6-2011

kb-anonymity: A model for anonymized behavior-preserving test and debugging data

Aditya BUDI
Singapore Management University

David LO
Singapore Management University, davidlo@smu.edu.sg

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

Lucia Lucia
Singapore Management University, lucia.2009@smu.edu.sg

PLDI 11: Proceedings of the 2011 ACM Conference on Programming Language Design and Implementation, San Jose, CA, June 4-8, 2011, Pages 447-457.
http://doi.org/10.1145/1993316.1993551

kb-Anonymity: A Model for Anonymized Behavior-Preserving Test and Debugging Data

Aditya Budi, David Lo, Lingxiao Jiang, and Lucia
School of Information Systems, Singapore Management University
{adityabudi, davidlo, lxjiang, lucia.2009}@smu.edu.sg

Abstract

It is often very expensive and practically infeasible to generate test cases that can exercise all possible program states in a program. This is especially true for a medium or large industrial system. In practice, industrial clients of the system often have a set of input data collected either before the system is built or after the deployment of a previous version of the system. Such data are highly valuable as they represent the operations that matter in a client's daily business and may be used to extensively test the system. However, such data often carries sensitive information and cannot be released to third-party development houses. For example, a healthcare provider may have a set of patient records that are strictly confidential and cannot be used by any third party. Simply masking sensitive values alone may not be sufficient, as the correlation among fields in the data can reveal the masked information. Also, masked data may exhibit different behavior in the system and become less useful than the original data for testing and debugging.

For the purpose of releasing private data for testing and debugging, this paper proposes the kb-anonymity model, which combines the k-anonymity model commonly used in the data mining and database areas with the concept of program behavior preservation. Like k-anonymity, kb-anonymity replaces some information in the original data to ensure privacy preservation so that the replaced data can be released to third-party developers. Unlike k-anonymity, kb-anonymity ensures that the replaced data exhibits the same kind of program behavior exhibited by the original data so that the replaced data may still be useful for the purposes of testing and debugging. We also provide a concrete version of the model under three particular configurations and have successfully applied our prototype implementation to three open source programs, demonstrating the utility and scalability of our prototype.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Symbolic Execution/Testing tools; H.2.8 [*Database Applications*]: Data Mining; K.4.1 [*Public Policy Issues*]: Privacy

General Terms Algorithms, Experimentation, Reliability, Security

Keywords k-anonymity, behavior preservation, privacy preservation, third-party testing and debugging, symbolic execution

1. Introduction

It is common for companies in healthcare, banking, and other industries to employ third-party software houses to develop software systems for their day-to-day business. These software systems are used to manage and process various datasets containing person-specific information. Building perfect software for these industries in the first attempt is hard; extensive testing and debugging are needed to detect and fix bugs.

From a software developer's point of view, real data that would be processed by the software system in the field may contain more *relevant* test cases and *rare* cases that are bug-revealing. Being able to use such real data can be very helpful for testing and debugging.

A data owner, on the other hand, wants high-quality software systems for their daily business, and helping developers (*e.g.*, providing real data for testing) aligns with their interest. However, much of the data is sensitive or confidential. Such data cannot be released to third parties without proper *anonymization* or *desensitization*. For example, data in a hospital could contain the list of diseases each patient has; data in a bank could contain all transaction records of a client. Sending these datasets directly to third-party software developers could pose security risks and privacy concerns. Even though legal mechanisms, such as non-disclosure agreements (NDAs), can be applied to protect the data from leaking further, it can be very costly for the data owners to recover from any damage caused by violations of NDAs. It would be much safer to avoid unnecessary releases of sensitive or confidential data in the first place.

One solution for protecting privacy is to mask away (*i.e.*, replace, or remove) *identifiers* in all data points that can be used to (uniquely) identify an individual, such as *names* and *full addresses*, before releasing the data to a third party. However, due to data sparsity, a *quasi-identifier*, which is *a set* of fields that can uniquely identify a data point, may still exist in the dataset. For example, Golle and Sweeney found that about 63%-87% of the U.S. population can be uniquely identified by gender, 5-digit US ZIP code, and full date of birth [17, 37]. Simply masking away all possible quasi-identifiers would result in much less useful data, and thus a more sophisticated masking scheme is needed.

Another solution is to mask away sensitive or confidential information in all data points so that no actual private information is seen even though each data point uniquely identifies an individual. The main issue of this solution for the purposes of testing and debugging is that particular sensitive information can be important for triggering and thus testing a particular functionality in a software system. For example, in a healthcare application, there might be code implementing a special process for patients with lung cancer. Thus, we cannot simply mask away all possible sensitive information; a more sophisticated masking scheme is needed.

In this work, we propose a model for anonymizing privacy data that addresses the following challenges:

- The resulting dataset *should not leak individual-identifying* information.
- The resulting dataset should still *preserve the utility* of the original dataset for the purposes of *testing and debugging*.

We address these challenges by performing selective data value replacement in the original dataset at a data owner's site. The value replacement will generate a new dataset satisfying the following requirements. Each data point in the new dataset can still be used as a test case but cannot identify any individual in the original dataset. Also, the behavior of a program on a new data point should be the same as the behavior of the program on some original data point. This would allow failures exhibited by the original dataset to be exhibited by the new dataset, or the test coverage achieved by the original dataset to be achieved by the new dataset.

More specifically, our model and its implementation are mainly a combination and extension of two ideas: the k-*anonymity* model from the data mining and database research communities [7, 32, 38] that can provide guidance on choosing data fields to mask, and *concolic execution* [16, 36] that can guide the generation of new test cases based on known ones and make sure the new test cases satisfy certain properties. Merging the two ideas in various configurations enables us to achieve both privacy and program behavior preservation in various degrees. We call our model kb-*anonymity*[1] to highlight that it can preserve *behavior* and satisfy requirements similar to k-anonymity.

Note that another important difference between k-anonymity and kb-anonymity is that the new data points generated by kb-anonymity may not correctly reflect the original data points, as fake values may be introduced and some original data points may be lost. Certain statistics of the original data (*e.g.*, the geographical distribution of all persons contained in the original dataset or the percentage of persons having cancer) can thus be distorted, rendering the new dataset unsuitable for purposes (*e.g.*, data mining and epidemiological studies) other than testing and debugging. However, we believe that maintaining statistics of the original dataset is not necessary for testing and debugging, as long as the new dataset can exhibit the same kinds of behavior as the original dataset.

We present our privacy and program behavior preservation model in Section 3 and 4. We have built a prototype of the model on top of Java Pathfinder (JPF) [39], JFuzz (an extension of JPF) [18], and an approximation algorithm for creating k-anonymized datasets [7]. The main contributions of this work are as follows:

1. We propose a new problem of privacy preservation for testing and debugging.
2. We propose a new model for preserving both privacy and behavior when generating and releasing data for testing and debugging, and analyze various configurations of the model.
3. We propose several algorithms to implement various configurations of the model.
4. We empirically evaluate our solution on several sliced real programs and show the feasibility of our model and implementation in generating useful anonymized test and debugging data.

The outline of this paper is as follows. Section 2 summarizes necessary concepts and definitions related to k-anonymity and program behavior. We present our model and its privacy and behavior preservation properties in Section 3, and describe various configurations of our model in Section 4. Section 5 elaborates our empirical evaluation. We discuss some further considerations and threats to validity in Section 6. Section 7 describes related work. Finally, we conclude with future work in Section 8.

[1] b stands for *behavior*

2. Preliminaries

In this section, we introduce some concepts and definitions relevant to our work.

2.1 On Privacy Preservation

DEFINITION 2.1 (**Data**). *Each* dataset *is a set of* data points*; each data point in a dataset is a tuple t of the same number of fields: $t = \langle f_1, f_2, \ldots, f_n \rangle$. The value of each field is from a domain specific to the field. We use $t[i]$ to denote the value of the i^{th} field in a tuple t, and $t[i_1, \ldots, i_j]$ to denote the sequence of values from the $i_1^{th}, \ldots, i_j^{th}$ fields.*

Given a dataset D, $D[i]$ denotes the set of values from the i^{th} field of all tuples in D, which can also be viewed as a tuple when the values are arranged in an arbitrary order.

DEFINITION 2.2 (**Raw Data**). *A* raw dataset *is a set of* raw tuples*; each raw tuple is a tuple whose fields may contain person-specific values. Releasing raw tuples to any third party may violate the privacy requirements of the data owner.*

EXAMPLE 1. *Consider a raw dataset containing four patient records, each of which has seven fields (*`NID` *means national identification number which is abbreviated to 3 digits here):*

Name	NID	Age	Gender	Address	Doctor	Disease
Bob	254	53	Male	Clementi	Dr. Joe	Cancer
Tom	284	53	Male	Clementi	Dr. Joe	Cancer
Bob	893	37	Male	Jurong	Dr. Anne	Hypertension
Sue	283	34	Female	Jurong	Dr. Jill	Flu

Some fields in the raw dataset (*e.g.*, `NID`) uniquely identify an individual, and thus are referred to as *identifiers*. Some sets of fields can also uniquely identify an individual when used together (*e.g.*, the set of fields {`name`, `age`, `gender`, and `address`} in this example); they are referred to as *quasi-identifiers*. Some other fields are usually not used to identify an individual, but provide information about an individual (*e.g.*, `Disease`); they are referred to as *sensitive fields*.

DEFINITION 2.3 ((**In-**)**distinguishable Tuples**). *For two tuples t_1 and t_2, t_1 is indistinguishable from t_2 if for every identifier or quasi-identifier field f, $t_1[f] = t_2[f]$.*

We cannot release the raw dataset as it is, since each tuple can uniquely identify an individual through either the identifiers or the quasi-identifiers, leaking the patient's privacy. We transform the raw dataset into an *anonymized dataset* by replacing the values of some fields in the raw tuples with some generic values or masking them away with asterisks. Here is an anonymized dataset for Example 1, with which it is impossible to pinpoint a person having a particular disease when the receivers of the anonymized dataset have no other knowledge about the raw dataset. Notice that the values of the identifier field (`NID`) and the quasi-identifier fields (`name`, `age`, `gender`, and `address`) of each patient are equal to those of some other patient.

Name	NID	Age	Gender	Address	Doctor	Disease
Patient	*	53	Male	Clementi	Dr. Joe	Cancer
Patient	*	53	Male	Clementi	Dr. Joe	Cancer
Patient	*	30–39	*	Jurong	Dr. Anne	Hypertension
Patient	*	30–39	*	Jurong	Dr. Jill	Flu

A well-known privacy protection model, k-anonymity [32, 38], provides guidance on choosing data fields to mask:

DEFINITION 2.4 (**k-Anonymity**). *A dataset K is said to satisfy k-anonymity if each tuple in K is indistinguishable from at least k–1 other tuples in K.*

```
 1: void processAPatient (int patient_id, int disease_id, int age) {
 2:   switch ( disease_id ) {
 3:     case 1: // Cancer
 4:       if (age >= 60) {
 5:         if (patient_id <= 1000) // VIP
 6:            treatment("Premium intensive cancer");
 7:         else
 8:            treatment("Intensive cancer");
 9:       } else {
10:         if (patient_id <= 1000)
11:            treatment("Premium standard cancer");
12:         else
13:            treatment("Standard cancer");
14:       }
15:     ...
```

Figure 1. An example to illustrate path conditions

The above example dataset is obviously 2-anonymized with respect to the first five fields. If we also consider `Doctor` as part of the quasi-identifier, only the first two anonymized tuples satisfy 2-anonymity. Without loss of generality, this paper considers the set of *all* fields in a dataset as one quasi-identifier for simpler exposition, unless some fields are explicitly identified as sensitive. Thus, we can simply denote two indistinguishable tuples t_1 and t_2 as $t_1{=}t_2$.

DEFINITION 2.5 (**Value Replacement Function).** *Given a dataset R with tuples having n fields, a function $F : R \rightarrow R_U$, where R_U denotes the set of all possible tuples with n fields, is a value replacement function for R, if $\forall t{\in}R$, $\forall i{\in}\{1,\ldots,n\}$, $F(t)[i]$ is defined or equals $*$.*

The number of value replacements induced by F for R is $\sum_{t\in R}\sum_{i=1}^{n} I(\,F(t)[i]{\neq}t[i]\,)$, where $I(\cdot)$ is an indicator function, which returns 1 if its parameter is true, and 0 otherwise.

DEFINITION 2.6 (**Minimal k-Anonymity).** *Given a raw dataset R and a k-anonymized dataset K, K is said to be* minimal k-anonymized *if there exists a value replacement function from R to K that requires the fewest number of value replacements among all possible replacement functions from R to any k-anonymized dataset.*

Constructing minimal k-anonymized datasets for a given raw dataset has been proved to be NP-complete [7], but there exist algorithms that can approximate minimal solutions in quadratic time [7].

2.2 On Program Behavior

For software testing and debugging, we are also concerned with concepts related to program states, program execution paths, program inputs and outputs, etc.

DEFINITION 2.7 (**Program State).** *A program state corresponds to the complete status of a program at a point at runtime, which can include the values of all program variables, the location of the program counter, the environment on which the program relies (e.g., the file system status), etc. An execution of a program can be viewed as a sequence of program state transitions.*

DEFINITION 2.8 (**Path Condition).** *A path condition for an execution through a control flow path in a program is the conjunction of all the conditionals along the path.*

Example. Consider the program shown in Figure 1. For the input arguments: `patient_id==10`, `disease_id==1`, and `age==65`, the path condition is: `disease_id==1` $\wedge$ `age`$\geqslant$`60` $\wedge$ `patient_id`$\leqslant$`1000`.

PROPERTY 1. *For each program execution, there is only one path condition. Two executions following the same control flow paths have equivalent path conditions, although the executions may follow different sequences of program state transitions.*

Last but not least, software testing and debugging is often concerned with reproducible bug reports and test cases, which requires the concept of *equivalent program behavior*. There exist various definitions for this concept. In this paper, we define equivalence based mainly on path conditions:

DEFINITION 2.9 (**Behavior Equivalence).** *Two program executions are said to have equivalent behavior if their corresponding path conditions are equivalent (i.e., they follow the same control flow paths).*

3. Privacy and Behavior Preservation Model

In this section, we aim to establish a formal privacy preservation model, kb-anonymity, that can guide the construction and evaluation of algorithms and tools that generate datasets suitable to be released to third parties for testing and debugging purposes.

3.1 Objectives

There are two dimensions that we need to consider in this work:

Privacy Preservation. We assume that the raw data points contain person-specific information, and the objective of our model is to ensure that the identity of every person whose information is contained in the raw dataset cannot be revealed in the released dataset.

Behavior Preservation. The other objective of our model is to ensure that the program behavior exhibited by the raw dataset can be reproduced by the released dataset to a certain extent without compromising privacy. How program behavior is defined can vary in different configurations of the model.

The above two goals are inversely related. The more privacy is desired, the more values in the raw dataset may need replacement, and the harder it is to preserve program behavior. Similarly, the more behavior preservation is desired, the harder it is to preserve privacy. In the worst case, for a particular level of behavior preservation, it may not be possible to preserve any privacy. We elaborate on our privacy and behavior model in the following subsections.

3.2 Privacy Preservation

k-anonymity [32, 38], summarized in Section 2, is a well-known privacy protection model that requires every released data point to be indistinguishable from at least k–1 other released data points with respect to all identifier and quasi-identifier fields.

Simply conforming to the k-anonymity model does not fit our needs because of the following issues:

- The k-anonymity model allows duplicated data points in the dataset to be released, which may not be meaningful for testing and debugging due to redundant test cases;
- The k-anonymity model often implies that certain values in raw tuples would be replaced with a more generic value (e.g., replacing `California, USA` with `USA`) or an asterisk (`*`, which means "Don't tell"). This may lead to unusable test cases since testing and debugging often require concrete data values.[2]

[2] A "concrete data value" refers to any value in the domain of program inputs. In the above example, if the program accepts any location as an input, then both `California, USA` and `USA` are concrete, but `*` is not; if the program requires a state name as part of its inputs, then only `California, USA` is concrete.

Our kb-anonymity model addresses the issues by requiring a value replacement function to satisfy the following requirements:

R1: All values in a released dataset are concrete so that each data point can be directly used to execute programs;

R2: All released tuples are distinguishable from each other so as to reduce redundant test cases;

R3: For each raw tuple and its corresponding released tuple generated by applying the value replacement function on the raw tuple, there exist at least k–1 other raw tuples that are mapped by the same function to a tuple indistinguishable from this released tuple.

Intuitively, kb-anonymity aims to provide a similar guarantee as k-anonymity (R3), but avoid generic values (R1) and duplicate data points (R2). However, note that kb-anonymity is different from k-anonymity. k-anonymity considers k indistinguishable tuples *within the released dataset* while kb-anonymity considers k indistinguishable tuples *with respect to replacements of the raw dataset*. If the number of raw tuples is m, the number of the released tuples will be at most $\lfloor m/k \rfloor$, while k-anonymity will still have m tuples. The following Theorem 1 states that the R2 and R3 requirements can be easily achieved by applying k-anonymity and suppressing indistinguishable tuples.

3.2.1 $R2 + R3 \equiv k$-Anonymity Modulo Uniqueness

LEMMA 1. *Given a raw dataset R and its k-anonymized version K, K satisfies R3.*

PROOF. Let the value replacement function used to generate the tuples in K from the tuples in R be $F : R \rightarrow K$. According to the definition of k-anonymity, for each t'_1 in K, there exist at least k–1 tuples in K, $t'_2, \ldots, t'_k$, such that $\forall i \in \{2, \ldots, k\},\ t'_1 = t'_i$. Since F maps each raw tuple in R to at most one tuple in K, there must exist at least k raw tuples $t_1, \ldots, t_k$ such that $\forall i \in \{1, \ldots, k\},\ F(t_i) = t'_i$. By transitivity of indistinguishability, $\forall i \in \{1, \ldots, k\},\ F(t_i) = t'_1$. □

THEOREM 1. *Given a raw dataset R and its k-anonymized version K, construct a new dataset K' as follows: for each group of indistinguishable tuples in K, add exactly one tuple from the group into K'. Then, K' satisfies R2 and R3.*

PROOF. *i*) K' trivially satisfies R2 due to the removal of indistinguishable tuples. *ii*) For each tuple t' in K', there must exist at least k tuples in K that are indistinguishable from t' (otherwise, t' would not appear in K'); thus R3 is true based on the same reasoning as Lemma 1. □

Theorem 1 provides us a base to build a dataset satisfying R2–R3. The other issue is to replace certain values and asterisks in K' to satisfy R1 while maintaining R2 and R3. Depending on which values to replace and how to perform replacement, we have various choices corresponding to various levels of privacy preservation. The following subsections introduce two options in our model.

3.2.2 No Field Repeat

This subsection proposes the `no field repeat` option for replacing values from raw data points. Its intuition is to ensure every value in the released dataset has not appeared in any raw data point. For example, if `age==30` appears in the raw dataset, `30` would not be used for the `age` field in the released dataset.

DEFINITION 3.1 (**No Field Repeat Option.**) *Given a raw dataset R, a released version X of R satisfies* the `no field repeat` option *if X satisfies the following: i) X satisfies R1–R3; ii) $\forall\ i \in \{1, \ldots, n\}, \forall\, t \in R, \forall\, t' \in X,\ t[i] \neq t'[i]$.*

In some cases, `no field repeat` is impossible. For example, when both `male` and `female` have appeared in the `gender` field in the raw dataset, we may have to use one of two values (unless the program could handle a more generic gender value, such as `unknown`). Thus, we also need a less restrictive option.

3.2.3 No Tuple Repeat

This subsection proposes the `no tuple repeat` option. This option ensures that every released tuple is distinguishable from every raw tuple, but allows them to share some field values.

DEFINITION 3.2 (**No Tuple Repeat Option.**) *Given a raw dataset R, a released version X of R satisfies* the `no tuple repeat` option *if X satisfies the following: i) X satisfies R1–R3; ii) $\forall\ t \in R$, $\forall\, t' \in X, t \neq t'$.*

Note that the `no field repeat` option subsumes the `no tuple repeat` option, meaning that a released dataset satisfying `no field repeat` also satisfies `no tuple repeat`.

In summary, we have the following four levels of privacy preservation: *(i)* `None` which imposes no restriction on released tuples, *(ii)* `Standard` k`-Anonymity` modulo uniqueness, *(iii)* `No Tuple Repeat`, and *(iv)* `No Field Repeat`.

3.3 Behavior Preservation

The second objective of our kb-anonymity model is to ensure the following:

R4: For each released tuple b and each raw tuple t that is mapped to b, b and t must exhibit the same behavior when run on the subject program.

As mentioned in Section 2, various definitions of program behavior and equivalence exist and affect our model that is mainly for the purposes of software testing and debugging. The personnel and tools performing testing and debugging often consider program runs with reusable or reconstructible program execution paths, program inputs and outputs, and program states. Thus, we consider the following four levels of behavior equivalence in this paper: `None`, `Same Path`, `Same Path with Input Restrictions`, and `Same Program States`.

The lowest level, `none`, is used as a baseline that allows arbitrary program behavior to be exhibited by the released dataset and provides no guarantee on behavior preservation.

The second level, `same path`, requires each released data point to follow the same execution path as the path followed by the raw data point mapped to it:

R4-1: A released tuple b is *path preserving* for a raw tuple t mapped to it if b and t follow *the same execution path* in subject programs.

If every raw data point could be mapped to a released data point, the resulting dataset would have the same path coverage (which is a commonly used test sufficiency criterion) as the raw dataset.

The third level, `same path with input restrictions`, aims to consider more program behaviors beyond execution paths, such as particular input values. This is useful for cases when two program runs have different observable effects even if they follow the same path. For example, the following Java function, which accepts the original bank account balance (`orig`) and the withdrawal (`amt`), and returns the final balance, has a functional error when a negative `amt` is fed to it:

```
double reduceBalance(double orig, double amt) {
   return orig - amt;
}
```

The error can be observed (*e.g.*, by an auditor) if the raw dataset contains the input $\langle 5.0, -2.0\rangle$. If the released dataset only contains $\langle 5.0, 2.0\rangle$, the error cannot be observed even though the same path is executed.

Thus, the third level introduces additional restrictions on program inputs, aiming to preserve more program behaviors. The restrictions may be expressed in terms of various constraints (*e.g.*, `amt==-2.0` for the above example). In this paper, we consider only one type of input restrictions: some (arbitrary) fields of the released data points shall preserve their original values. We refer to this as *input preservation*. The framework allows future extensions including the consideration of constraints provided by users and expressed as general first-order predicates. However, this paper utilizes a minimal k-anonymization algorithm to decide the input preservation constraints (*cf.* Section 4.4) without the need for user-specified constraints. Even though the minimal k-anonymization algorithm cannot guarantee to retain all desired values (*e.g.*, –2.0 in the above example), we still have the following benefits: (1) the total number of replaced values in the released dataset can be minimized so that more program behaviors may hopefully be preserved; (2) the need for user-specified constraints is avoided so that there is no risk of leaking privacy data by incorrectly specified constraints.

R4-2: A released tuple b is *path and input preserving* for a raw tuple t if b is path preserving for t and it satisfies the given input constraints.

The fourth level, `same program states`, is an extreme level used as a reference point where the sequence of program states exhibited by each release tuple and the raw tuple mapped to it should be the same.

The following subsection discusses the various combinations of privacy and behavior preservation levels in our model.

3.4 Combining Privacy and Behavior Preservation

Since we have four levels of privacy preservation and four levels of behavior preservation, privacy and behavior presevation could be combined in 16 possible ways, as shown in Figure 2.

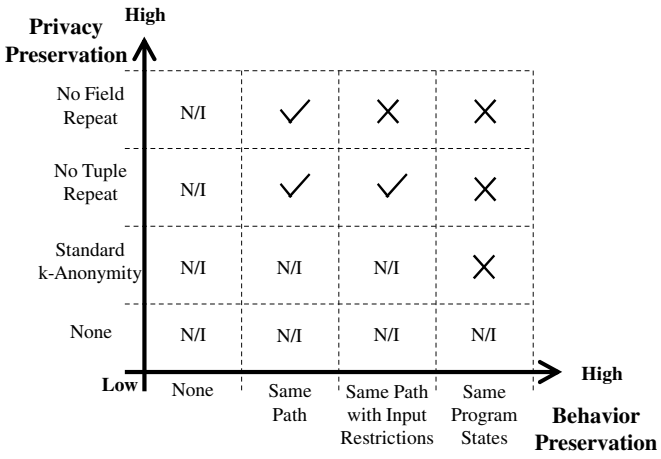

Figure 2. Privacy vs. behavior preservation

Some combinations marked with "X" are impossible to achieve. Many others marked with "N/I" are not interesting for the purpose of preserving privacy and behavior. Those marked with a tick are of interest and possible to achieve. The following paragraphs describe the combinations in more details.[3]

PROPERTY 2. *The privacy preservation level* `none` *is not interesting.*

PROPERTY 3. `Standard` k`-anonymity` *alone is not interesting since it cannot satisfy R1 required by our* kb*-anonymity model.*

PROPERTY 4. *The behavior preservation level* `none` *is not interesting for the purposes of software testing and debugging.*

PROPERTY 5. *It is impossible to pair* `same program states` *with any privacy preservation level (except* `none`*).*

EXPLANATION: Since program states include input values to a program, `same program states` implies the same input values. Thus, the released tuple must be the same as a raw tuple, which is only allowed when the raw dataset is already k-anonymized or when there is no desire for privacy. □

PROPERTY 6. *It is impossible to pair* `same path with input restrictions` *with* `no field repeat`.

EXPLANATION: In this paper, we consider input restrictions to be constraints that require some (arbitrary) input fields to have the same values as found in raw tuples. Thus, by definition, datasets complying with `same path with input restrictions` cannot also conform to `no field repeat`. □

With the combination of k-anonymity and behavior preservation, we have the following property stated in Theorem 2. This property can be used to efficiently check whether a dataset can be kb-anonymized (*cf.* Section 4.1).

THEOREM 2. *For each behavior exhibited by a released tuple satisfying R1–R4, there must exist at least* k *raw tuples that exhibit the same behavior.*

PROOF. *The requirement R1 ensures that a released tuple can be used to run a program, as all raw tuples can. R3 ensures that at least* k *raw tuples map to each released tuple. R4 ensures that a raw tuple only maps to a released tuple with the same behavior. Thus, it must be the case that for each released tuple, there are at least* k *raw tuples that map to it and they all have the same behavior.*[4] □

In a nutshell, our kb-anonymity model requires R1–R4 altogether, and there are three interesting configurations:

- (`same path`, `no field repeat`)
- (`same path`, `no tuple repeat`)
- (`same path with input restriction`, `no tuple-repeat`)

As shorthand notations, we refer to these as `P-F`, `P-T`, and `I-T`, respectively. The next section presents our realization of these configurations. In cases when some raw data points cannot be mapped to a released tuple, we simply output error messages.

4. Model Realization

In this section, we describe the algorithms and tools used to realized the three configurations of our kb-anonymity model: `P-T`, `P-F`, and `I-T`. The overall framework of our realization is illustrated in Figure 3.

[3] We do not consider cases where the raw dataset satisfies k-anonymity. These uninteresting cases happen in situations, such as when k=1, or when there is no identifier or quasi-identifier.

[4] R2 is unnecessary for proving Theorem 2, but it eliminates indistinguishable tuples and thus may help to save the cost of testing and debugging.

4.1 Overall Framework

All three configurations require path preservation, which means we need to collect the execution paths of all raw tuples. The *Program Execution* module takes raw tuples and executes a program with each of the tuples, then it collects the path conditions exercised by each execution. We assume that each tuple is processed by the program independently; we do not consider any dependency among program states or path conditions that may be introduced by multiple runs of the program with different tuples. Theoretically, two executions having the same path condition follow the same execution path. Thus, this module can group raw tuples based on the equivalence of their path conditions. At the end of this step, groups of size less than k are discarded due to Theorem 2.

For each of the groups left, the *k-Anonymization* module may replace some field values with asterisks and make sure that each tuple is indistinguishable from at least k–1 other tuples in the group. Then, it outputs a set of unique tuples a la Theorem 1.

Next, the *Constraint Generation* module takes the set of unique tuples from the *k-Anonymization* module and the path conditions for every tuple associated with the unique tuple. Various constraints are then generated for each of the unique tuple according to each of the three configurations.

Last, the *Constraint Solver* takes the constraints for each of the unique tuple and tries to generate one new tuple satisfying the constraints. If the solver finds a satisfying tuple, this tuple will be part of the released dataset. When the solver cannot find a satisfying tuple, our framework simply outputs error messages.

Theorems 3, 4, and 5 state that the datesets outputted by the algorithms satisfy kb-anonymity (R1–R4) and can be released for software testing and debugging.

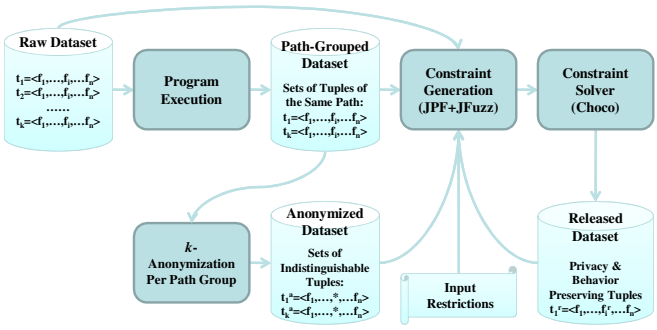

Figure 3. Overall framework of our model realization

The overall parameterized Algorithm 1 realizes the three configurations. At Lines 3–4, we obtain the path condition for every raw tuple. At Lines 2–7, we group the raw tuples into different buckets based on their path conditions. At Lines 8–11, we remove buckets of size less than k because of Theorem 2. In an extreme case when all buckets are of size less than k, the released dataset will be empty. The remaining buckets are those that require value replacement satisfying the given configuration. Then, Lines 14 and 15 run a k-anonymization algorithm for every bucket and generate the anonymized version a la Theorem 1, and Lines 16–21 store the k-anonymized buckets that satisfy certain conditions. Note that for configuration `I-T`, we require Line 13 to invoke the approximate minimal k-anonymization, and only store an anonymized tuple if it contains some concrete values. We can simply discard a tuple if it contains no concrete value (Lines 17–20) since it means no values in the corresponding raw tuples can remain and the `I-T` configuration cannot be satisfied for these tuples. Then, for each bucket that can be k-anonymized (Line 22), Lines 23–29 construct sufficiently strong constraints for the given configuration, feed the conjunction

Algorithm 1 A Realization of kb-Anonymity

Input: R: Raw dataset
k: Level of anonymization
P: A subject program
O: Configuration option: `P-T`, `P-F`, or `I-T`.
Output: R': Anonymized dataset for release

1: $R' \leftarrow \emptyset$
// The *Program Execution* module
2: $PCBuckets \leftarrow \emptyset$, which groups tuples based on path conditions
3: For each t in R
4: Execute P with t and collect the path condition pc
5: If $PCBuckets$ does not contains pc
6: $PCBuckets \leftarrow PCBuckets \cup \{\langle pc, \emptyset\rangle\}$
7: $PCBuckets[pc] \leftarrow PCBuckets[pc] \cup t$
8: For each $Bucket = \langle pc, B\rangle \in PCBuckets$
9: If $|B| < k$
10: $PCBuckets \leftarrow PCBuckets - \{Bucket\}$
11: Output "`Error: unsatisfiable case`" and **continue**
// The *k-Anonymization* module
12: $A \leftarrow \emptyset$, holding intermediate k-anonymized datasets
13: For each $\langle pc, B\rangle \in PCBuckets$
14: Invoke a k-anonymization algorithm on B,
15: and get its result B' with no duplicates a la Theorem 1
16: For each tuple $b' \in B'$
17: If $O =$ `I-T`
18: If $|b'| <= 1$ or no field in b' contain concrete values
19: Output "`Error: unsatisfiable case`"
20: **continue**
21: $A \leftarrow A \cup \{\langle b', pc, B\rangle\}$
// The *Constraint Generation* module
22: For each $\langle b', pc, B\rangle \in A$
// Construct constraints for various configurations
23: If $O =$ `P-F`
24: $S \leftarrow$ Invoke Algorithm 2 on R
25: Else If $O =$ `P-T`
26: $S \leftarrow$ Invoke Algorithm 3 on B
27: Else If $O =$ `I-T`
28: $S \leftarrow$ Invoke Algorithm 4 on $\langle B, b'\rangle$
29: Else Output "`Error: unimplemented option`" and **continue**
30: $S \leftarrow$ Conjunction of S and pc
// The *Constraint Solver* module
31: Invoke a constraint solver on S, and get its result r
32: If r is not an error
33: $R' \leftarrow R' \cup \{r\}$
34: **Return** R'

of the constructed constraints and the path condition for this bucket (Line 30) into a constraint solver (Line 31), and add a valid solution from the solver into the resulting dataset (Lines 32–33). We finally output the dataset for release (Line 34).

We have implemented a prototype of this framework. This prototype uses Java PathFinder (JPF) [39] and jFuzz [18], one of JPF's extensions that supports the combination of concrete and symbolic executions of Java programs, to emulate executions of our subject programs and collect path conditions. We rely on JPF's internal canonical representation of path conditions and use string comparison to check for equivalent path conditions. At the same time, we have implemented an approximate algorithm that can construct k-anonymized datasets [7]. Also, we have extended JPF to allow manipulation of the collected path conditions and generation of the desired constraints. Finally, we utilize the constraint solver used in JPF—Choco [1]—to generate new tuples from the constraints. For now, we only handle constraints with integers and real numbers. In the future, with the coming support for string constraints in JPF [2], we hope our framework can handle programs with strings easily.

The following subsections describe more details and properties for each of the three configurations in our model.

4.2 Same Path, No Field Repeat (P-F)

To realize this configuration option, we perform Algorithm 2 within Algorithm 1 at Line 24.

Algorithm 2 Generation of Constraints for P-F

Input: T: A set of (raw) tuples
Output: S: A (conjunctive) set of constraints for P-F

For each field i
  Construct its constraint variable v_i
  For each $t \in T$
    $S \leftarrow S \cup \{v_i \neq t[i]\}$
Return S

The realization of this configuration basically generates new data values different from all of the original ones (via Line 24 in Algorithm 1) to ensure no violation of privacy, and ensure preservation of program behavior (via Line 30 in Algorithm 1). Algorithm 2 is invoked on the whole raw dataset to ensure `no field repeat` for all tuples, which of course imposes more constraints and makes it harder for the constraint solver to find a tuple. Alternatively, we could relax the `no field repeat` requirement and invoke Algorithm 2 on B instead (Line 24) without compromising privacy.

Note that there is no particular input restrictions or minimal anonymization requirements in this configuration. It is not necessary to run minimal k-anonymization algorithms (Line 14). We just need to ensure that there are at least k tuples in a bucket B having the same path (Lines 8–11) via Theorem 2 and simply treat all tuples in B as one equivalence partition in its anonymized version. Doing so may only produce one new tuple for each B, but help to speed up the anonymization process. Also, it is not necessary to remove duplicated, intermediate tuples (Line 15), although doing so may help to prevent redundant operations at Lines 22–33 and make the control flows for different configuration options simpler.

THEOREM 3. *An output dataset from Algorithm 1 along with Algorithm 2 satisfies kb-anonymity (R1–R4).*

Proof Sketch: Each dataset generated at Line 14 satisfies R2 and R3 according to Theorem 1; Each tuple generated at Line 31 is obviously path-preserving (via Line 30) and satisfies R4 and R1 (all values concrete). Also, the constraints imposed by P-F, P-T, or I-T ensure no tuples from the raw dataset will appear at Line 31. Thus, the resulting dataset R' at Line 34 satisfies kb-anonymity. □

4.3 Same Path, No Tuple Repeat (P-T)

To realize this configuration, we perform Algorithm 3 within Algorithm 1. Similar to the realization for P-F, P-T generates new data values and tuples to ensure privacy, although its privacy requirement is weaker than that of P-F. Some data values from the raw dataset may remain even though there are no tuples in the intersection between the raw and the released datasets. In particular, Algorithm 3 in this paper only considers changing the first field of all tuples to satisfy `no tuple repeat`. Alternatively, we could choose a random field of all tuples or some random fields from each tuple to strengthen privacy protection.

Also similar to P-F, it is not necessary to run minimal k-anonymization algorithms (Line 14), as there is no need to satisfy any input constraint or maximize the number of released tuples.

THEOREM 4. *An output dataset from Algorithm 1 along with Algorithm 3 satisfies kb-anonymity.*

Proof Sketch: Similar to that of Theorem 3. □

Algorithm 3 Generation of Constraints for P-T

Input: T: A set of (raw) tuples
Output: S: A (conjunctive) set of constraints for P-T

For the first field in T
  Construct its constraint variable v_1
For first field in each t in T
  $S \leftarrow S \cup \{v_1 \neq t[1]\}$
Return S

4.4 Same Path & Some Input, No Tuple Repeat (I-T)

To realize this configuration, we perform Algorithm 4 within Algorithm 1. Different from the previous realizations, we need to ensure that some—arbitrary one or a few—raw input values are preserved. To do this, we need to run a minimal k-anonymization algorithm at Line 14 in Algorithm 1 to decide the minimal number of field values that need to be masked away to realize k-anonymity. By doing this, we can retain the maximal number of raw concrete values in released tuples (*i.e.*, input preservation) so as to preserve more program behaviors. The process has been proven to be NP-complete, thus we run a variant of k-anonymization [7] to achieve approximate results. After applying the k-anonymization algorithm, we could simply output an error message (Lines 17–20) for tuples for which all fields need to be masked away (*i.e.*, no concrete values can be preserved).

Algorithm 4 Generation of Constraints for I-T

Input: T: A set of (raw) tuples
  b: A tuple with generic values or '*'
Output: S: A (conjunctive) set of constraints for I-T

// The If-Else ensures No Tuple Repeat
If b contains no generic values or '*'
  $S \leftarrow$ Invoke Algorithm 3 on T
  $i \leftarrow 1$
Else
  For the first field i in b containing a generic value or '*'
    Construct its constraint variable v_i
  For each t in T
    $S \leftarrow S \cup \{v_i \neq t[i]\}$
// The following helps to ensure some fields maintain their values
For each field j in b containing a concrete value c and $j \neq i$
  Construct its constraint variable v_j
  $S \leftarrow S \cup \{v_j = b[j]\}$
Return S

THEOREM 5. *An output dataset from Algorithm 1 along with Algorithm 4 satisfies kb-anonymity.*

Proof Sketch: Similar to that of Theorem 3. □

5. Empirical Evaluation

We evaluate the capability and scalability of the realization of our kb-anonymity model with three sample programs: OpenHospital [4], iTrust [3], and PDManager [5]. All experiments are performed on an Intel Xeon server with a 2.53GHz quad-core E5540 CPU and 24 GiB of RAM running 64-bit Windows Server 2008 R2 Standard. Algorithm 1 is implemented in Visual C#.Net, while Algorithms 2, 3 and 4 are implemented in Java within jFuzz [18].

5.1 OpenHospital

OpenHospital is an open source hospital management system. It provides various functionalities including managing patient records, pregnancy management, disease information, drug control, pharmacy management, etc. All patient records are stored in a

backend database. We convert a part of the program into an integer program that reads inputs from a file, as our current implementation based on jFuzz does not handle string constraints or database queries.

The part that we convert, denoted as OH_c, is comprised of three Java classes which validate a patient record before storing to database. It validates private information of a patient, including first name, last name, age, gender, address, city, number of siblings, telephone number, birth date, blood type, mother's name, mother's deceased status, father's name, father's deceased status, insurance status, and whether parents live together. Many of the input fields are of string type. We then manually convert them into integers based on their value domains and change corresponding string operations into integer operations.

Even though the program is not big, it demonstrates that our prototype implementation is capable of anonymizing data for testing and debugging, for a large set of inputs. The following example shows sample output for a given set of raw tuples under the `P-T` configuration.

Example. We randomly create synthetic tuples as a raw dataset and run our tool on OH_c with this set. Table 1 shows the raw tuples as well as the results of our tool using `P-T` configuration for $k = 2$. There is an error message for the fifth tuple as it could not satisfy our kb-anonymity requirements. This tuple should not then be released to third-party developers.

No	Raw Data Point	Released Tuple
1	⟨90207, 10125, 2, -1, 16261, 22549, 69883, 914, 8201, -2, 68353, -1, -53, -1, -1, -2⟩	⟨-9999, 10000, 0, -10000, 16261, 22549, 69883, 914, 8201, -2, 68353, -1, -53, -1, -1, -2⟩
2	⟨19892, 16536, 78, 1, 36688, 88797, 172, 7519, 50896, -1, 44500, 1, 7452, -2, -1, 1⟩	
3	⟨35778, 21908, 89, -1, 89965, 41493, 35861, 50182, 79181, 1, 30668, -1, 34926, -2, -1, 1 ⟩	
4	⟨9543, 23693, 48, 1, 18133, 75043, -173, 38100, 14912, 1, 69504, 0, 14969, -1, -2, 1⟩	
5	⟨42164, 40607, -6, 1, 46920, 21328, 15089, 42147, 81975, 1, 24382, -2, -252, -2, -1, -1⟩	Error Message

Table 1. Sample raw tuples and the output tuples after anonymization for OpenHospital. The integers could be mapped to real values. For example, the 3rd field directly maps to age; The 10th field in the tuples corresponds to blood types: 0 means type O, 1 means A, 2 means B, 3 means AB, and negative numbers mean unknown.

Scalability Evaluation. The computational complexities of k-anonymization algorithms and constraint generation and solving are potentially exponential to the number of raw tuples. To investigate the scalability of our proposed approach, we stress test our tool by increasing the number of raw tuples to be anonymized. We experiment with 2,000, 4,000, 6,000, 8,000, and 10,000 tuples, and evaluate the runtime of our tool using the `P-T` and `P-F` configurations for k set to 2, and `I-T` configurations for k set to 2 and 5. We plot the results in Figure 4. Figure 4(a) shows that the runtime per tuple remains practically constant when we increase the number of tuples from 2,000 to 10,000 for all configurations. This implies that in practice the runtime increases linearly with the number of tuples to be processed. Figures 4(a) & (b) show that the `I-T` configuration is slower than the `P-T` and `P-F` configurations. This is because we run a *minimal* k-anonymization algorithm *o*nly for `I-T` configuration (see Section 4) which can take quadratic time. Furthermore, almost all raw tuples were successfully anonymized. In our experiments, at most two tuples out of the thousands of raw tuples failed to be anonymized. This occurred when using `I-T` configuration with k set to 5, while the other configurations were only unable to anonymize at most one tuple.

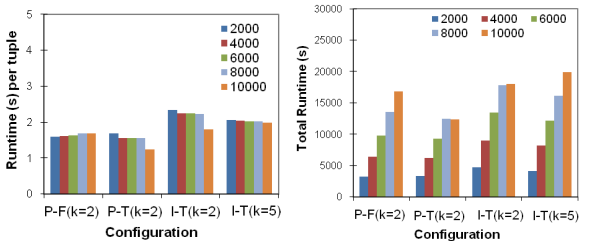

Figure 4. OH_c runtime: per tuple (left) & all tuples (right)

5.2 iTrust

iTrust is an open source medical application that enables patients to maintain their medical history, records, and communications with their doctors. All patient records are also stored in a backend database. Similar to OpenHospital, we convert a part of the program into an integer program that reads input from a file.

The part that we convert, denoted as $iTrust_c$, is comprised of ten Java classes that validate the insurance record of a patient. It involves private and sensitive information including: 1) first name, last name, email, address, city, state, postal code, insurance number, credit card type, credit card number, and telephone number of the patient, 2) name, address, city, state, and postal code of the insurance company, and 3) name and telephone number of an emergency contact person.

Example. Table 2 shows a sample output from our tool for a set of synthetic integer tuples under the `P-F` configuration and $k = 2$.

No	Raw Data Point	Released Tuple
1	⟨70, 549, 288, 499, 490, 246, 15, 110, 137, 519, 840, 91, 128, 31, 3, 28, 466, 113⟩	⟨ 0, 200, 100, 200, 200, 201, 10, 40, 40, 200, 200, 0, 6, 10, 0, 10, 200, 40⟩
2	⟨508, 147, 195, 491, 277, 224, 32, 48, 210, 275, 147, 97, 119, 11, 3, 36, 314, 160 ⟩	
3	⟨754, 418, 72, 779, 437, 69, 13, 91, 94, 567, 710, 203, 104, 34, 1, 20, 848, 79⟩	
4	⟨585, 727, 295, 662, 304, 125, 26, 49, 83, 398, 313, 10, 77, 26, 4, 50, 338, 62⟩	
5	⟨141, 280, 68, 249, 267, 149, 11, 67, 139, 748, 669, 173, 105, 14, 2, 24, 739, 146⟩	

Table 2. Sample raw tuples and the corresponding output after anonymization for iTrust

Scalability Evaluation. Similar to OH_c, we stress test our tool with various numbers of raw tuples, *i.e.*, 2,000, 4,000, 6,000, 8,000 and 10,000, and evaluate the runtime for `P-T` and `P-F` configurations for k set to 2, and `I-T` configurations for k set to 2 and 5. The performance results shown in Figure 5 are similar to those of OH_c. For iTrust, for each configuration and number of raw tuples, all raw tuples were successfully anonymized.

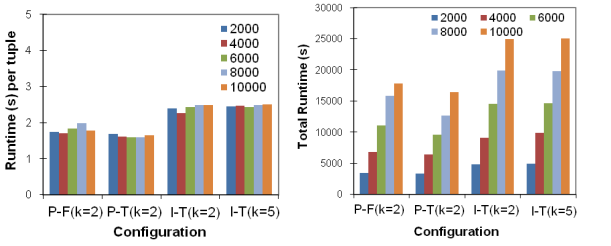

Figure 5. IT_c runtime: per tuple (left) & all tuples (right)

5.3 PDManager

PDManager is an open source insurance agent management system that provides features to manage clients, contracts, and commis-

sions. Similar to OpenHospital and iTrust, we also convert a part of the program into an integer program that reads input from a file.

The part that we convert, denoted as PDM_c, is comprised of eight Java classes that validate the record of an insurance client of an agent. PDM_c takes in 13 input fields representing insurance's name, agent's name, as well as the first name, surname, gender, address, city, state, postal code, telephone number, fax, mobile number, and email of each client.

Example. Table 3 shows sample output of our tool for a set of synthetic integer tuples under the `I-T` configuration with $k = 2$. Note that the 8^{th} fields of the tuples are unchanged.

No	Raw Data Point	Released Tuple
1	⟨1003, 7865, 9479, 1956, 8223, 203, 53, 2, 133, 4707, 1124, 37, 1570 ⟩	⟨ -9999, 10000, 1, -10000, 1, -59, -38, 2, 97, 3505, 600, 289, 809 ⟩
2	⟨1461, 9877, 1786, 4948, 5486, -59, -38, 2, 97, 3505, 600, 289, 809 ⟩	
3	⟨1305, 4352, 994, 6664, 2163, -97, 129, 2, 130, 7458, 9671, -51, 288 ⟩	
4	⟨4629, 1672, 8447, 586, 1072, 112, -42, 1, 93, 3426, 8158, 74, 6900 ⟩	⟨ -9999, 10000, 1, -10000, 1, 112, -42, 1, 93, 3426, 8158, 74, 6900 ⟩
5	⟨7274, 5952, 3798, 3813, 8113, 488, 104, 1, 107, 6176, 7896, -62, 6186 ⟩	

Table 3. Sample raw tuples and the output tuples for PDManager

Scalability Evaluation. Similar to OH_c and $iTrust_c$, we stress test our tool with various numbers of raw tuples and evaluate the runtime for `P-T` and `P-F` configurations for k set to 2, and `I-T` configurations for k set to 2 and 5. The results plotted in Figure 6 are similar to those of OH_c and $iTrust_c$. In our experiments, at most 8 tuples failed to be anonymized for `I-T` and $k = 5$, at most 4 for other configurations, and only 1 failure for all configurations with 2,000 raw tuples.

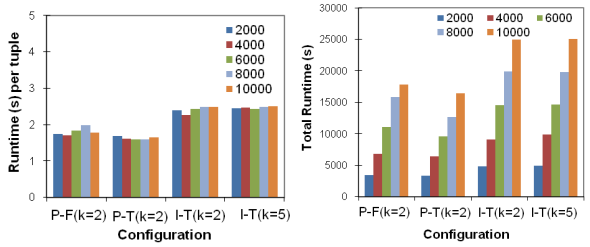

Figure 6. PDM_c runtime: per tuple (left) & all tuples (right)

6. Discussion

Other privacy preservation models. In this paper, we build our kb-anonymity model mainly on top of k-anonymity. There are various other models proposed in the literature, such as l-diversity [25], m-invariance [41], and t-closeness [22]. We leave the possible extensions to cover other privacy preservation models as future work.

Handling more complex programs. In this work, our implementation only handles programs involving integers and real numbers, and cannot solve non-linear or string constraints. In the future, with the advances of symbolic execution, in particular JPF [2], we hope to be able to handle programs with strings and evaluate our prototype implementation on larger, more complex programs.

Also, this paper assumes that each tuple is processed independently. Privacy-related programs, such as healthcare management systems, often deal with individuals and thus only take one tuple as its input, and program states or path conditions for one tuple are not dependent on others. However, there are indeed real-world applications that use multiple individuals' records together (*e.g.*, when analyzing a patient's family medical history), and our current approach will treat those records as one combined tuple. If a piece of code involves multiple tuples in a "batch mode" (*e.g.*, when iterating over a set of patients and printing out each record), it is possible and interesting future work to explore automated program slicing techniques to extract the essential code that deals with one patient only, and then investigate the applicability of our model.

Attacks. We have not considered attack models beyond the natural mappings that come with value replacement functions. We have assumed attackers can only link a released tuple back to a raw tuple through the inverse of a value replacement function. Based on Theorem 2, reversing the function would give no less than k raw tuples and would not breach kb-anonymity. However, it is worthwhile to discuss other kinds of attacks.

k-anonymity is proposed to address the *linking attack* [32, 38] against released datasets that only remove identifier fields. The linking attack works as follows: Some publicly available data (*e.g.*, the voter list of a particular state) contains real values for the identifiers (*e.g.*, a person's name) and the quasi-identifiers that are also contained in a released dataset; then, the values of the quasi-identifiers in the released dataset and the publicly available data may be uniquely matched to recover the values of the identifiers and the sensitive fields of each person. k-anonymity addresses this attack by ensuring that at least k released tuples are indistinguishable from each other. kb-anonymity also addresses the attack by ensuring that each released tuple is mapped to at least k raw tuples. Other attacks against k-anonymity, such as *unsorted matching attack*, *complementary release attack*, and *temporal attack*, can also be addressed by minor modifications of how k-anonymity is applied [32, 38]. kb-anonymity can address these attacks in a similar way by treating all fields *together as one* quasi-identifier.

k-anonymity, however, cannot address other attacks, such as *homogeneity attack* due to the lack of diversity among the values of sensitive fields [25]. Consider the example 2-anonymized dataset in Section 2. One can identify that both Bob and Tom have cancer just by knowing that their age is 53 and their records are in the dataset. As an advantage, our model can address this issue when applied with the `no field repeat` configuration. In this configuration, the disease would be replaced by another value that does not exist in the dataset. There are also other attacks against k-anonymity discussed in the literature [22, 41]. We will investigate the susceptibility of our model to those attacks in future work.

In addition, program versioning may also lead to a privacy concern: our current kb-anonymity model may generate different released datasets for various versions of the same program and attackers may link these versions together to increase the probability of identifying an individual in the raw dataset. Similarly, program back doors may even be a bigger concern. Also, data owners may be tricked into applying our model to programs that violate our assumptions (*e.g.*, a hacker may construct a program whose execution is dependent on not only the current input tuple, but also previous executions with different tuples), then the released data may be used to infer original data. These potential attacks mean that kb-anonymity model needs to be enhanced or applied with appropriate policies to disable undesirable information linkage among various data sources. We leave this as future work.

Data Distortion. As noted in Section 1, kb-anonymity may generate values that do not correctly reflect raw data values. Also, many raw tuples may be suppressed into one released tuple (a la Algorithm 1), causing loss of information. For example, there may be only two released tuples for the example in Section 2, as shown in Table 4, and some sensitive but useful information (*e.g.*, the existence of hypertension) is lost.

Name	NID	Age	Gender	Address	Doctor	Disease
Joel	999	54	Female	Bishan	Dr. Joe	Cancer
Bar	888	32	Female	Changi	Dr. Anne	Flu

Table 4. Sample kb-anonymized tuples that lose information.

Our algorithms may be adapted to remedy the issues for certain cases: Change Line 31 in Algorithm 1 to produce more than one tuple for each path condition so that the values in the released tuples may contain any desired information. However, whether generating additional released tuples is cost-effective or robust against known or unknown attacks is still a question for future investigation. As a result, kb-anonymized datasets are only suitable for the purposes of testing and debugging, where preserving statistics of raw datasets is not a concern.

Threats to Validity. To address threats to construct validity, we have evaluated our model and its realization both qualitatively (by some examples) and quantitatively (by the stress tests for scalability). To reduce threats to external validity on the generalizability of our results, we have evaluated our approach on three different programs. There may still be threats to internal validity (*e.g.*, selection bias) as we choose and slice the three sample programs by ourselves in a similar way. To reduce the threats further, it is possible to include more subject programs and realistic datasets into our evaluations, which we leave as future work. In particular, we plan to employ our model to help a partner in the healthcare industry to anonymize their data and obtain their feedback. In the future, we hope that there will be more interest in this area and a comprehensive benchmark could be built to evaluate similar methodologies and tools.

7. Related Work

In this section, we describe some related threads of work on privacy preservation, symbolic execution, testing & debugging, and legal issues in software engineering.

Privacy Preservation. Motivated by many threats to privacy and issues raised from identity thefts, there have been various studies on privacy preservation. Samarati and Sweeney propose the concept of k-anonymity [32, 38]. There have been various studies on applying and extending their work [6, 7, 10]. Aggarwal *et al.* [7] prove that finding the minimum number of value substitutions to ensure k-anonymity is an NP-complete problem; they thus propose an approximation algorithm to ensure approximate k-anonymity in a dataset. We leverage their algorithm in the realization of our kb-anonymity model (*cf.* Section 4).

There are also other models for privacy preservation, including l-diversity by Machanavajjhala *et al.* [25], m-invariance by Xiao and Tao [41], and t-closeness by Li *et al.* [22]. In this study, we only focus on k-anonymity as the model for privacy preservation.

Information Flow Security. In the areas of programming language and security research, many studies use variants of taint analysis and information flow for protecting sensitive information [24, 26, 31]. There are also studies on removing sensitive information from program execution records, such as core dumps, stack frames, and profiles [11, 40]. Some recent studies focus on monitoring the flow of sensitive information in and across applications, such as TaintDroid [14] and TaintEraser [45], so as to prevent misuse of users' private information.

Our approach is different from all these techniques that we do not allow sensitive information going into a program in the first place, while they detect, block, and remove sensitive information from within programs. However, information flow and taint analysis may be helpful to scale up our approach to identify parts of program and program inputs that involve sensitive information.

Symbolic Execution. Symbolic execution [21] is a program analysis technique that tracks variable values symbolically. A common use of symbolic execution is to compute and manipulate path conditions associated with an execution, and when combined with constraint solvers, to guide program analysis, testing, etc. It has been widely used for various software engineering tasks, including testing, debugging, program analysis, and model checking [9, 20, 29, 34, 42]. Recent studies combine symbolic execution with concrete execution to improve its utility and scalability [12, 16, 36].

For Java PathFinder (JPF) [39], there is also a symbolic execution extension [8]. Jayaraman *et al.* [18] provide a concolic white-box fuzzer, jFuzz, on top of JPF that allows both symbolic and concrete executions. We use jFuzz to collect the path condition corresponding to each test case, which is similar to many existing studies. However, we utilize path conditions for a quite different purpose, which is to guide test data anonymization.

Testing & Debugging. There are numerous studies on testing and debugging. Many test case generation techniques are based on symbolic and concrete executions [12, 15, 36]. Test case prioritization and selection have received much research interest [30, 33]. This is also the case with automated fault localization [19, 23, 35, 43, 44].

In this work we propose a new line of research on data anonymization for testing and debugging. This complements the other research directions on testing and debugging, although it shares some fundamental techniques with them. There might be further interesting research issues raised by combining anonymization problems with test case prioritization, selection, and even fault localization. We leave this research direction as future work.

Legal Issues in Software Engineering. Recently, there have been studies investigating legal issues in software engineering [13, 27, 28]. Metayer *et al.* [27] propose a set of methods and tools to define and establish software liability. Di Penta *et al.* [28] investigate the evolution of software licensing. Cleland-Huang *et al.* [13] propose a machine learning approach for tracing regulatory code to requirements. Technically, these studies are all concerned with software itself, while our study is concerned more about the data that software executes with. However, our study on privacy issues could potentially impact legal aspects in software engineering.

8. Conclusion and Future Work

In this paper, we propose a new problem of privacy preserving testing and debugging. We address the problem of lack of test cases on a developer's side by allowing sensitive yet available test cases to be shipped from software users and data owners to third-party software vendors through anonymization. Anonymization by naively masking away identifiers and sensitive information would not work well due to the issues with quasi-identifiers and ineffectiveness of masked data for testing and debugging. Our approach combines the concept of privacy preservation and program behavior preservation in some interesting ways, and provides guidance on replacing private data values. We build a framework on the top of k-anonymity and concolic execution and implement several configurations. Our empirical evaluations on three sliced real programs show the utility of our prototype on providing effective anonymization for testing and debugging purposes. Our approach would help users to convey more testing and debugging information to software vendors without disclosing private information.

In the future, we plan to address other privacy preservation criteria aside from k-anonymity, incorporate further progress on symbolic execution that is able to handle strings and more complex data structures in programs, and carry out larger case studies.

Acknowledgements

We would like to thank for valuable feedback from the anonymous reviewers and our shepherd Michael Burke. We also thank Julia Lawall and Zhendong Su for their useful comments. Their insightful advice helped to improve our paper.

References

[1] Choco solver. `http://www.emn.fr/z-info/choco-solver/`.

[2] Fujitsu develops technology to enhance comprehensive testing of java programs. `http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html`.

[3] iTrust. `http://sourceforge.net/projects/itrust/`.

[4] Open hospital. `http://sourceforge.net/projects/angal/`.

[5] PDmanager. `http://sourceforge.net/projects/pdmanager/`.

[6] G. Aggarwal, T. Feder, K. Kenthapadi, S. Khuller, R. Panigrahy, D. Thomas, and A. Zhu. Achieving anonymity via clustering. In *PODS*, pages 153–162, 2006.

[7] G. Aggarwal, T. Feder, K. Kenthapadi, R. Motwani, R. Panigrahy, D. Thomas, and A. Zhu. Approximation algorithms for k-anonymity. In *Int. Conf. on Data Theory*, 2005.

[8] S. Anand, C. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extenion to Java PathFinder. In *TACAS*, 2007.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, pages 49–60, 2010.

[10] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x? Anonymized social networks, hidden patterns, and structural steganography. In *WWW*, pages 181–190, 2007.

[11] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *12th USENIX Security Symposium*, pages 273–284, 2003.

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[13] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *ICSE*, pages 155–164, 2010.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[17] P. Golle. Revisiting the uniqueness of simple demographics in the US population. In *5th ACM Workshop on Privacy in Electronic Society (WPES)*, pages 77–80, 2006.

[18] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic tester for NASA Java. In *NASA Formal Methods Workshop*, 2009.

[19] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, pages 167–178, 2008.

[20] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Int. Conf. Data Eng.*, 2007.

[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, June 2003.

[24] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, pages 75–86, 2009.

[25] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), 2007.

[26] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.

[27] D. L. Métayer, M. Maarek, V. V. T. Tong, E. Mazza, M.-L. Potet, N. Craipeau, S. Frénot, and R. Hardouin. Liability in software engineering: Overview of the LISE approach and illustration on a case study. In *ICSE*, pages 135–144, 2010.

[28] M. D. Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *ICSE*, pages 145–154, 2010.

[29] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.

[30] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. In *IEEE Trans. Software Eng.*, pages 929–948, 2001.

[31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[32] P. Samarati. Protecting respondents' identities in microdata release. In *IEEE Transactions on Knowledge and Data Engineering*, 2001.

[33] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[34] R. A. Santelices and M. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA*, pages 195–206, 2010.

[35] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

[36] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[37] L. Sweeney. Uniqueness of simple demographics in the U.S. population. Technical Report LIDAP-WP4, Carnegie Mellon University, School of Computer Science, Data Privacy Laboratory, 2000.

[38] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10:557–570, 2002.

[39] W. Visser and P. Mehlitz. Model checking programs with Java PathFinder. In *SPIN*, `http://babelfish.arc.nasa.gov/trac/jpf`, 2005.

[40] R. Wang, X. Wang, and Z. Li. Panalyst: Privacy-aware remote error analysis on commodity software. In *17th USENIX Security Symposium*, pages 291–306, 2008.

[41] X. Xiao and Y. Tao. m-invariance: Towards privacy preserving republication of dynamic datasets. In *SIGMOD*, pages 689–700, 2007.

[42] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

[43] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.

[44] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[45] D. Zhu, J. Jungy, D. Song, T. Kohnoz, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1), 2011.