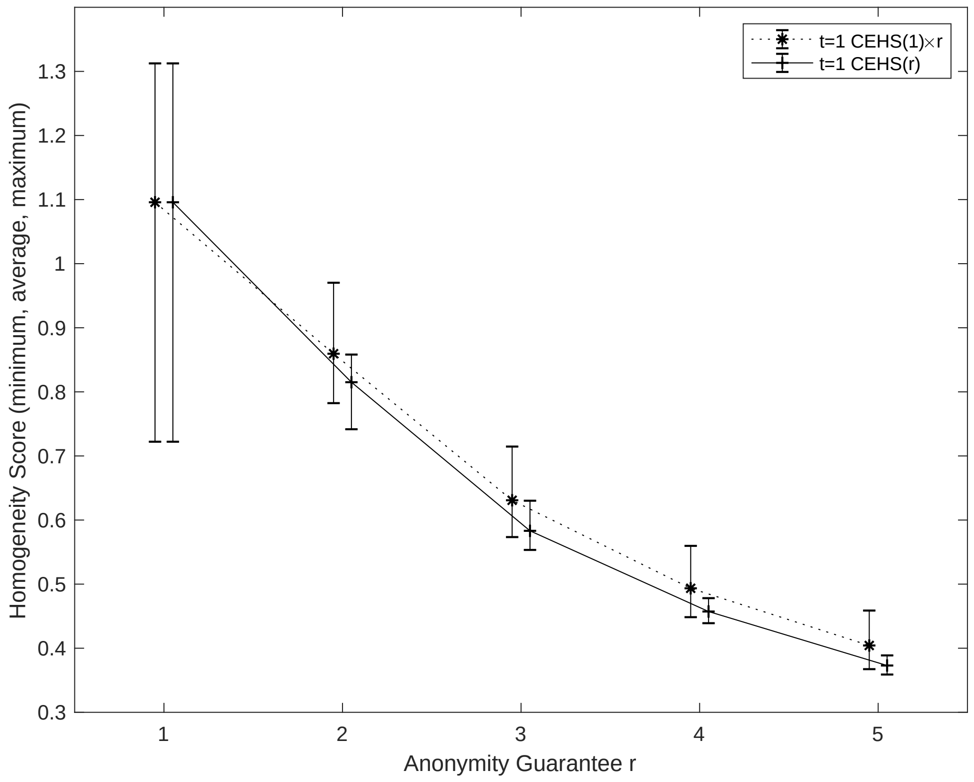


Figure 4.17: Comparison of homogeneity scores for $(r, 1)$-anonymizing arrays constructed by CEHS and copy construction. Top bar is maximum, midpoint is average, and bottom bar is minimum homogeneity score for each array.

sulting array to the $\mathsf{AA}(48; 2, 2, 10, (5^1 4^2 3^3 2^4))$ created by the copy construction. The rows are not constructed randomly to ensure that no hard constraints are introduced into the array. (Note that this method is not without bias due to the pool of rows from which these rows are selected. This method is an ad-hoc comparison in the absence of a standardized homogeneity metric that adequately compares arrays with differing numbers of rows. It is not intended for practical use.) Both have $r = 2$. We also remove the last 5 rows from the copy construction array to compare to the CEHS array. The removal of these rows caused it to drop to anonymity guarantee $r = 1$, so we are no longer comparing two anonymizing arrays with exactly the same pa-

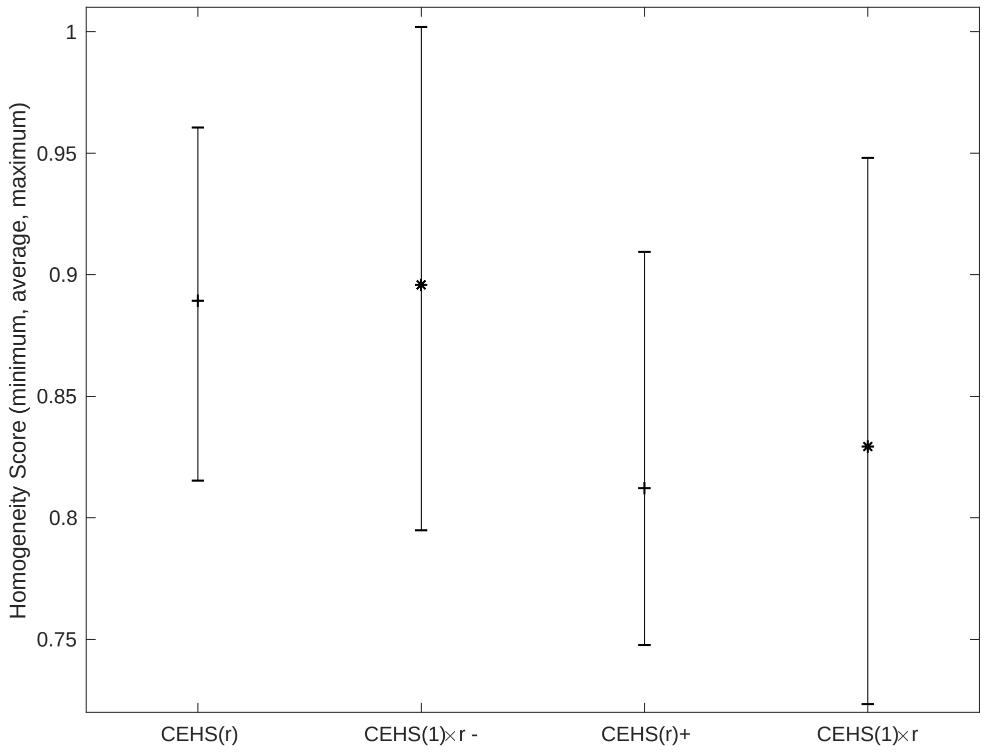


Figure 4.18: Comparison of homogeneity scores for $(2, 2)$-anonymizing arrays constructed by CEHS and copy construction when rows are added or deleted. Top bar is maximum, midpoint is average, and bottom bar is minimum homogeneity score for each array.

rameters. The minimum, average, and maximum local homogeneity scores are shown in Figure 4.18 with the '-/+' notation indicating that rows were removed/added. In both comparisons, the CEHS array had a lower global homogeneity score than the copy construction array. The copy construction array has a larger spread of homogeneity scores, so while the local minimum is lower, the local maximum is higher, suggesting that arrays made by the copy construction provide less anonymity in the worst case for some subjects.

4.4.3 Comparison to Replicated Covering Arrays without Constraints

In Chapter 2, we develop a conditional expectation algorithm to construct SCPHFs which expand to covering arrays by column replacement. In this section, we use one covering array found by that method and apply the copy construction and a permuted copy construction to make anonymizing arrays. We then compare the copy construction methods against CEHS.

A set of arrays, $\mathsf{AA}(245r; r, 3, 10, 5)$, is constructed by making $1 \leq r \leq 10$ vertical copies of a $\mathsf{CA}(245; 3, 10, 5)$. Call this method CAcopy. The initial covering array made by the SCPHF conditional expectation (CE) algorithm has 62 fewer rows, but the CEHS algorithm produces anonymizing arrays with fewer rows than CE for $r \geq 2$ shown in Figure 4.19.

Another method, CAperm, constructs anonymizing arrays with permuted copies. For each copy $i > 1$ and for each column j in the copy, choose a random permutation over the levels of a column, $p_{c_{i,j}} : v \mapsto v$. The resulting permuted copy is still a covering array, so the larger array is (r, t)-anonymous. Consider a row ρ in the original covering array. In the array made by CAcopy, ρ appears (at least) r times, and this forms a cluster of rows sharing the same credentials and therefore neighborhoods. In the array made by CAperm, k independent permutations are applied to the columns of the ρth row in a copy, so the likelihood that this row closely matches ρ is reduced. Therefore, we expect the arrays made by CAperm to exhibit lower homogeneity than direct copies, and this is supported by the results. For all data points, the permuted arrays have lower average and maximum homogeneity scores, and in all but one data point, they have lower minimum homogeneity scores.

The arrays created by CEHS have fewer rows than the arrays found by the copy construction, and here again, the results suggest that increasing N while controlling

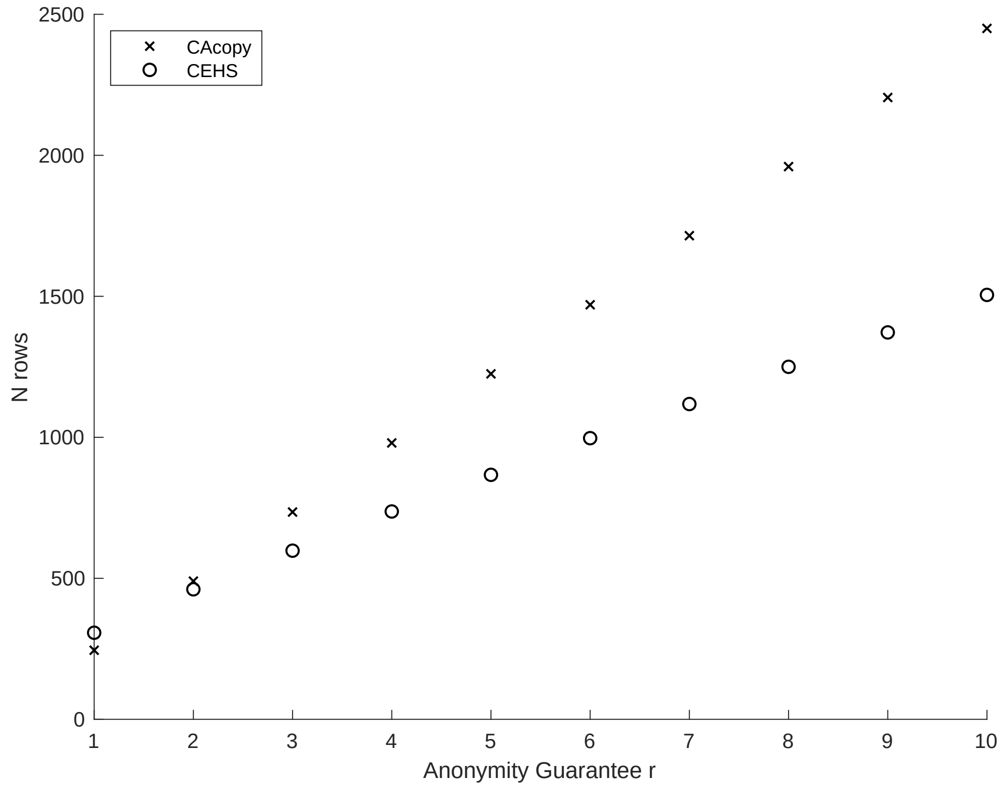


Figure 4.19: Comparison of CEHS versus copy construction of covering arrays to build unconstrained anonymizing arrays

the other parameters results in lower homogeneity, making arrays of different sizes difficult to compare on this metric. The one exception is $r = 2$ when CEHS has both fewer rows and lower homogeneity scores. Figure 4.20 gives the minimum, average, and maximum local homogeneity scores for each constructed array horizontally offset for readability.

Despite the differences in number of rows, we compare the constructions further. As CAperm produces the lower homogeneity scores of the two copy constructions, it is compared against CEHS. As there are no hard constraints to avoid and every credential appears at least r times, appending randomly generated rows will still result in an anonymizing array. Therefore, randomly generated rows are appended to

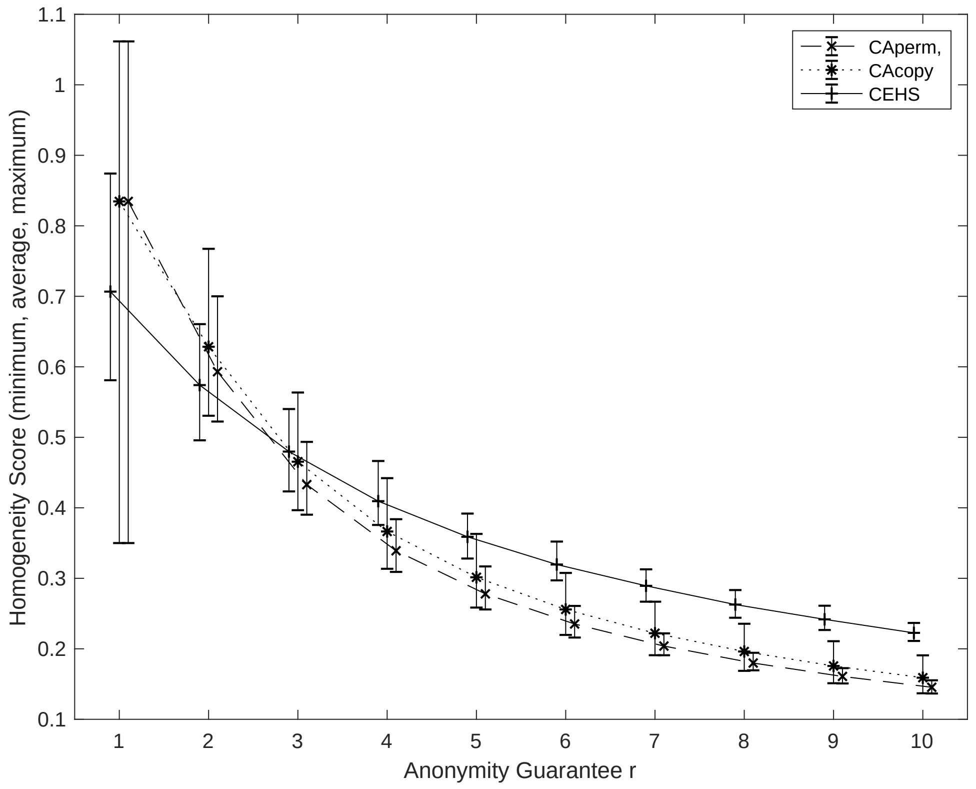


Figure 4.20: Comparison of homogeneity scores for anonymizing arrays constructed by CEHS, CAcopy, and CAperm. Top bar is maximum, midpoint is average, and bottom bar is minimum homogeneity score for each array.

whichever array has fewer rows, producing arrays with the same parameters. (As before, this method is intended for comparison here only in the absence of a standardized homogeneity metric. Additionally, appending a large number of rows could produce an anonymizing array with a larger value for r.) The array with fewer rows is the CEHS array when $r > 1$ and the CAperm array for $r = 1$. After the number of rows has been equalized, in all except one case, CEHS produces arrays with lower homogeneity scores. The one exception is that CAperm produces an array with lower maximum homogeneity for $r = 10$. The homogeneity scores for the equalized arrays are shown in Figure 4.21.

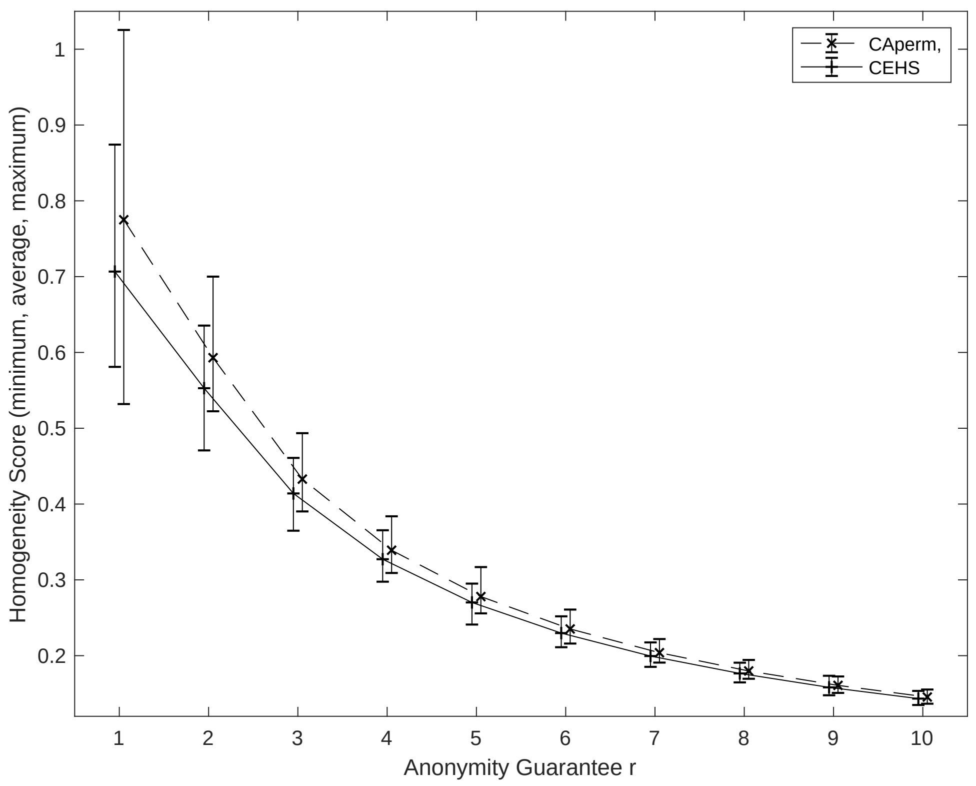


Figure 4.21: Comparison of homogeneity scores for anonymizing arrays constructed by CEHS and CAperm when number of rows has been equalized by appending random rows. Top bar is maximum, midpoint is average, and bottom bar is minimum homogeneity score for each array.

This is not to say that CEHS produces arrays with lower homogeneity, though the evidence for $r = 2$ suggests that it does. Instead, the claim is that whether the goal is fewer rows or lower homogeneity, CEHS is tailored to anonymizing arrays and seems to perform as well or better than the copy construction on covering arrays. Specifically, CEHS likely produces anonymizing arrays with fewer rows than the copy construction, especially as r increases. Further, constructing to meet the anonymity guarantee with CEHS and then appending random rows produces arrays with lower homogeneity than the permuted copy construction.

4.4.4 Evaluation of HP

The Homogeneity Post-optimization algorithm contains a number of tunable parameters: *maxGenerations*, *blocksize*, S, and *mutationRate*. It is outside the scope of this work to tune the parameters to provide guidance on the best setting as is done in Chapter 3. Instead, we conduct small experiments to provide a proof of concept that the algorithm does reduce the homogeneity of a provided array and to provide support for choices made in algorithm design. The array on which HP is performed is an $\mathsf{AA}(62; 3, 2, 10, (5^1 4^2 3^3 2^2))$ generated by CEHS with constraints $\mathcal{H}_2$ in Table 4.10. We execute the algorithm once for each combination of $maxGenerations = \{100, 500, 1000, 1500\}$ and $numChildren = \{10, 20\}$. As one expects, the trend is that the algorithm produces a result with lower homogeneity scores when provided more children to choose from and is run for more generations. A few exceptions are likely due to the randomness of the algorithm. The results are shown in Table 4.11 where each cell contains a tuple (minimum local homogeneity, global homogeneity, maximum local homogeneity) for the corresponding combination of *maxGenerations* and *numChildren*. The homogeneity scores for the ingredient array are $(0.628, 0.675, 0.792)$. An example of the run with $maxGenerations = 1000$ and $numChildren = 20$ is in Figure 4.22.

We also compare using maximum local homogeneity for the fitness function instead of global homogeneity for a subset of the previous combinations. These scores are in Table 4.12. In every case, the *maxHomogeneity* as the fitness function produces higher global and maximum local homogeneity scores. In half of the cases, the minimum local homogeneity score is lower when *maxHomogeneity* is used. We conclude that *globalHomogeneity* is likely the better choice for the fitness function as it appears to reduce the overall and worst case homogeneity.

Table 4.11: Comparison of parameters *maxGenerations* and *numChildren*

	numChildren	
maxGenerations	10	20
100	(0.632, 0.662, 0.689)	(0.630, 0.654, 0.682)
500	(0.630, 0.652, 0.672)	(0.624, 0.652, 0.668)
1000	(0.621, 0.649, 0.677)	(0.620, 0.652, 0.673)
1500	(0.635, 0.657, 0.682)	(0.620, 0.651, 0.672)

Table 4.12: Comparison of fitness using *globalHomogeneity* versus *maxHomogeneity*

	numChildren	
maxGenerations	10	20
	globalHomogeneity	
100	(0.632, 0.662, 0.689)	(0.630, 0.654, 0.682)
500	(0.630, 0.652, 0.672)	(0.624, 0.652, 0.668)
	maxHomogeneity	
100	(0.637, 0.672, 0.695)	(0.627, 0.667, 0.688)
500	(0.619, 0.660, 0.677)	(0.632, 0.657, 0.677)

All runs reduce the global homogeneity and maximum local homogeneity from the original array, though a few increase the minimum local homogeneity.

There are scenarios where HP is not able to reduce the homogeneity of the array. For example, consider the covering array $\mathsf{CA}(21; 2, 7, 4)$ from Chapter 3. An $\mathsf{AA}(42; 2, 2, 7, 4)$ is produced by CAcopy for $r = 2$. The homogeneity scores are (0.569, 0.642, 0.735) with an expectation of 0.571. Therefore, this array is more homogeneous than expected, likely due in part to the replication of rows. However, no matter how many children or generations are allowed, HP does not produce any child array with lower homogeneity. An array built by CEHS, $\mathsf{AA}(45; 2, 2, 7, 4)$, has only 3 extra rows. The homogeneity scores are (0.452, 0.561, 0.649) with expectation 0.554 and, in just 100 generations with 10 children, HP reduces the scores to (0.442, 0.529, 0.607). When three rows that do not appear in the array made by CAcopy are added

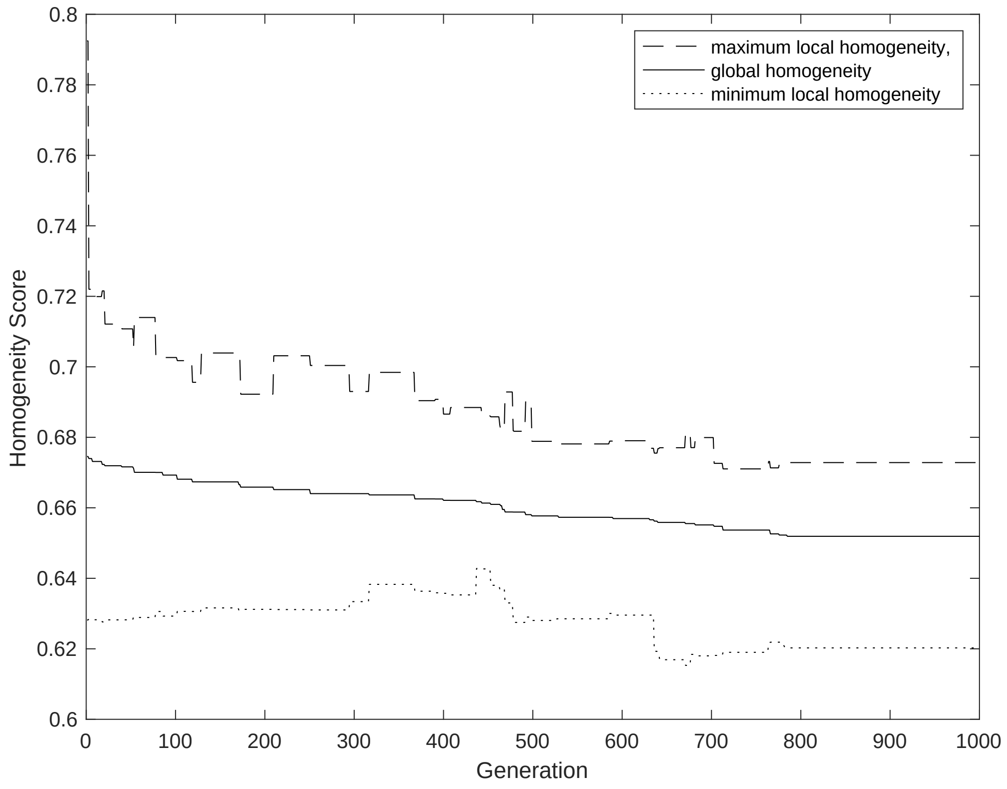


Figure 4.22: Reduction in minimum, maximum and global homogeneity scores over 1000 generations with 20 children for $\mathsf{AA}(62; 3, 2, 19, (5^1 4^2 3^3 2^4))$

to the $\mathsf{AA}(42; 2, 2, 7, 4)$, the algorithm makes improvement, though not as rapidly. See Figure 4.23 for a comparison of the reduction of scores for these two arrays.

It is possible that the covering array is too efficient and the lack of redundant rows prevents room to move credentials around. Another possibility is that there is something in the structure of this particular array. When swapping a credential, the other credentials of the access profile that involve the same rows are “broken.” If there is not enough redundancy, this can cause some credential to not appear r times in the array, and mutation is introduced to handle this. However, mutation of a single row may not be enough and multiple mutations may be needed.

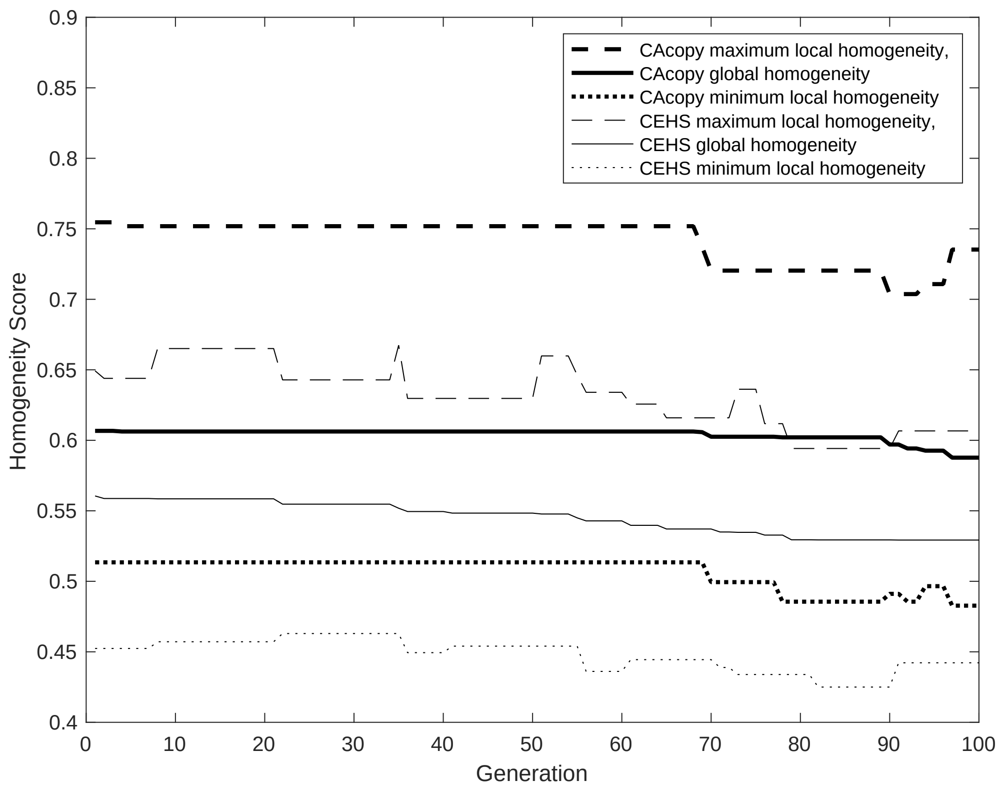


Figure 4.23: Comparison of reduction on arrays made by CAcopy and CEHS

4.5 κ-Anonymity for Statistical Databases

There is a large body of work on anonymizing the data in statistical databases to allow records to be released without revealing sensitive or identifying information about the individuals corresponding to the records while preventing loss of statistical value of the aggregate data [59]. The privacy goal is to prevent the use of "quasi-identifiers" in linking records from the table being released to other databases. If linking occurs, individuals are re-identified in the released data and a "sensitive attribute" of one or more individuals is disclosed. While anonymizing a set of access profiles for attribute-based authorization and anonymizing records in a statistical database both share the goal of providing anonymity up to a set degree, the applications are different and so the techniques tend to be different. In the κ-anonymity problem, when the anonymized table is released, attacks attempt to determine the individual associated with a record or some sensitive information about an individual, such as the likelihood that individual A has attribute B. In our problem, all attributes of subjects are assumed to be known to the system when subjects register. Instead, we seek to prevent the system from identifying the subject presenting a credential at the moment of an access control decision.

In the κ-anonymity literature, k is the anonymity degree, while in the covering array literature that serves as the basis for creating a combinatorial design to solve the attribute distribution problem, k is the number of columns in an array. Therefore, we use k for the columns of an anonymizing array and the number of attributes. Throughout this section, we use the symbol κ for κ-anonymity to avoid confusion.

An anonymizing array satisfies the property of "κ-anonymity" in [58] where every t-set of attributes is a "quasi-identifier" and $r = \kappa$. A relaxation of κ-anonymity, (κ, ℓ)-anonymity is defined for statistical databases when κ-anonymity causes excessive information loss and is equivalent to our definition of (r, t)-anonymous [57]. The primary methods for κ- and (κ, ℓ)-anonymity are *generalization*, replacing an attribute value with a less specific value based on the semantics of the domain of the attribute, and *suppression*, not releasing a value at all. These are applicable to the issue of anonymizing potentially sensitive data from a statistical database. For the κ-anonymity generalization methods to be useful in the attribute-based authorization domain, there must exist a "domain generalization hierarchy" among values of an attribute. For example, "Ph.D. student" and "Masters student" might be generalized to "graduate student" and possibly further generalized to "student." *Symbol fusing*, or mapping two symbols of a column of an array into one, is an operation used in covering arrays that is similar to generalization and might be useful when attribute values do not have a hierarchy. It is not clear when these techniques are applicable in access control scenarios as attributes are not typically mutable.

The ℓ-diversity principle is a set of properties for databases satisfying κ-anonymity that requires that “sensitive values” within an “equivalence class” of a quasi-identifier, what we refer to as the neighborhood of a credential, appear not too frequently [39]. The purpose of ℓ-diversity is to protect against probabilistic inference attacks. This has a similar purpose as our homogeneity metrics, but homogeneity measures relationships between access profiles rather than counting the appearance of certain attribute values. It is not clear if ℓ-diversity is a useful metric for anonymizing arrays.

When an entire row is treated as a quasi-identifier, κ-anonymity requires that κ identical rows exist. The computational complexity of achieving κ-anonymity of a matrix with N rows by suppression, replacing attribute values with $\star$, is given in [4]. In their work, *input (output) homogeneity* is the number of different input (output) rows. In our problem, we require r identical credentials, but we prefer that entire access profiles not be identical. Identical access profiles lead to high homogeneity, which is undesirable in attribute-based authorization as it may lead to tracking of a group of access profiles. This is similar to our problem when a credential is an entire row, $t = k$. It is not clear if their results generalize to our problem when $t < k$ and where the sets of attributes we seek to anonymize overlap necessarily.

Chapter 5

CONCLUSION

5.1 Interaction Testing Conclusions

In Chapter 2, we develop three algorithms for Sherwood Covering Perfect Hash Family (SCPHF) construction. We combine a greedy algorithm, Conditional Expectation (CE), with a recursive column replacement approach to build covering arrays via Sherwood covering perfect hash families. CE is a greedy algorithm that repeatedly chooses a symbol that is as good as the expectation for a row. When choosing a symbol for a column, it only considers the uncovered t-sets involving that column with at least one fixed symbol. CE finds useful new intermediate-sized SCPHFs and, by column replacement, covering arrays. Additionally, the SCPHFs found by the CE algorithm serve as good ingredients for the Random Extension (RE) algorithm in [19] and our recursive algorithms. The combination of CE-RE has been shown to produce SCPHFs that expand to covering arrays with the fewest rows known for some problem parameters.

Composition is a viable cut-and-paste recursive technique for building larger arrays from small ingredients and allows us to construct covering arrays for numbers of factors that are infeasible by direct construction, though it often results in more rows. The first algorithm, Composition (COMP), makes m copies of an ingredient SCPHF to form an SCPHF with km columns. Its efficiency results from covering a large fraction of the $\binom{km}{t}$ t-sets and leaving only those t-sets that have more than one copy of a column from the ingredient. CE is executed to cover the t-sets that remain. Affine Composition (AF-COMP) reduces the occurrences of exact copies of columns of the ingredient appearing in the composed SCPHF by applying an affine transformation to each row of each subarray. In practice, it is substantially faster than COMP and produces SCPHFs that compare favorably in number of rows to CE.

AF-COMP is possible because the classes of noncovering and covering tuples are closed under arithmetic, rows of the SCPHF are independent, and the same affine transformation is applied to all columns of a row of a subarray. Therefore, each subarray of the composed AF-COMP is guaranteed to be an SCPHF. Another benefit of AF-COMP is that it retains subspace restrictions that are used to form redundant rows in the SCPHF [18]. That is, an SCPHF constructed with a subspace restriction produces another SCPHF with the same restriction after AF-COMP. The redundant rows can then be removed from the composed SCPHF in a similar manner as shortened permutation vectors.

An affine transformation involves a choice of adder and multiplier for each coordinate of a permutation vector. This results in $(v(v-1))^{N(t-1)}$ possible affine transformations. Currently, AF-COMP selects transformations randomly, but some choices of transformations may be better than others conditional on the transformations chosen for previous subarrays. An open problem is to determine a strategy for choosing good sets of affine transformations. Without knowledge of the permutation vectors in an ingredient SCPHF and the affine transformations chosen, at best we can guarantee that AF-COMP covers $\binom{k}{t}m$ of the $\binom{km}{t}$ t-sets in the composition phase, less than what is guaranteed by COMP, yet AF-COMP covers substantially more in practice. A conditional expectation approach to choose the affine transformation that results in the fewest uncovered t-sets appears to require extensive computation [18]. A second open problem is to determine if there is a better bound for the number of t-sets covered by AF-COMP or if the expected number can be computed. Another direction is to examine if a restriction on the affine transformations chosen can lead to a computation of the expectation of the number of uncovered t-sets. This would be useful as the number of t-sets to cover leads to an upper bound on the expected number of rows.

5.2 Fault Location Conclusions

In Chapter 3, we develop the Partitioned Search with Column Replacement (PSCR) algorithm. PSCR includes the first locating array verifier for an arbitrary number of faults and builds locating arrays that are, in general, better than the higher strength covering array construction when $v > 2$. As it does not rely on an underlying mathematical structure, it can be applied to any number of symbols (provided that $d < v$ due to the locating array definition) and is demonstrated to be competitive with other available search techniques for mixed-level locating arrays (MLAs). PSCR includes options to prioritize speed, accuracy, or a combination of both for a particular set of problem parameters, and guidance on how to set the parameters has been provided. PSCR can build a locating array from an ingredient array without disturbing the initial rows to provide a set of required tests that must be present in the final test suite. PSCR is both a useful tool to construct locating arrays for testing purposes and a beneficial research resource for verifying locating array candidates made by other constructions.

The running time of the algorithm may be improved by replacing the insertion sorted list with a hash table, which requires examining the kind of hash function that is useful here and ensuring that collisions in the sense that we have defined them can be differentiated from hash function collisions. Based on the results of our testing, the algorithm can be streamlined with *CHO* options 1 and 2 removed. The adaptive option should be rewritten to use only the column option along with a tunable early exit based on the ratio of collisions found to the amount of the array checked. The ingredient build feature may be useful for testing other construction theories such as vertically stacking copies of a strength t covering array and allowing resampling only on the redundant copies. It remains to be seen how the ingredient building approaches compare as the number of columns grows, especially as this relates to whether the ingredient rows should be retained or allowed to be resampled.

A practical issue with the locating array definition occurs when $d \geq v_i$ for some column i [40]. In this case, some interactions are necessarily not locatable and thus a locating array cannot exist, but location of the remaining interactions is possible; relaxing the requirements yields partial location. Similarly, a constraint is an interaction that cannot be tested. Constrained locating arrays (CLAs) are introduced in [30], and satisfiability-based approaches have been used to generate CLAs with a focus on (1,2)-locating [29]. For locating arrays, the presence of constraints may prevent other interactions from being locatable. A comprehensive solution for handling constraints might be useful to also accommodate partial location.

5.3 Attribute-Based Authorization Conclusions

Access control decisions made on the basis of attributes afford the opportunity for anonymous authorization, but do not guarantee it when the distribution of attributes allows for the composition of policies that one or few subjects possess the credentials to satisfy. In Chapter 4, we propose anonymizing arrays as a mechanism for attribute distribution so that if credentials are restricted to t or fewer attributes, subjects cannot be identified with greater than $\frac{1}{r}$ probability. We provide an algorithm that computes r given an anonymizing array and maximum credential size t.

Anonymizing arrays are related to covering arrays with constraints and higher λ. We prove several theorems that explain this relationship and provide insight on when and how a covering array extends to an anonymizing array. Many mathematical methods for building covering arrays restrict the parameters allowed and fail to handle constraints, but computational approaches for covering array construction are often adaptable for building anonymizing arrays. We develop two algorithms, Moser-Tardos-style Column Resampling (MTCR) and Conditional Expectation Heuristic Search (CEHS), that build anonymizing arrays either from scratch or to add padding rows to a given set of access profiles to achieve the anonymity guarantee. Both algorithms handle constraints. CEHS provides better performance in terms of number of rows produced than MTCR in most cases. While CEHS may have cases that abort due to failing to avoid a hard constraint in a particular execution, we found no cases where CEHS fails to terminate or where it cannot find a solution given more than one run when a solution exists. The conflicting goals of having r coverage and avoiding hard constraints makes MTCR a poor choice for this problem. The more rows it is given to work with, the more likely each credential is repeated r times, but this also increases the likelihood of a hard constraint appearing in the array. Finding a solution even in "small" instances may require billions of iterations, while CEHS builds an array for the same case in one to two seconds. We also compare CEHS to a construction that makes copies of a covering array. For $t > 1, r > 1$, CEHS always produces arrays with fewer rows than the copy construction in our tests.

We additionally provide metrics, local and global homogeneity, to compare two anonymizing arrays on the same parameters, and we provide an algorithm to compute homogeneity scores. The homogeneity metrics are useful to determine progress while mutating an anonymizing array or when comparing two anonymizing arrays with the same parameters, but the inverse correlation between rows and homogeneity makes them difficult to use for arrays of different sizes. A standardized metric that is independent of the number of rows is needed. We compare CEHS and the copy construction on homogeneity. A random permutation is selected for each column of each copy of the array built from copies of a covering array. We show that CEHS produces arrays with lower homogeneity when the number of rows are equalized by adding random rows to the array with fewer rows, even when the copies are permuted. We also develop an algorithm, Homogeneity Post-optimization (HP), that reduces the homogeneity of an array by swapping credentials between access profiles that have low closeness scores to break up groups of access profiles that share many credentials.

Our model assumes that all attributes are known to the system and are issued by the same authority, thus any set of t attributes constitutes a credential. In real world scenarios, it may be the case that different attributes are issued by different authorities [7]. The implementation of a credential – a card preloaded with attribute values, a certificate proving possession, or a private key in Ciphertext-Policy Attribute-Based Encryption (CP-ABE) – may dictate that only attributes from the same authority appear together in a credential. That is, their attribute sets are disjoint. Anonymizing arrays are (r, t)-anonymous for any t-set of attributes. Anonymizing arrays cover all of the potential credentials, but in a multi-authority case, this may be unnecessary. This is not the same as a hard constraint; the attributes from two authorities can and likely do appear in the same access profile, but we do not need to provide r repetitions of credentials that cross boundaries of different authorities as they are never used in a policy. One solution here is to add an additional type of soft constraint, "don't care," that does not require those credentials to appear exactly zero or r times.

The set system nature of anonymizing arrays is best suited when attribute levels are categorical. When attribute levels are numerical, such as *age*, and a policy can be created, such as $age > 30$, it is not clear how best to extend anonymizing arrays to work in this case. One solution is to create attributes that are age ranges, such as $age = \{[20, 29], [30, 39], [40, 49], \ldots\}$ and then list policies again in disjunctive normal form, e.g., $[30, 39] \vee [40, 49]$. Another solution is to enumerate all possible acceptable ages as attribute levels. This works when the levels are discrete, but not when they are continuous.

We might ask whether an anonymizing array can be used with an access control list or capability list. Access control lists are mappings of subjects to permissions with one access control list per object, while capability lists are mappings of objects to permissions with one capability list per subject. In both cases, identity of the subject is required in order to determine whether the requested permission should be granted. In our approach, the identity-less nature of attribute-based access control (ABAC) is necessary though not sufficient to achieve anonymity. If there is some scenario where it is required that the subject present her identity, then this is not an application where anonymizing arrays are appropriate.

An open problem for ABAC is how to achieve *separation of duties* or requiring more than one subject to complete sensitive tasks to reduce the risk of error and fraud [53]. In role-based access control (RBAC), static separation of duties can be achieved by defining two mutually exclusive roles and restricting a subject from being assigned both roles required for the task. As the notion of roles can be supported in ABAC by attribute distribution and policy engineering, we hypothesize that separation of duties can be achieved in ABAC by defining one rule for each "role" and requiring that the two rules contain at least one attribute in common but with different values. Suppose R_1, R_2 are rules and $a_1 = 1$ is an attribute a_1 with value 1. Then the set of rules $R_1 : a_1 = 1 \wedge a_2 = 2, R_2 : a_1 = 0 \wedge a_3 = 3$ require two different subjects because one access profile cannot have both $a_1 = 1$ and $a_1 = 0$ simultaneously if attributes can take on a single value. If attributes can be assigned sets of values, we can modify our approach depending on the implementation of the system, such as $R_1 : a_1 = 1 \wedge a_1 \neq 0 \wedge a_2 = 2$, etc., or we could attempt to use a Chinese

Wall-like approach and devise policies that state that if the subject has presented one attribute to obtain one permission, she may not present a conflicting attribute for a mutually exclusive permission in the same session. Attribute distribution and policy engineering for ABAC together may provide a rich environment in which to find solutions for other open problems.

A problem for most access control systems is *least privilege* or only allowing the permissions necessary to complete the job functions of the subject. There is a natural parallel with the trade-off between greater privileges and greater privacy. As a subject presents more attributes, the risk of being identified increases; as proven, there is an inverse relationship between t and r. Therefore, the subject desires to present the fewest attributes that allow access. One approach is to require credentials composed of more attributes in order to acquire greater privileges. That is, a system guarantees a degree of anonymity r for credential size t to grant minimum privileges. A tiered structure may be devised so that a lower degree of anonymity $r - q$ is guaranteed when the subject presents more attributes $t+s$, but the subject also receives increased privileges.

When padding is added to reach a given anonymity guarantee, the preference is to add as few access profiles as possible as maintaining the anonymizing array requires storage and searching a large array can create computational efficiency challenges. In this work, we develop two algorithms to add rows to a partial anonymizing array. A bound on how many rows are needed to complete a partial anonymizing array is an open problem. In the worst case, when $t = k$ and each of the N access profiles already in the array is unique, $N(r-1)$ replicates are required to be (r, t)-anonymous. When t is smaller than k, we expect to need fewer than $r-1$ replicates of each of the original N access profiles, even if each access profile is unique, due to some t-subset agreement by the original access profiles. Pick one of the $\binom{k}{t}$ subsets of columns, T, for some access profile. The probability that T agrees in all t positions with some other arbitrary access profile is $\frac{1}{v^t}$. By linearity of expectation, the number of access profiles that agree with this one on T is $(N-1)(\frac{1}{v^t})$. For fixed N, when t is small, we expect a larger number of agreements and as t approaches k, we expect a smaller number. If $(r-1) \leq (N-1)(\frac{1}{v^t})$, no additional rows are expected for T to be repeated r times. Of course, T is only one of the $\binom{k}{t}$ interactions for one access profile, but the expected value of agreement is the same for all of them. Then we expect to add $(r-1)-(N-1)(\frac{1}{v^t})$ replicates of each access profile on average, and we need at most $(r-1)-\min_{c\in\mathcal{C}}(c)$ where $\mathcal{C}$ is the set of $\binom{k}{t}v^t$ credentials. A credential may not need all of the replicated rows in order to appear r times. Once it has reached r repetitions, the places in which it appears after this row in the array can be used to replicate credentials of the same t-set for other access profiles. Suppose the first t-subset of attributes for access profile i already has r replicates, and access profile j is only missing replicates for these positions. Merging the replicates together using the first t positions of i's duplicates for the values needed for j instead reduces the number of added rows. A construction algorithm exploiting "don't care" positions may be useful for constructing anonymizing arrays with fewer rows.

Maintenance of access profiles when subjects leave the system is an open problem. Ideally, the access profile is deleted, but that could cause some credentials to appear fewer than r times. The simplest option is to leave the access profile in place as a padding row, but this could result in a larger anonymizing array than necessary. Another approach might be to modify the access profile to one that best balances the array. For example, the attributes that are required for meeting the anonymity guarantee are kept and others are replaced by the attributes that are most likely to be needed if some other access profile is deleted, such as those that appear the fewest times in the array. An algorithm for anonymizing array maintenance that prevents retaining too many padding rows and post-optimization for row reduction via necessity analysis seems like a reasonable starting point.

Locating arrays are designed to locate the interactions participating in failures rather than simply indicating that a test is failing. Anonymizing arrays are designed so that you can not locate the specific access profile that is being used for an access control decision. Superficially, these seem to be dual concepts, but it does not appear to be the case that an anonymizing array is equivalent to an "anti-locating array." We use the term interaction synonymously with the term credential in order to discuss both types of arrays. In a locating array, the goal is to facilitate the location of interactions, so as rows are added, we attempt to place previously correlated interactions into different rows to distinguish them. This array is also (r, t)-anonymous, where r is the number of appearances of the least frequently appearing interaction, and r grows as we increase the replication of interactions. In an anti-locating array, if we were to propose such a thing, the goal is to prevent location of interactions, so as rows are added, we continue to place correlated interactions together into the same rows. This array is also (r, t)-anonymous, where r is the number of appearances of the least frequently appearing interaction, and r grows as we increase the replication of interactions. There does not seem to be a difference between locating arrays and anti-locating arrays in how they relate to the simple definition of anonymizing arrays. Where they do differ is in the homogeneity of the anonymizing array. In a locating array, an interaction appears with many other interactions. If it always appears with the same interactions, they share the same set of rows. Conversely, in an anti-locating array, an interaction may appear frequently, but with few other interactions, forming a small neighborhood so that interactions cannot be distinguished in terms of their rows. That is not to say that an anonymizing array is simply a locating array, as we can fall short of location and yet still achieve our objective of anonymity, but that techniques to build locating arrays may be good for building low homogeneity anonymizing arrays and vice versa. An open problem is if an (r, t)-anonymous array with no constrained credentials is constructed while prioritizing low homogeneity, are there cases where it is guaranteed to be (d, t')-locating?

An idea from the experimental design literature that appears to be similar to anonymizing is aliasing. Two or more effects that have the property that the same linear combination of observations is used to estimate their effects are aliases, and this renders it impossible to differentiate between them [41]. In a design table, the observations are rows, factors are columns, and effects are interactions of varying sizes. Aliasing is another way to describe when interactions appear together in the same rows. The more a set of interactions appears together, the more strongly they are aliased. Aliasing seems desirable for anonymity — to not be able to tell things apart — but instead the reverse appears to be true. In the design literature, aliasing primarily occurs in fractional factorial designs. As rows are added, the preference is for rows that have not been seen already, as repeated rows cannot result in de-aliasing. Then as the number of rows increases, aliasing decreases, and repetition of interactions increases. This suggests an inverse relationship between aliasing and anonymity, which is not surprising because aliasing is a relationship between columns while anonymity is achieved by a relationship between rows. There is a relationship between how to choose rows to break aliasing and how to choose rows for location. In both cases, the goal is to break up correlations between sets of columns. It seems likely that designs with low aliasing could correspond to anonymizing arrays with low homogeneity, but this has yet to be confirmed.

5.4 Algorithms for t-Restriction Problems

From a macroscopic view, the testing problems seem to have nothing in common with attribute-based authorization, but when examined at a finer degree of granularity, commonalities become apparent. For testing, the consideration is at first ensuring that the coverage demands for t-way interactions are satisfied in the test suite. When the goal is to not just detect presence of faults but to identify the faulty interactions, distinguishing t-way interactions is added to the coverage demand. To guarantee anonymity in attribute-based authorization, the demand is for coverage of the sets of t-sized credentials to be either zero or r. A locating array is a covering array and an anonymizing array is a covering array of higher λ with constraints.

Due to the commonalities in the problem structure, techniques are adaptable from one problem to the next. The conditional expectation algorithm developed for SCPHFs provides a useful building block for constructing anonymizing arrays. The computation for the best symbol is adapted to accommodate the higher coverage and a heuristic that applies a cost relative to the probability of a future event is added to avoid the placement of constraints. The cut-and-paste recursive algorithms, COMP and AF-COMP, have parallels in the copy construction and permuted copy constructions that can be used to create anonymizing arrays from covering array ingredients. PSCR is inspired by the Moser-Tardos algorithm for building covering arrays, and PSCR can build a locating array from a covering array ingredient.

Yet from a microscopic view, these are still unique problems. The Moser-Tardos style column resampling that is effective for locating arrays does not perform well in the presence of constraints, so it proves to be not very useful for constructing anonymizing arrays. The diversity metric for covering arrays is not useful for comparing anonymizing arrays on their tendency to form small, tight-knit groups of access profiles, requiring the development of the homogeneity metrics which are specific to anonymizing arrays. And while obscurity seems to be the opposite of locatability, it does not seem to be the case that an anonymizing array is somehow the opposite of a locating array.

Our final conclusion is that t-restriction problems may appear in disguise, but formalizing a t-restriction problem as such allows for ideas that have been proven useful for interaction testing and fault location to be applied in a seemingly unrelated area. These ideas can then be tailored to the specific problem at hand. We need not stop there for inspiration. Practices from design of experiments prove to be useful for tuning algorithmic parameters and basic concepts from genetic algorithms provide a post-optimization method for improving the homogeneity of anonymizing arrays.

REFERENCES

[1] A. N. Aldaco, C. J. Colbourn, and V. R. Syrotiuk, “Locating arrays: A new experimental design for screening complex engineered systems,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 31–40, Jan. 2015.

[2] M. Backes, J. Camenisch, and D. Sommer, “Anonymous yet accountable access control,” in *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*. ACM, 2005, pp. 40–46.

[3] J. Bethencourt, A. Sahai, and B. Waters, “Ciphertext-policy attribute-based encryption,” in *2007 IEEE Symposium on Security and Privacy (SP ’07)*. IEEE, May 2007, pp. 321–334.

[4] R. Bredereck, A. Nichterlein, R. Niedermeier, and G. Philip, “The effect of homogeneity on the computational complexity of combinatorial data anonymization,” *Data Mining and Knowledge Discovery*, vol. 28, no. 1, pp. 65–91, 2014.

[5] R. C. Bryce and C. J. Colbourn, “A density-based greedy algorithm for higher strength covering arrays,” *Software Testing, Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[6] ——, “Prioritized interaction testing for pair-wise coverage with seeding and constraints,” *Information and Software Technology*, vol. 48, no. 10, pp. 960 – 970, 2006.

[7] M. Chase, “Multi-authority attribute based encryption,” in *Theory of Cryptography*, S. P. Vadhan, Ed. Berlin, Heidelberg: Springer, 2007, pp. 515–534.

[8] M. A. Chateauneuf, C. J. Colbourn, and D. L. Kreher, “Covering arrays of strength three,” *Designs, Codes and Cryptography*, vol. 16, no. 3, pp. 235–242, May 1999.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi, “Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, Sep. 2008.

[10] M. B. Cohen, “Designing test suites for software interaction testing,” Ph.D. dissertation, The University of Auckland, 2004.

[11] C. J. Colbourn, B. Fan, and D. Horsley, “Disjoint spread systems and fault location,” *SIAM Journal on Discrete Mathematics*, vol. 30, no. 4, pp. 2011–2026, 2016.

[12] C. J. Colbourn and V. R. Syrotiuk, “Coverage, location, detection, and measurement,” in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2016, pp. 19–25.

[13] ——, “On a combinatorial framework for fault characterization,” *Mathematics in Computer Science*, vol. 12, no. 4, pp. 429–451, Dec. 2018.

[14] C. J. Colbourn. Covering array tables for t=2,3,4,5,6. [Online]. Available: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

[15] ——, “Constructing perfect hash families using a greedy algorithm,” in *Coding and Cryptology*, Y. Li, S. Ling, H. Niederreiter, H. Wang, C. Xing, and S. Zhang, Eds. Singapore: World Scientific, 2008, pp. 109–118.

[16] ——, “Covering arrays and hash families.” in *Information Security, Coding Theory and Related Combinatorics : Information Coding and Combinatorics*, D. Crnkovič and V. Tonchev, Eds. Amsterdam: IOS Press, 2011, pp. 99–135.

[17] C. J. Colbourn and B. Fan, “Locating one pairwise interaction: Three recursive constructions,” *Journal of Algebra Combinatorics Discrete Structures and Applications*, vol. 3, no. 3, pp. 127–134, 2016.

[18] C. J. Colbourn and E. Lanus, “Subspace restrictions and affine composition for covering perfect hash families,” *Art of Discrete and Applied Mathematics*, vol. 1, no. 2, pp. 1–19, 2018.

[19] C. J. Colbourn, E. Lanus, and K. Sarkar, “Asymptotic and constructive methods for covering perfect hash families and covering arrays,” *Designs, Codes and Cryptography*, vol. 86, no. 4, pp. 907–937, Apr 2018.

[20] C. J. Colbourn and D. W. McClary, “Locating and detecting arrays for interaction faults,” *Journal of Combinatorial Optimization*, vol. 15, no. 1, pp. 17–48, Jan. 2008.

[21] C. J. Colbourn and P. Nayeri, “Randomized post-optimization for t-restrictions,” in *Information Theory, Combinatorics, and Search Theory*, H. Aydinian, F. Cicalese, and C. Deppe, Eds., vol. 7777. Berlin, Heidelberg: Springer, 2013, pp. 597–608.

[22] R. Compton, M. T. Mehari, C. J. Colbourn, E. D. Poorter, and V. R. Syrotiuk, “Screening interacting factors in a wireless network testbed using locating arrays,” in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 650–655.

[23] E. Estrada and J. A. Rodriguez-Velazquez, “Complex networks as hypergraphs,” *arXiv preprint physics/0505137*, 2005.

[24] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker, “Identifying failure-inducing combinations in a combinatorial test set,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 370–379.

[25] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2006, pp. 89–98.

[26] A. Hartman, "Software and hardware testing using combinatorial covering suites," in *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, M. C. Golumbic and I. B.-A. Hartman, Eds. Boston, MA: Springer US, 2005, pp. 237–266.

[27] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1, pp. 149–156, 2004.

[28] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone *et al.*, "Guide to attribute based access control (abac) definition and considerations (draft)," *NIST special publication*, vol. 800, no. 162, 2013.

[29] H. Jin, T. Kitamura, E. Choi, and T. Tsuchiya, "A satisfiability-based approach to generation of constrained locating arrays," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 285–294.

[30] H. Jin and T. Tsuchiya, "Constrained locating arrays for combinatorial interaction testing," *arXiv preprint arXiv:1801.06041*, 2017.

[31] A. Kapadia, P. P. Tsang, and S. W. Smith, "Attribute-based publishing with hidden credentials and hidden policies." in *NDSS*, vol. 7. Citeseer, 2007, pp. 179–192.

[32] T. Konishi, H. Kojima, H. Nakagawa, and T. Tsuchiya, "Finding minimum locating arrays using a SAT solver," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Mar. 2017, pp. 276–277.

[33] D. R. Kuhn, E. J. Coyne, and T. R. Weil, "Adding attributes to role-based access control," *Computer*, vol. 43, no. 6, pp. 79–81, June 2010.

[34] D. R. Kuhn, V. Hu, D. F. Ferraiolo, R. N. Kacker, and Y. Lei, "Pseudo-exhaustive testing of attribute based access control rules," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2016, pp. 51–58.

[35] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, June 2004.

[36] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Combinatorial testing: Theory and practice," in *Advances in Computers*, A. Memon, Ed. Elsevier, 2015, vol. 99, ch. 1, pp. 1 – 66.

[37] D. R. Kuhn, R. N. Kacker, and Y. Lei, *NIST Special Publication 800-142. Practical combinatorial testing.* National Institute of Standards & Technology, 2010.

[38] E. Lanus, C. J. Colbourn, and D. C. Montgomery, "Partitioned search with column resampling for locating array construction," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops*, to appear.

[39] A. Machanavajjhala, M. Venkitasubramaniam, D. Kifer, and J. Gehrke, "ℓ -diversity: Privacy beyond κ -anonymity," in *22nd International Conference on Data Engineering (ICDE'06)(ICDE)*, 2006.

[40] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Locating errors using ELAs, covering arrays, and adaptive testing algorithms," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 4, pp. 1776–24, 2009.

[41] D. C. Montgomery, *Design and analysis of experiments*, 8th ed. Hoboken, NJ: John Wiley & Sons, Inc., 2013.

[42] R. A. Moser and G. Tardos, "A constructive proof of the general Lovász local lemma," *Journal of the ACM*, vol. 57, no. 2, pp. 11:1–11:15, Feb. 2010.

[43] T. Nagamoto, H. Kojima, H. Nakagawa, and T. Tsuchiya, "Locating a faulty interaction in pair-wise testing," in *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, Nov. 2014, pp. 155–156.

[44] P. Nayeri, "Post-optimization: Necessity analysis for combinatorial arrays," Ph.D. dissertation, Arizona State University, 2011.

[45] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[46] C. Nie, H. Wu, X. Niu, F.-C. Kuo, H. Leung, and C. J. Colbourn, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures," *Information and Software Technology*, vol. 62, no. 1, pp. 198–213, 2015.

[47] X. Niu, C. Nie, H. K. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang, "An interleaving approach to combinatorial testing and failure-inducing interaction identification," *IEEE Transactions on Software Engineering*, pp. 1–33, 2018.

[48] M. Portnoi and C.-C. Shen, "Location-enhanced authenticated key exchange," in *2016 International Conference on Computing, Networking and Communications (ICNC).* IEEE, 2016, pp. 1–5.

[49] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2005, pp. 457–473.

[50] K. Sarkar, "Covering arrays: Algorithms and asymptotics," Ph.D. dissertation, Arizona State University, 2016.

[51] JMP® statistical software from SAS, version 14.0.0. SAS. [Online]. Available: https://www.jmp.com

[52] S. A. Seidel, K. Sarkar, C. J. Colbourn, and V. R. Syrotiuk, "Separating interaction effects using locating and detecting arrays," in *Combinatorial Algorithms*, C. Iliopoulos, H. W. Leong, and W.-K. Sung, Eds. Cham: Springer International Publishing, 2018, pp. 349–360.

[53] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Computing Surveys*, vol. 49, no. 4, pp. 65:1–65:45, Feb. 2017.

[54] G. B. Sherwood, S. S. Martirosyan, and C. J. Colbourn, "Covering arrays of higher strength from permutation vectors," *Journal of Combinatorial Designs*, vol. 14, no. 3, pp. 202–213, 2006.

[55] C. Shi, Y. Tang, and J. Yin, "Optimal locating arrays for at most two faults," *Science China Mathematics*, vol. 55, no. 1, pp. 197–206, Jan. 2012.

[56] A. Squicciarini, A. Trombetta, A. Bhargav-Spantzel, and E. Bertino, "k-anonymous attribute-based access control," in *International Conference on Information and Computer Security (ICICS'07)*, 2007.

[57] K. Stokes, "On computational anonymity," in *International Conference on Privacy in Statistical Databases.* Springer, 2012, pp. 336–347.

[58] L. Sweeney, "Achieving k-anonymity privacy protection using generalization and suppression," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 571–588, 2002.

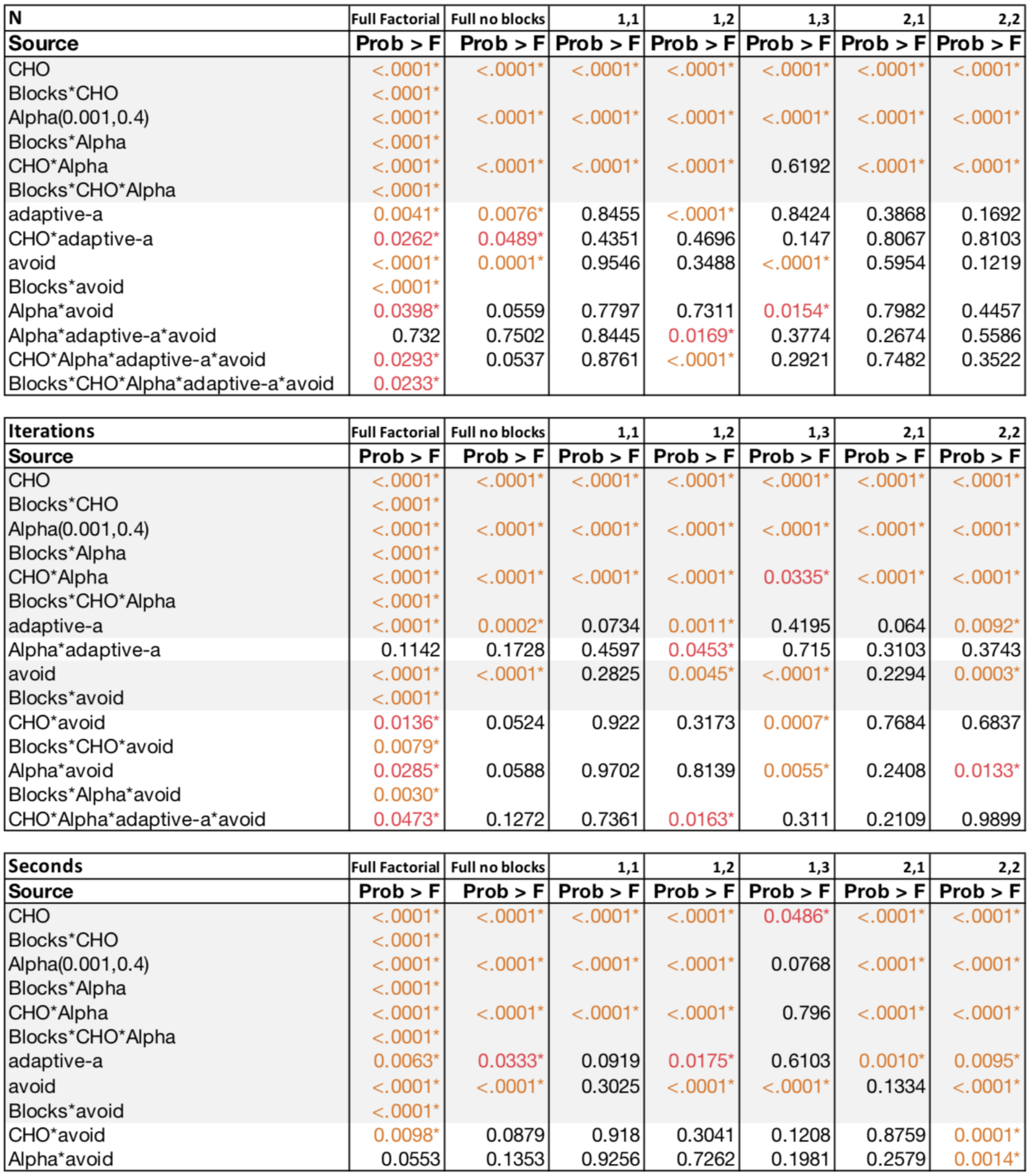
[59] ——, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[60] Y. Tang, C. J. Colbourn, and J. Yin, "Optimality and constructions of locating arrays," *Journal of Statistical Theory and Practice*, vol. 6, no. 1, pp. 20–29, 2012.

[61] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.

APPENDIX A

LOCATING ARRAY ADDITIONAL FIGURES



N	Full Factorial	Full no blocks	1,1	1,2	1,3	2,1	2,2
Source	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**
CHO	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*
Blocks*CHO	<.0001*						
Alpha(0.001,0.4)	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*
Blocks*Alpha	<.0001*						
CHO*Alpha	<.0001*	<.0001*	<.0001*	<.0001*	0.6192	<.0001*	<.0001*
Blocks*CHO*Alpha	<.0001*						
adaptive-a	0.0041*	0.0076*	0.8455	<.0001*	0.8424	0.3868	0.1692
CHO*adaptive-a	0.0262*	0.0489*	0.4351	0.4696	0.147	0.8067	0.8103
avoid	<.0001*	0.0001*	0.9546	0.3488	<.0001*	0.5954	0.1219
Blocks*avoid	<.0001*						
Alpha*avoid	0.0398*	0.0559	0.7797	0.7311	0.0154*	0.7982	0.4457
Alpha*adaptive-a*avoid	0.732	0.7502	0.8445	0.0169*	0.3774	0.2674	0.5586
CHO*Alpha*adaptive-a*avoid	0.0293*	0.0537	0.8761	<.0001*	0.2921	0.7482	0.3522
Blocks*CHO*Alpha*adaptive-a*avoid	0.0233*						

Iterations	Full Factorial	Full no blocks	1,1	1,2	1,3	2,1	2,2
Source	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**
CHO	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*
Blocks*CHO	<.0001*						
Alpha(0.001,0.4)	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*	<.0001*
Blocks*Alpha	<.0001*						
CHO*Alpha	<.0001*	<.0001*	<.0001*	<.0001*	0.0335*	<.0001*	<.0001*
Blocks*CHO*Alpha	<.0001*						
adaptive-a	<.0001*	0.0002*	0.0734	0.0011*	0.4195	0.064	0.0092*
Alpha*adaptive-a	0.1142	0.1728	0.4597	0.0453*	0.715	0.3103	0.3743
avoid	<.0001*	<.0001*	0.2825	0.0045*	<.0001*	0.2294	0.0003*
Blocks*avoid	<.0001*						
CHO*avoid	0.0136*	0.0524	0.922	0.3173	0.0007*	0.7684	0.6837
Blocks*CHO*avoid	0.0079*						
Alpha*avoid	0.0285*	0.0588	0.9702	0.8139	0.0055*	0.2408	0.0133*
Blocks*Alpha*avoid	0.0030*						
CHO*Alpha*adaptive-a*avoid	0.0473*	0.1272	0.7361	0.0163*	0.311	0.2109	0.9899

Seconds	Full Factorial	Full no blocks	1,1	1,2	1,3	2,1	2,2
Source	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**	**Prob > F**
CHO	<.0001*	<.0001*	<.0001*	<.0001*	0.0486*	<.0001*	<.0001*
Blocks*CHO	<.0001*						
Alpha(0.001,0.4)	<.0001*	<.0001*	<.0001*	<.0001*	0.0768	<.0001*	<.0001*
Blocks*Alpha	<.0001*						
CHO*Alpha	<.0001*	<.0001*	<.0001*	<.0001*	0.796	<.0001*	<.0001*
Blocks*CHO*Alpha	<.0001*						
adaptive-a	0.0063*	0.0333*	0.0919	0.0175*	0.6103	0.0010*	0.0095*
avoid	<.0001*	<.0001*	0.3025	<.0001*	<.0001*	0.1334	<.0001*
Blocks*avoid	<.0001*						
CHO*avoid	0.0098*	0.0879	0.918	0.3041	0.1208	0.8759	0.0001*
Alpha*avoid	0.0553	0.1353	0.9256	0.7262	0.1981	0.2579	0.0014*

Figure A.1: Comparison of effects in $5 \times 5 \times 3 \times 2 \times 2$ full factorial

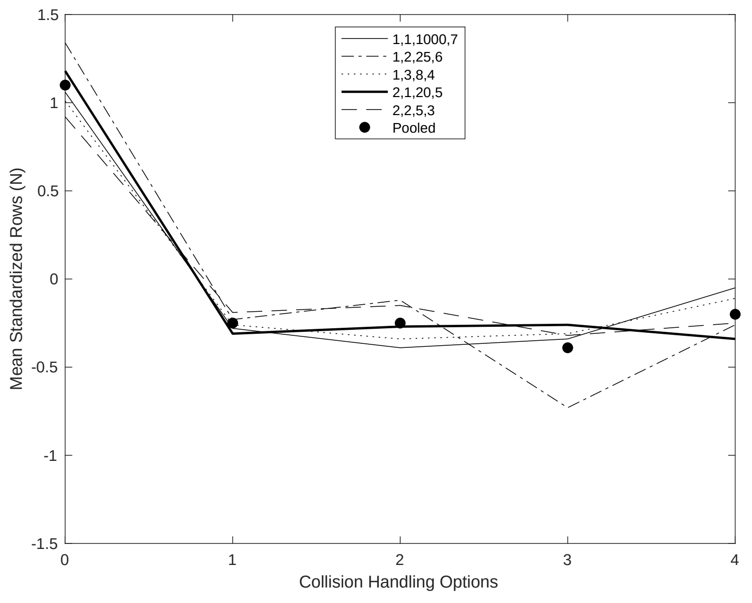


Figure A.2: Effect of *CHO* on rows by block

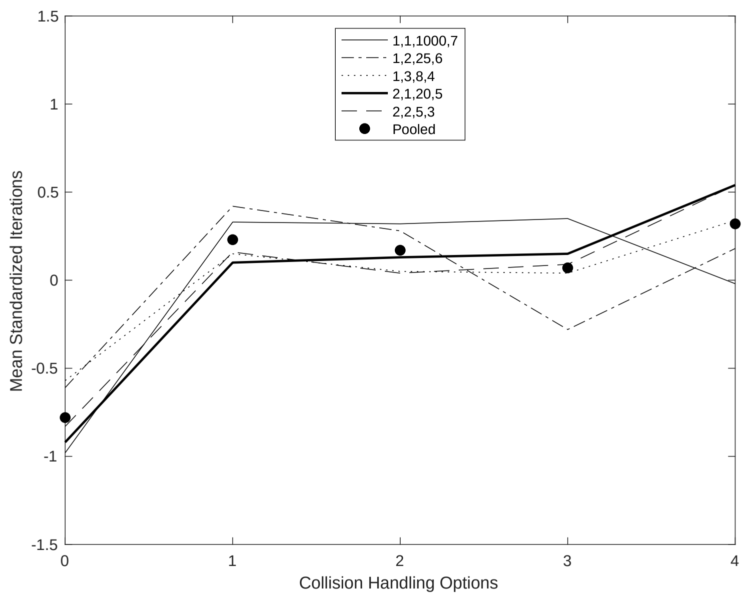


Figure A.3: Effect of *CHO* on iterations by block

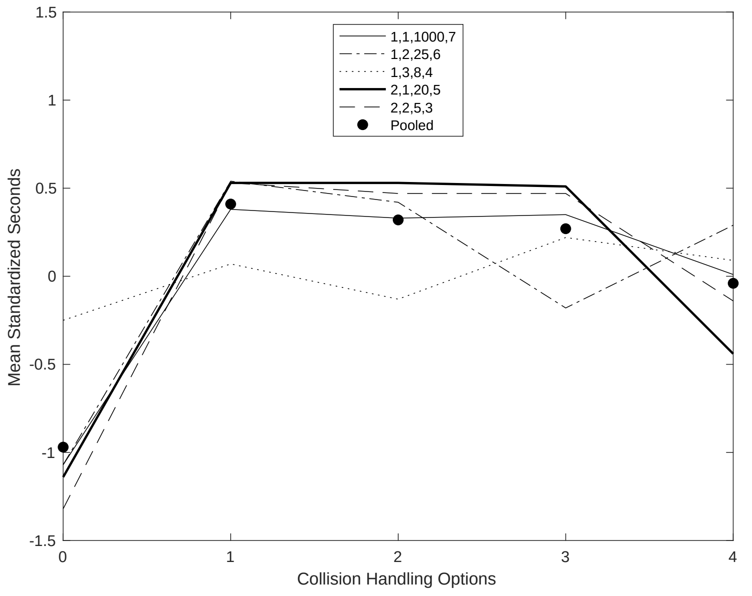


Figure A.4: Effect of *CHO* on seconds by block

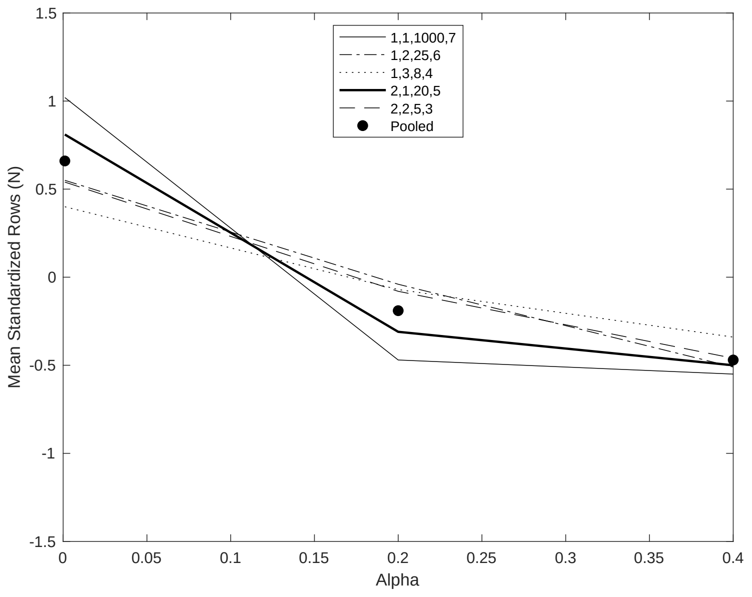


Figure A.5: Effect of α on rows by block

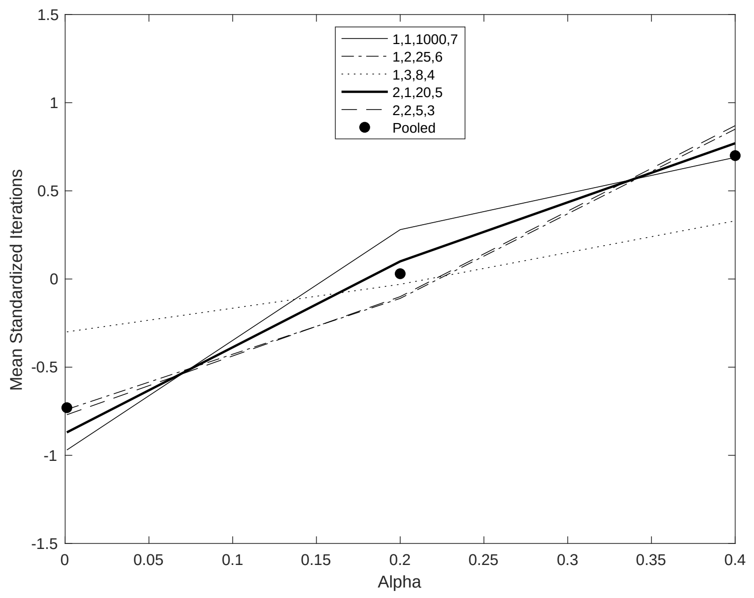


Figure A.6: Effect of α on iterations by block

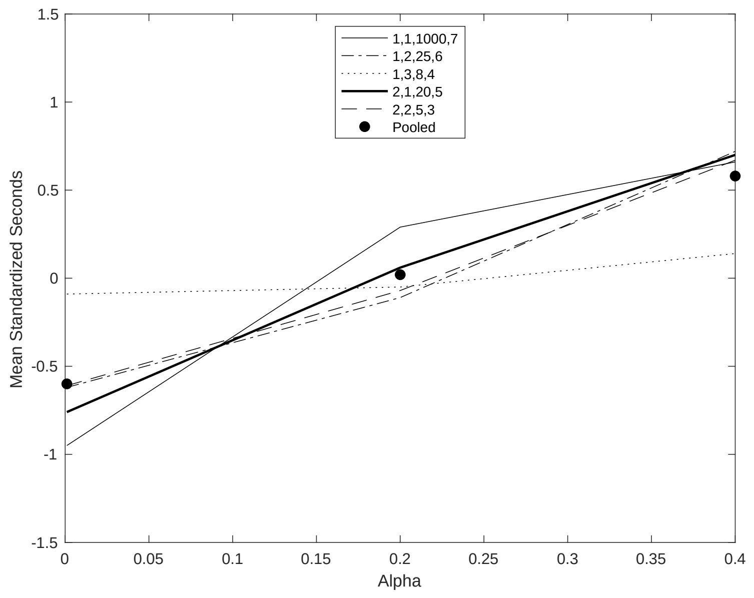


Figure A.7: Effect of α on seconds by block

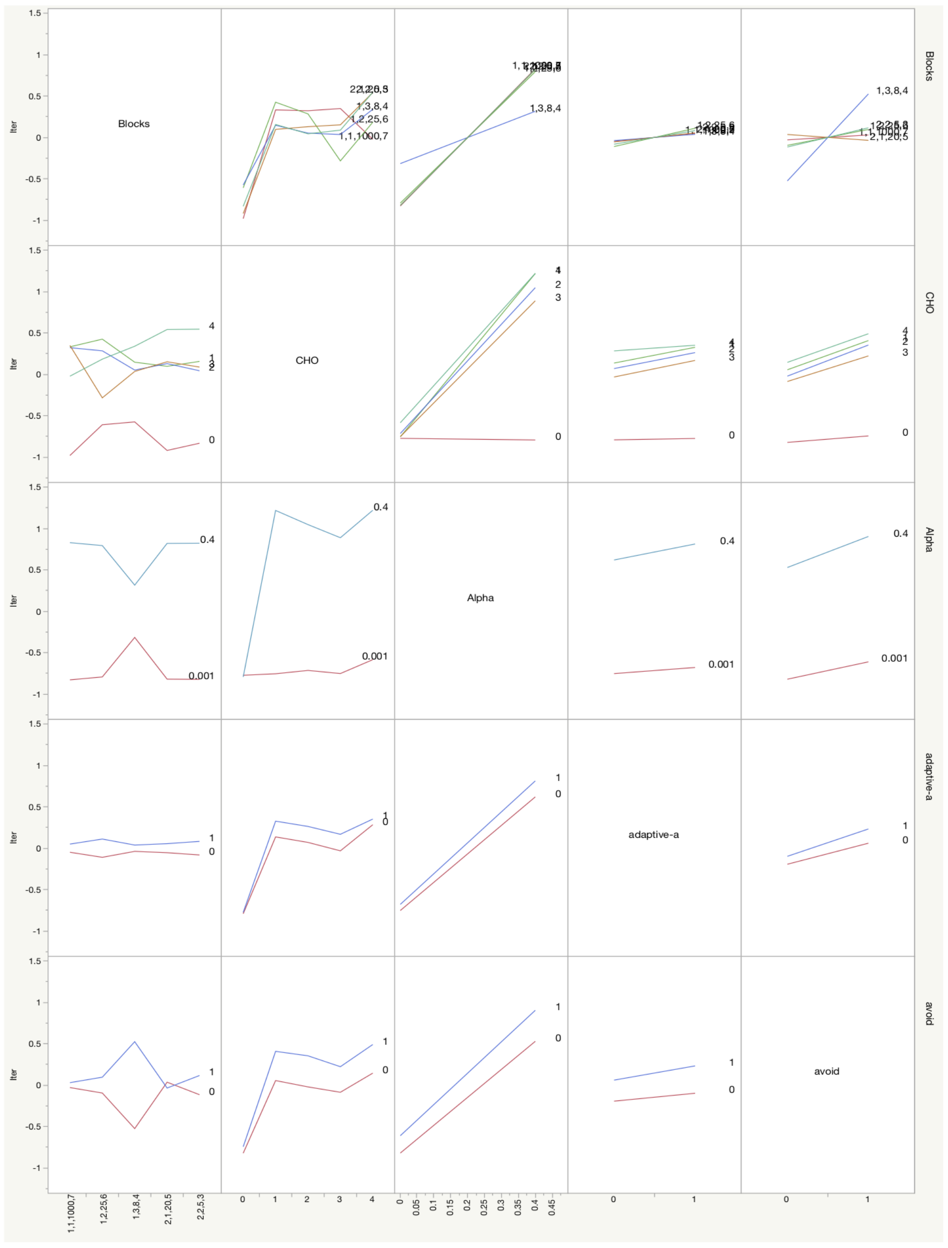


Figure A.8: JMP Interaction Profiler for iterations

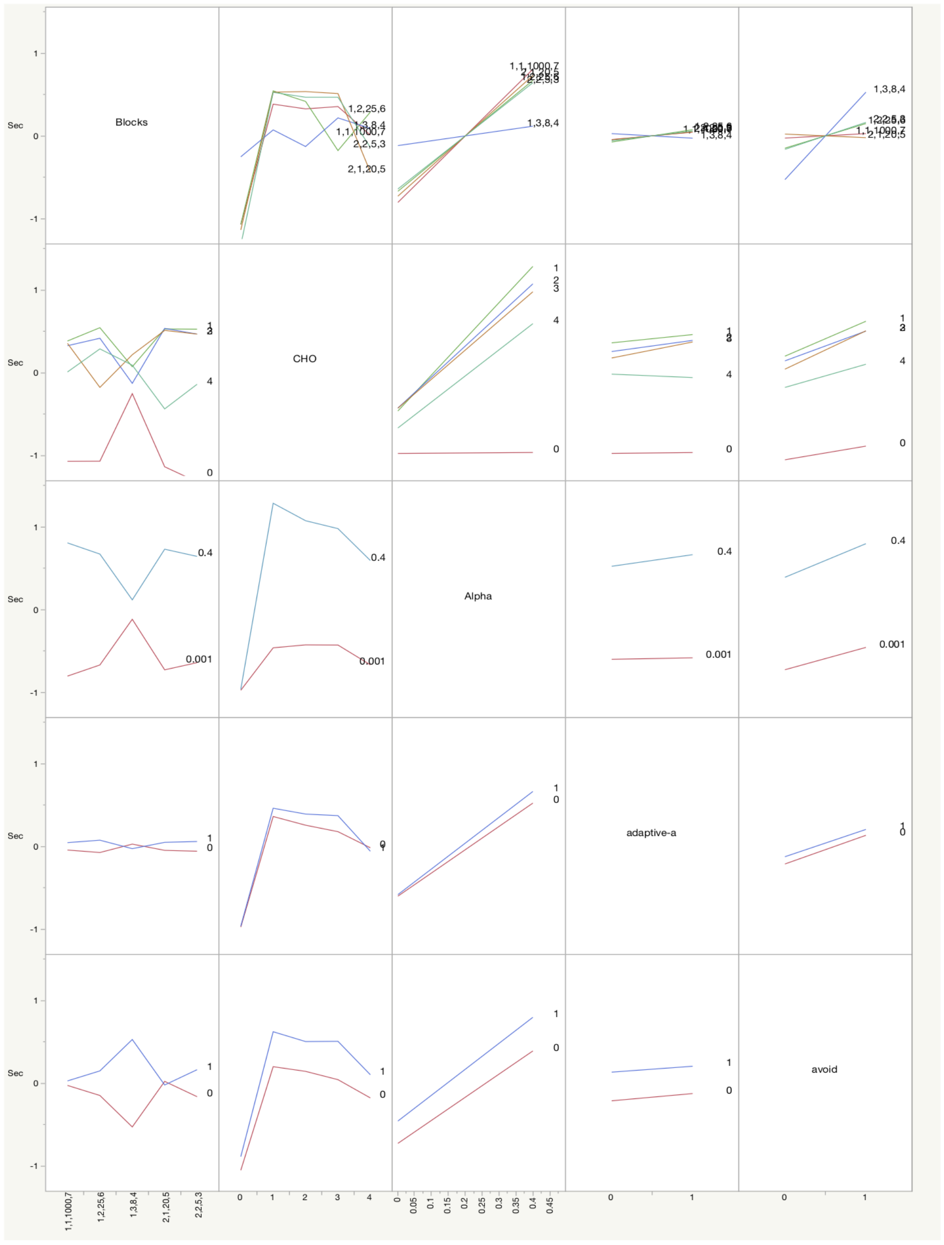


Figure A.9: JMP Interaction Profiler for seconds