TransformeR: Safe R-like Variable Transformations for PSI (Draft)

Michael LoPiccolo
Privacy Tools for Sharing Research Data REU
PI: Salil Vadhan, **Mentors:** James Honaker, Jack Murtagh
Special thanks: Stephen Chong, Marco Gaboardi, Victor Balcer, Remy Wang

August 13, 2017

1 Background

PSI (the Private data Sharing Interface) is a tool being developed by the Privacy Tools project. Its goal is to enable the wider sharing and use of research data while providing a strong guarantee of privacy protection to data subjects.

It aims to achieve this by enforcing the mathematical constraint of differential privacy. To satisfy the condition of differential privacy, an algorithm must add sufficient noise to ensure that it is very difficult to trace the results back to any particular data subject's contribution. (Formally: it must ensure that, for any database and for any binary decision rule on the results, the probability of deciding one way doesn't increase by a factor more than $exp(\varepsilon)$ when any single row is removed or changed.) [1]

It immediately follows from the definition that we can allow 'per-row' transformations to the database before dispatching to a differentially private algorithm. (A 'per-row' transformation is one which treats each row of the database independently.) This will not compromise the guarantee of differential privacy: Note that the guarantee must hold with respect to every possible input database — this includes the post-transformation database. Changing or removing one row in the original database will change or remove exactly one row in the post-transformation database, so the same guarantee still holds!

The TransformeR project aims to take advantage of this result to allow the construction of many more useful queries in PSI. For example, TransformeR might be used to take the mean of the log of a variable, whereas before one could only take the mean of the variable itself. There are a great number of possibly useful transformations (logs, powers, sums, weighted sums, boolean logic, category joining, missing-value coercion, etc...), so it would be difficult or impossible to anticipate every possible useful transformation and provide an ad-hoc implementation for each one. Therefore, we designed TransformeR to let users supply arbitrary R-like transformation formulas. Currently, we've implemented TransformeR for the budgeter interface — we also imagine that it will be very useful in the interactive query interface when it is ready.

2 Roadmap

My goal was to create a working implementation of TransformeR and fully integrate it into PSI. I built upon work done by Remy Wang in 2016: a paper[3] and some prototype Haskell modules. I did end up rolling my own Haskell modules, but I often looked to Remy's structure for guidance, and the theoretical foundation laid in the paper was greatly valuable!

This paper will detail the technical aspects of my implementation, survey the practical and theoretical issues I encountered while creating it, and provide analysis of some remaining unsolved issues and directions for future development.

3 Technical Details

3.1 Choice of Language

Although most of the PSI backend is written in R, the parser and evaluator for TransformeR are implemented in Haskell. This has added some complexity to the project, but it was motivated by several compelling reasons.

The most concerning aspect of developing TransformeR is that it must accept and act on arbitrary input from users, without ever violating differential privacy. Releasing even one bit of information, such as "was there a crash when transforming the database?" is unacceptable. [2] Haskell's strong typing and strictness help us to be confident that we won't be hit by unexpected edge cases.

It would be extremely dangerous to run user-provided R code in the same context as the rest of the server, so we'd almost definitely have to write a parser even if we stayed in R. Haskell's purely functional paradigm makes it an excellent language for writing parsers, and the Parsec library got us off the ground much more quickly than if we'd rolled our own lexer and parser.

Using Haskell has many benefits, but it also creates a new danger: the possibility of miscommunication between TransformeR and the rest of the backend. We've had to give careful consideration to the interface between these parts to make sure there's no miscommunication with escaped characters, differing type formats, etc.

3.2 Executable Design

TransformeR is an executable which communicates via standard input/output. The first line of input should be a formula describing the transformation(s). (Each transformation is an assignment statement, e.g. `{test <- var1 + var2}`. Semicolons can be used to sequence multiple assignments into a single formula.)

If the formula successfully parses, the user can terminate the input (if they only wanted to check parseability), or continue. If continuing, the second line should be a list of the column names in the database. TransformeR will respond with the names of the columns in the post-transformation database. After that, simply send each row of the database to TransformeR. For each row, TransformeR will read the values, run the transformation, and output the transformed version. (TransformeR expects and outputs tab-separated values at all points.) This format has the benefit of mimicking exactly the format in which Dataverse provides us the data, so the backend can read data from TransformeR exactly as it would have read it from Dataverse.

It's fine to only send the first one or two lines to TransformeR — this can be used to verify transformations before processing. TransformeR will immediately return a nonzero exit code if the first line is unparseable or if the second line doesn't have all the variables the first line tries to access.

3.3 Integration Work

We added one endpoint to rApache: `verifyTransformApp`, which takes a transformation formula and returns either a parse error or an indication of success. It is intended to be called when the user enters a new transformation, so we can notify them early if they've made an error.

We modified the `privateStatisticsApp` endpoint to now optionally accept a transformation string. If it recieves one, it verifies the formula again, and then applies it to create the transformed columns before releasing statistics as normal.

We also modified the budgeter UI to support transformations. When the user launches the final dispatch, the budgeter takes all the transformations, joins them by semicolons, and sends them to the backend. The backend will send the formula and the data to TransformeR (failing early if there's somehow a parsing error), and get the transformed database back. After this point, transformed variables are treated exactly the same as normal variables!

3.4 Type System

One aspect of Remy's work was to design a very strict type system — it would allow us to take the type and range of every incoming variable, and determine the type and maximum range of every outgoing variable.

This would be a valuable system, but not necessary, and I did not have time to bring over Remy's implementation. One issue that would have limited its usefulness is that we often don't already know the types and ranges of columns in the database, so we'd have to ask the user for a lot of information. [1] Also, even if we can infer a range, it may be too large, and the user will have to refine it anyway.

Because we don't yet have a good system to record a reasonable type and range for every column in the database, we might often need to ask users for a lot of information. In addition, users might sometimes need to make ranges narrower than what we can infer (if they know some inputs are not possible).

Instead, I moved to a very loose type system, more like R. Users will have to specify what output type they expect (although in most cases this could probably be automatically determined — good avenue for future work).

There's no privacy issue with type inference when actually evaluating transformations, as long as we treat each row independently! However, we do have to decide what type behaviors are intuitive to users. I've written TransformeR to be as loose as possible — we always attempt to convert to other types before going to NA. This mimics the way R works, so it should be useful to social scientists who are used to this behavior. There are a couple concerns, however:

First, when TransformeR ingests data, it doesn't know the "actual type" of data coming in. Right now if we read a string saying "True" it's going to be a Boolean because we can't tell the difference and we can't believe Dataverse's inclusion of quotes or not. I think this is only a problem with the `==` operator: Transformer thinks that `"1" == "TRUE"` should return TRUE. This could be fixed by allowing users to annotate the actual type of columns (even better if Dataverse could remember these annotations for us!) Second, the type system is often looser than R. For example, I allow `"3" * 2` to evaluate to 6, instead of throwing an error as R does. I've been assuming that going to NA is always worse than the alternative, but that isn't always the case: some DP algorithms might make use of NAs instead of coercing them, or some users might want to do NA coercion intentionally. This wouldn't be that hard to fix, but I think we'd need to require a type annotation for every column to avoid unpleasantness — probably not worth it.

4 Privacy and Security Concerns

4.1 Securing TransformeR

The paper "Differential Privacy Under Fire" [2] was an excellent resource for considering possible side-channel attacks and developing the proper thinking for handling user-provided queries.

4.1.1 Code Correctness, Input Validation

We took care to ensure that per-rowness is guaranteed in TransformeR, giving consideration to a range of possible attacks. Per-rowness may be violated if changing one row influences an answer to another, or if TransformeR can be caused to crash on certain data. (Parse errors are fine, because those happen before any data is considered.)

Thanks to the properties of Haskell, the former is highly unlikely. Each output row is a function which takes an AST and an input row. Haskell is purely functional, so the result is guaranteed to be the same when the arguments are the same. The AST is only created once, and parsing each row should be deterministic. (We do have to be careful that the parser reads each row correctly and doesn't get misaligned — we use newlines to separate rows to make sure this doesn't happen.)

4.1.2 Math Concerns

Many differential privacy applications are concerned with floating point math. However, there shouldn't be any worry in TransformeR about floating point math. Even if we don't know the exact result of a floating point computation, at least we know it will be the same for each row, which is all we need.

[1] Inferring types or ranges cannot be done in a differentially private fashion. However, we may have a better solution in the future, such as asking the data depositor to attach types to the data at the moment of uploading.

Right now, we don't have any transformations with randomness. Technically, using a pRNG violates per-rowness[2], but with a cryptographically secure pRNG this should not be an issue. So, if we want to add transformations with randomness, just make sure the randomness is cryptographically secure.

4.1.3 Error Handling

I gave careful consideration to the structure of error handling in TransformeR. It first parses the transformation and makes all other possible checks before reading the actual rows of the database; then it makes sure there's no problem ingesting the database; only then does it start evaluating expressions. This is to limit the ability of adversaries to divine information about the data by checking whether, or when, an error is thrown. Unfortunately, because of Haskell's lazy evaluation, we can't promise that the chronological order I described earlier is actually strictly followed, but in practice it is.

4.1.4 Exceptions

I attempted to limit the danger of exceptions — we don't want one row to cause all of TransformeR to crash. If an operation can't be performed, it returns NA instead of throwing an exception. If something in a row does throw an exception (very rare in Haskell), we attempt to catch it and print a default value for that row, without affecting any other rows. There are still some factors beyond our control (running out of memory or stack space, cosmic rays, etc...) which might cause TransformeR to crash, but it's highly unlikely that an attacker can leverage these effectively. We will limit the total number of AST nodes in a transformation to ensure that transformations do not use too many resources. One concern I do have is that if PSI is used for very large datasets, we will have to give careful consideration to the memory use (the backend and TransformeR aren't particularly efficient.)

4.1.5 Timing Attacks

One more side-channel that we need to consider is timing attacks. A user could provide a transformation which takes longer on some rows than on others, thereby gaining information on the contents of the database. I haven't implemented a solution to this issue, but we did start the discussion on how to solve it.

One possible solution to this issue is to treat the amount of time a computation takes as another released value, and add a noisy amount of additional delay.[3] Under this approach, it shouldn't be difficult to integrate TransformeR. Because we limit the size of an AST and don't allow loops, we know that evaluating a row shouldn't take too much time no matter its contents. It's possible that changing a single row could change the number of times garbage collection is triggered, so we should analyze whether we are ever likely to hit garbage collection, and if so, how long it might take at most.

We should be especially cognizant of how error handling might enable timing attacks. For example, we might add noise only at the very end of a computation, expecting that noise to cover all aspects of the computation, including TransformeR. However, suppose we apply the transformation, and only then notice that the query is malformed (perhaps due to a malicious user). We immediately stop, returning an error document. In this case, the malicious user would be able to determine the amount of time the transformation took much more accurately than we expected.

To resolve this issue, we should delineate a clear point in the code before which we do not look at data. Failing before then is necessarily totally safe. Afterwards, we should make sure to catch failures and add the whole amount of timing noise.

4.2 Securing PSI around TransformeR

I took it as my mission to help ensure that not only TransformeR, but PSI as a whole, is secure and privacy-protecting in all cases. In fact, building TransformeR into the backend helped us to think defensively about general patterns of data flow inside PSI. As a result, we made some changes to how PSI handles type inference and missing values (two things that are very transparent and easy to ignore in R). These changes

[2]Deleting a row or changing the number of pRNG calls it makes will affect the answers to other rows

[3]Do we need to know the range of times a computation might take, or can we do with the maximum amount of time increase/decrease from changing a single row?

were necessary to protect differential privacy in certain edge cases (which TransformeR would have made easier to exploit).

4.2.1 Missing Values

Currently, not all the algorithms PSI dispatches to are able to handle missing values in a DP-preserving manner (hopefully, someone will take on the project of resolving this.) In the meantime, I added an imputation step[4] to the PSI backend: after transforming, any remaining missing values are imputed to an arbitrary value within the expected range. This isn't desirable from a utility standpoint, but it does protect privacy — and, ideally, users will use transformation functionality to impute NAs before this point.[5]

Note that someone will need to add an implementation for categorical data (right now, the logic hasn't been written to handle constraints on bins at all.)

4.2.2 Type Inference

We also gave careful thought to the process of type inference, to avoid mistakenly leaking information. When R ingests the data (with read.table), it makes the type of each column the tightest type possible (if every entry looks like a number, the type of the column will be numerical — if there's even one that doesn't look right, it'll be categorical). This violates DP, so I took it out — instead, we only convert the data to the type the user said it was.[6]

5 Remaining Issues

A big possible point of improvement is the UI for transformations. There's a lot of opportunity to make it easier for users, without needing to make any significant changes to the backend. We can bring over the feature from TwoRavens that suggests possible transformations in a drop-down menu. We can provide UI shortcuts to effect common-but-unintuitive transformations. NA imputation would be trivial: for example, a user could use the GUI to choose "impute NAs to 4", and get out `x <- ifelse(is.na(x), 4, x)`. With some changes to the result viewer, we could construct queries to calculate the number of NAs, means of subsets (which is the quotient of two other queries), and possibly other very useful functions — without having to write new differentially private routines. NA imputation could also help users be more aware of the danger that we might impute their NAs for them, although we also don't want to scare users away or confuse them with too many options. (Perhaps we want to hide some options behind an advanced mode?)

Then, there are some other changes big and small that would be useful.

- As discussed earlier, we still need to implement protection against timing attacks.
- Someone needs to code the section in enforceConstraints to do with categorical values — just leaving this here in case you were going to forget!
- To sum up the improvements that could be made with to the type system:
  - We could sometimes have TransformeR tell us what type a transformation is going to be, saving the user a click.
  - We could add type annotations, as described above, to keep us from ingesting strings as numerics/booleans. It would be even better if Dataverse could remember type information, because then users wouldn't have to fill out the annotations themselves.
  - Using Remy's type reasoning system would be really convenient in cases where we know all the types and ranges of incoming variables.

[4] See Git commit b6394ab7 in privateZelig

[5] We should consider creating a UI functionality to do this transformation without requiring the user to think of the imputation transformation themselves.

[6] See Git commits f1406238, bec56489 in privateZelig.

- We'd like to support switch, Euclidean norm, etc in TransformeR, but it'll take a refactor to support functions with an arbitrary number of arguments. I might work on this in the next weeks if I feel inclined.
- I need to implement is.na, is.numeric, etc. in TransformeR — should be a small change.
- The floating point parse code could be rewritten to accept numbers without a integral or fractional part: `.2` or `2.`
- We are considering various ways to have PSI communicate with Dataverse — it's worth looking into the possiblity of letting the data depositor's transformations be permanent, and then we might need to have per-user transformation storage in the metadata.
- I described some remaining integration work in the `README.md` file: Some work needs to be done to include TransformeR compilation in the install, and I wrote some guidance text that ought to be added to the budgeter website.
- It would be nice to have unit tests for TransformeR, especially if people want to refactor or expand it.
- Ultimately we'd like to hook TransformeR into the interactive query interface, using the TwoRavens code.

References

[1] Cynthia Dwork. "Differential Privacy". In: *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*. ICALP'06. Venice, Italy: Springer-Verlag, 2006, pp. 1–12. ISBN: 3-540-35907-9, 978-3-540-35907-4. DOI: 10.1007/11787006_1. URL: http://dx.doi.org/10.1007/11787006_1.

[2] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. "Differential Privacy Under Fire". In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 33–33. URL: http://dl.acm.org/citation.cfm?id=2028067.2028100.

[3] Yisu Remy Wang. "TransfomeR: A DSL for Safe Variable Transformation". Aug. 12, 2016. URL: http://privacytools.seas.harvard.edu/files/privacytools/files/remy-final-paper.pdf.