kb^e-Anonymity: Test Data Anonymization for Evolving Programs

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

ABSTRACT

High-quality test data that is useful for effective testing is often available on users' site. However, sharing data owned by users with software vendors may raise privacy concerns. Techniques are needed to enable data sharing among data owners and the vendors without leaking data privacy.

Evolving programs bring additional challenges because data may be shared multiple times for every version of a program. When multiple versions of the data are cross-referenced, private information could be inferred. Although there are studies addressing the privacy issue of data sharing for testing and debugging, little work has explicitly addressed the challenges when programs evolve.

In this paper, we examine *kb*-anonymity that is recently proposed for anonymizing data for a single version of a program, and identify a potential privacy risk if it is repeatedly applied for evolving programs. We propose kb^e-anonymity to address the insufficiencies of *kb*-anonymity and evaluate our model on three Java programs. We demonstrate that kb^e-anonymity can successfully address the potential risk of *kb*-anonymity, maintain sufficient path coverage for testing, and be as efficient as *kb*-anonymity.

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, Reliability, Security

Keywords

k-anonymity, behavior preservation, privacy preservation, testing and debugging

1. INTRODUCTION

Quality of test data is important for the effectiveness of testing and debugging. High quality test data could expose hard-to-spot bugs in a program. This data is often available on users' site. However, sharing data owned by a user with software vendors could raise privacy concerns. For example, patient information in United States are protected under HIPAA privacy and security regulation [13]. Techniques are needed to enable data sharing with the vendors for testing and debugging without leaking any private information.

Additional challenges occur when programs evolve. Can we ensure that a set of released data has sufficient coverage for multiple program versions? When data is released multiple times for different versions, can we ensure the multiple shared datasets would not leak private information?

There exist studies that address the privacy issue of data sharing for testing and debugging [4–7, 12, 15]. However, to our best knowledge, no study has explicitly addressed the additional challenges when programs evolve. Often the case, it is assumed that there exist certain strategies to share data multiple times, but no study has examined the implications of data sharing across program versions.

In this paper, we investigate the challenges of generating high quality test data for evolving programs and protecting private information. In particular, we analyze the effects of applying *kb*-anonymity [5] on evolving programs, identify the associated risk, and propose an enhanced model that addresses the risk. Our contributions are as follows:

- We identify a potential privacy risk, namely probing attack that may break *kb*-anonymity if data is shared for multiple versions of a program.
- We propose an enhanced model for evolving programs, kb^e-anonymity, that could address probing attacks.
- We empirically evaluate our model on three Java programs with several synthesized versions and demonstrate that our model can successfully thwart the attacks, maintain sufficient path coverage, and be as efficient as *kb*-anonymity.

The structure of this paper is as follows. Section 2 presents preliminaries related to privacy preservation. Section 3 describes a potential attack, called probing attack. Section 4 describes our kb^e-anonymity model. Section 5 evaluates our model. Section 6 explores more related work, and Section 7 concludes.

2. PRELIMINARIES

We use an example to introduce terms related to privacy preservation. Consider a piece of code provided by a software vendor to the user, as shown in Figure 1, and consider a set of data points (*a.k.a. tuples* or *records*) owned by the user, as shown in Table 1.

ASE '12, September 3–7, 2012, Essen, Germany

Table 1: Sample Private Patient Data.

Name	Zip Code	Age	Disease
Tom	11112	101	Cancer
Jane	21111	105	Diabetes
Jack	11112	14	Cancer
Mary	21111	16	Diarrhea

```
public static void main(String[] args) {
   String name = args[0];
   int age = Integer.parseInt(args[1]);
   String disease = args[2];
   System.out.print("Please take good care of your health: ");
}
```

Figure 1: Sample Program—Initial Version 0

The dataset in Table 1 is comprised of a set of four tuples and four fields: name, zip code, age, and disease. Based on literatures [5,11], the dataset that has no anonymization applied, is called *raw* data or a set of *raw tuples*.

Some fields (*e.g.*, name or national ID) may uniquely identify an individual and called *identifiers*. Some other fields (*e.g.*, age, zip code) may not identify a person, but can identify a person collectively when linked together to publicly available data (*e.g.*, the voter's registration list in US that contains every registered person's name, date of birth, gender, etc.). A set of such fields is called a *quasi-identifier*. A third kind of fields (*e.g.*, disease, time of last visit to a hospital) cannot reveal an identity, but they are private information about a person, called *sensitive fields*.

Anonymizing raw data aims to break the linkage between sensitive fields and identifiers and/or quasi-identifiers. *kb*-anonymity [5] is recently proposed to anonymize data for testing purpose. It is built on top of a well-known privacy model, k-anonymity [11]. To maintain the data quality and privacy, *kb*-anonymity requires that each path executed by a tuple in the anonymized dataset should be executed by at least k tuples in the raw data and each tuple in the raw dataset should not appear in the anonymized dataset. The code in Figure 1 has four raw tuples executing the same path. Thus, when $k = 2$, only one anonymized tuple (e.g., <Jill, 11111, 104, Flu> *a.k.a. shared data*) which is needed to be generated and shared with a vendor. This tuple appears random to the vendor, protecting individual's privacy in the raw data and covering all paths in the code.

3. PROBING ATTACK

When a vendor upgrades a software system, he could request test data *repeatedly* from a data owner for each version of the system. Private information may be leaked if the vendor maliciously designs the versions so that the shared data for a previous version and that for another version can be linked to reveal private information.

We identify risk that related to multiple shared dataset for evolving programs, called *probing attack*. The general steps of a probing attack are as follows:

1. An attacker obtains the identifiers and/or quasi-identifiers for an individual based on public data or previous anonymized data from a data owner.
2. The attacker constructs a program path to be executed by the particular individual's tuple and a limited number of other tuples in the raw data and send this version to the data owner for anonymization.
3. The attacker guesses the sensitive information about the individual and introduces another program path to be executed if the guess is correct, and send the new version to the data owner for data anonymization.
4. The attacker looks for differences between the new anonymized dataset and the previous one, and infer private information based on the differences.

Consider Version 1 of the original code shown in Figure 2 without the part in dashed box. The raw data in Table 1 cover both paths in Version 1. The anonymized tuple in Section 2 (<Jill, 11111, 104, Flu>) reduces the path coverage (covers one path represented by a *path condition*[1] age>100), although the privacy is protected for $k = 2$.

```
public static void main(String[] args) {
   String name = args[0];
   int age = Integer.parseInt(args[1]);
   String disease = args[2];
   System.out.print("Please take good care of your health: ");
   if ( age > 100 ) {
      System.out.println("Venerable elderly");
      if ( name.equals("Tom") && disease.equals("Cancer") ) {
Version 1'   System.out.println("Hi Tom!");
      }
   } else {
      System.out.println("Adorable youth");
   }
}
```

Figure 2: Sample Program—Attack Versions 1 & 1'

To maintain the path coverage, two anonymized tuples would be generated by applying *kb*-anonymity on Version 1, each of which covers one of the two paths. However, repeated applications of *kb*-anonymity on multiple program versions may leak individual-specific information.

Consider Version 1' of the code (all code in Figure 2) that has an additional if condition nested within Version 1. Since only one raw tuple goes to this if branch (represented by the *path condition* age>100 && name.equals("Tom") && disease.equals("Cancer")) and another tuple goes to the else branch (age>100 && !name.equals("Tom")), *kb*-anonymity generates no tuple to go through either of the two paths. Thus, there is only one released tuple that goes through the path age<=100 for the Version 1'. Since the number of released tuples for the version 1' is *one fewer than* that for the version 1, the attacker knows that it is because of the additional if condition and accurately infer that Tom has cancer. This is a breach of privacy!

4. kb^e-ANONYMITY

We present our solution, named kb^e-anonymity, to tackle probing attacks. In a probing attack, an attacker creates a new program path p_{new} in a new version of a program such that a particular raw tuple that executes the same path p_{old} as k-1 other tuples in the older version will execute the new path p_{new}. We cannot release the tuple that executes p_{new}; otherwise, the attacker could infer the private information associated with the *probing tuple*. Also, we cannot release a tuple that executes p_{old} because *kb*-anonymity requires that a shared tuple must execute a path executed by at least k raw tuples. Thus, a second application of *kb*-anonymity on the new version would result one fewer anonymized tuple; this could help the attacker to infer the existence of the probing tuple in the raw dataset. This attack implies that *kb*-anonymity may not be directly suitable for anonymizing data multiple times for evolving programs.

4.1 Subpath Equivalence

Our intuition is that the essence of probing attacks is to split a set of raw tuples executing the same path into subsets that execute different paths; the attack is succeed if some subsets are small enough to single out a tuple. What

[1] A *path condition* for an execution is the conjunction of all the conditionals along the path for the execution.

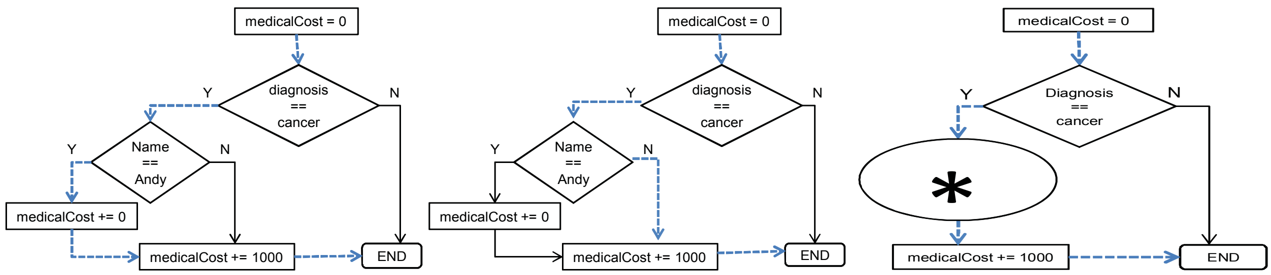

Figure 3: A path on the left (dashed arrows) merged with the path in the middle becomes the *super-path* on the right.

if we prevent the occurrence of small tuple sets? If we can ensure each tuple always "belongs to" a set of size at least k, the attacker could not single it out. kb-anonymity requires anonymized tuples to preserve "behavior", i.e., *whole* program paths. If we can define an appropriate "behavior" preservation, we could ensure each tuple set always contains at least k tuples and prevent probing attack.

Based on this intuition, we propose a new kind of behavior preservation, called *subpath equivalence*.

DEFINITION 4.1 (**Subpath Equivalence**). *Two program execution paths p_1 and p_2 and their corresponding input tuples t_1 and t_2 are* subpath equivalent *w.r.t. a given sequence of program elements sp, if sp is a common subsequence of p_1 and p_2.*

In whole-path equivalence, two path are inequivalent if they differ at least one element, while subpath equivalence allows two paths to be treated as equivalent by adjusting the given common sequence. This allows us to form behaviorally equivalent tuple sets of large enough sizes. Thus, the concept of subpath equivalence is the foundation for kb^e-anonymity.

4.2 Path Merging

The key to foil probing attack is to generate an anonymized dataset that satisfies a relaxed variant of kb-anonymity by preserving subpath equivalence, instead of whole-path equivalence. Whenever a path is executed by less than k of raw tuples, the path is merged with another path, by removing the different segments in the two paths, to form a *super-path sp* that may be executed by enough ($\geq k$) raw tuples. Thus, when we generate one anonymized tuple t' for the super-path sp, at least k raw tuples have subpath equivalent to t' with respect to sp. The actual path taken by t' would be nondeterministically chosen when a constraint solver is used by kb^e-anonymity to resolve sp. Thus, the attacker could not accurately observe the effect of his probes.

Figure 3 illustrates our path merging. The dashed arrows in the left and middle subfigure indicate the paths executed by two different raw tuples. A possible super-path when the two paths need to be merged is indicated by the dashed arrows in the right subfigure; "*" means the parts that are removed. An anonymized tuple is generated to execute the parts in the super-path. Path within the "*" part is actually executed by the tuple, but it would be nondeterministic, thus an attacker could not infer accurately.

Consider the code in Version 1 & 1' in Figure 2. Tom's and Jane's data execute the same path in Version 1, but execute different paths in Version 1' due to the nested `if`. As described in Section 3, if we release an anonymized dataset by applying kb-anonymization on Version 1', Tom's disease would be leaked. Based on subpath equivalence, we generate an anonymized tuple to represent both Tom and Jane by finding a random tuple that satisfies the super-path merged from Tom's and Jane's paths in Version 1' (represented by the path condition `age>100`). Whether this random tuple goes through the nested `if` would be nondeterministic, thus the attacker could not tell whether Tom's data goes into the nested `if` (*i.e.*, whether he has cancer or other diseases).

Refining kb-anonymity with subpath equivalence is how kb^e-anonymity works against probing attack. To enhance the protection, we could require the constraint solver to produce random solutions every time. If the solver always produces the same result for the same path condition, an attacker could infer whether his probe is successful by observing whether the anonymized tuples are changed or not. Although such a randomness requirement is difficult to satisfy, most constraint solvers utilize nondeterminism internally, and we may consider their outputs pseudo-random. To simplify our implementation, we require that every tuple in a kb^e-anonymized dataset must be different from all tuples in the previous anonymized datasets.

5. EMPIRICAL EVALUATION

5.1 Experimental Setting

We evaluate our approach on three Java programs: OpenHospital (OH) [2], iTrust (IT) [1], and PDManager (PD) [3]. OH and IT are medical related applications. PD is an insurance application. We convert parts of the programs into integer programs that take tuple inputs from a file because our implementation is based on JPF [14] and jFuzz [9] which only handles integer constraints so far.

To demonstrate the code path coverage benefit and attack prevention capability of our approach, we modified each of the converted versions of the programs (v_0) to produce an attack version v_p, by adding various branching statements. We semi-randomly created thousands of inputs as raw datasets for each v_0 (3900 for OH, 7300 for IT, and 1900 for PD) that covers many paths in v_0. We implemented our solution in Visual C#.Net and Java, and performed all experiments on a Windows Server 2008 with an Intel Xeon CPU clocked at 2.53GHz.

5.2 Effectiveness on Path Coverage

We compare our approach to kb-anonymity [5] with the *one-time release* strategy (*i.e.*, always releasing the same data anonymized by kb-anonymity for v_0), and show that we achieve higher path coverage.

For each version v_0, we apply kb-anonymity with the raw dataset D, and obtain the anonymized data D_0^{kb} for the one-time release. Also, we add tens of `if-else` statements into v_0 to create a new version v_p with more paths. The branching conditions are chosen so that the additional paths in v_p can be covered by some tuples in D. Then, we apply kb^e-anonymity to v_p for each v_0 to generate the anonymized data $D_p^{kb^e}$. For each program, we compare the path coverage of all data in D_p^{kb} with that of all data in $D_0^{kb^e}$ for v_p.

Table 2 shows the relative path coverage gains by kb^e-anonymity with respect to kb-anonymity. The gains can be quite different for different programs, raw dataset, and privacy requirements for k, ranging from 2% to 100%.

Table 2: Path Coverage Evaluation on v_0 and v_p.

Program	OH		IT		PD	
k	2	5	2	5	2	5
Gain	27.3%	27.3%	2.6%	2.6%	100.0%	100.0%

5.3 Effectiveness on Preventing Attacks

For demonstration, we modify each program to enable probing attack, by adding new branching statements. Note that the attack cannot happen with one-time release strategy, thus we assume data owners use *repeat release* strategy (*i.e.*, applying kb-anonymity on the raw dataset for each new version to release a different anonymized dataset).

For each version of the aforementioned programs v_0, we take its v_p and add a new `if-else` statement to create a new version v_1. The branching conditions are chosen by us so that there are only k tuples in raw dataset D executing the `if` branch. Then, we create another version v_1' by adding another `if-else` statement inside the first `if` branch so that there is only one tuple in D executes the second `if` branch.

We apply kb-anonymity on v_1 first. There would be k tuples executing the first `if` branch, and kb-anonymity generates one anonymized tuple to represent the k tuples. Then, we apply kb-anonymity on v_1'. Only one of the k tuples executes the second `if` branch, and kb-anonymity on v_1' generates no tuple that executes either the first or the second `if` branch. When one fewer tuple is generated, the attacker could know that his probe is successful. But, applying kb^e-anonymity on v_1' would merge the two `if` branches and generate one anonymized tuple. Whether this tuple executes the second `if` branch appears nondeterministic. Thus, the attacker could not know whether his probe is successful.

5.4 Performance

Our kb^e-anonymity collects and solves path conditions and is potentially expensive. A heuristic optimization is applied to obtain an efficient implementation almost linear to the size of the raw dataset. We perform a scalability evaluation with the thousands of tuples generated for each of the three programs. The time of applying kb^e-anonymity on various versions ranges from 1 to 2.3 second(s) per tuple on average, close to the performance of kb-anonymity [5].

6. RELATED WORK

There are other studies on protecting data privacy for testing and debugging. Grechanik *et al.* [8] find that code coverage of k-anonymized data may decrease significantly. Taneja *et al.* [12] show that higher code coverage could be achieved by using a random data-swapping algorithm for maintaining *guessing anonymity*. Compared with our work, their work do not explicitly enforce behavior preservation. Also, the randomization based privacy protection schemes may change certain raw data unnecessarily, and the code coverages of their solutions depend on their internal randomness.

Other studies remove or anonymize sensitive information from program traces [4, 15]. Scrash [4] uses dynamic taint analysis to remove sensitive information. Panalyst [15] reproduces failure-inducing inputs in an interactive setting. Castro *et al.* [6] use execution record-replay techniques with dynamic symbolic execution to anonymize a failure-inducing input. Clause and Orso propose Camouflage [7] to anonymize a failure-inducing input. They mostly generate an anonymized input having the same path as *one* failure-inducing input, while we consider repeated anonymization of a *set* of inputs for *evolving* programs.

7. CONCLUSION

Much data useful for testing and debugging is only available on users' site. Unfortunately, sharing this data would raise privacy concerns. Mechanisms are needed to protect privacy while sharing the data with software vendors for testing and debugging, especially when programs evolve.

We propose kb^e-anonymity, a privacy model for anonymizing data for evolving programs based on a new concept of *subpath equivalence*. Privacy is protected by using path merging, and path coverages of anonymized data are maintained with constraint solving. Our evaluation shows that our model can efficiently anonymize test data, effectively protect privacy against probing attacks, and maintain reasonable path coverages. We also identify and handle other attacks when programs evolve which are discussed in detail in our technical report [10].

8. REFERENCES

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

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

[3] PDM. `http://sourceforge.net/projects/pdmanager/`.

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

[5] A. Budi, D. Lo, L. Jiang, and Lucia. kb-anonymity: A model for anonymized behavior-preserving test and debugging data. In *PLDI*, 2011.

[6] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, 2008.

[7] J. Clause and A. Orso. Camouflage: Automated anonymization of field data. In *ICSE*, 2011.

[8] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE*, 2010.

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

[10] Lucia, D.Lo, L. Jiang, and A. Budi. kb^e-anonymity: Test data anonymization for evolving programs. In *Technical Report, Singapore Management University*, `http://www.mysmu.edu/phdis2009/lucia.2009/Publication.htm`, 2012.

[11] P. Samarati. Protecting respondents' identities in microdata release. In *IEEE TKDE*, 2001.

[12] K. Taneja, M. Grechanik, R. Ghani, and T. Xie. Testing software in age of data privacy: A balancing act. In *FSE*, pages 201–211, 2011.

[13] U.S. Department of Health & Human Services. Health information privacy. `http://www.hhs.gov/ocr/privacy/`.

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

[15] 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.