On Synthesis for Differential Privacy

Calvin Smith
University of Wisconsin – Madison
cjsmith@cs.wisc.edu

Abstract

Data analysis has the capability to enrich the lives of many, yet it presents a fundamental threat to individual privacy. Formal privacy constraints such as *differential privacy* serve to protect individual rights while allowing large-scale data analytics to proceed relatively unhindered. However, such constraints are a significant barrier to the access of relevant data. In this abstract, we give a solution that leverages *type-directed synthesis* and the privacy-aware type system DFuzz to allow users to automatically and efficiently synthesize programs that respect privacy constraints. Furthermore, we introduce a technique for proving randomized privacy mechanisms are sufficiently *accurate* via reduction to synthesis.

Keywords differential privacy, program synthesis

1 Introduction

The ability to access and manipulate data is critical to make predictions and inferences about the world around us. However, many important datasets now contain sensitive information that should be kept secure so as to protect the privacy rights of individuals. *Differential privacy* [7] has become a standard definition of privacy, where the goal is to obscure the presence of sensitive information in a dataset via the addition of *noise*. Differential privacy is now used by major corporations, including Google [8], Apple [3], and Uber [13], and government bodies, such as the U.S. Census Bureau [4], in their analyses and information releases.

The inherent complexity of differential privacy makes accessing data complicated, and presents an significant barrier-of-entry. This abstract presents two lines of work towards automation of data analytics in the face of differential privacy via program synthesis. The first, conditionally accepted at ICFP 2019, is a type-directed synthesis technique that leverages the linear dependent type system DFuzz [9] to enable end users to easily synthesize privacy-aware queries. The second, presented at POPL 2019 [15], leverages the proof technique *trace abstraction* [10] to reduce proving accuracy conditions of randomized programs to program synthesis.

We assume the reader is familiar with standard notions of differential privacy (hereafter DP). If not, we recommend the excellent overview by Dwork and Roth [7].

2 Synthesizing DP Programs

To facilitate the democratization of data, we desire a system that allows non-technical end users to interact with privacy-aware systems in a straightforward manner. The following section presents a technique for synthesizing privacy-aware queries from simple constraints by inverting the typing rules of the linear dependent type system DFuzz [9].

Overview of DFuzz We leave the full presentation of DFuzz to Gaboardi et al. [9], and merely present the relevant pieces of the linear subsystem here. DFuzz is a *linear* type system, and so uses a *linear modality* $!_k\sigma$ to keep track of the *sensitivity* of values. This modality appears in two primary forms:

1. *Function domains* - a type $!_k\sigma \multimap \tau$, hereon written $\sigma \multimap_k \tau$, is a function that is k-sensitive in its argument, i.e., if we have *distance metrics* d_σ and d_τ, then for all $x, y \in \sigma$, $d_\tau(f(x), f(y)) \leq k \cdot d_\sigma(x, y)$.
2. *Typing contexts* - A type context containing the assumption $!_k x : \sigma$ can type expressions that are *at most* k-sensitive in x.

DFuzz maintains a distinction between deterministic and probabilistic values of type σ via a *probability monad*, denoted by $\bigcirc\sigma$. By reasoning about metric preservation, Gaboardi et al. arrive at the following critical observation:

Theorem 2.1. *The execution of any closed program e such that $\vdash e : \sigma \multimap_\epsilon \bigcirc\tau$ is an ϵ-DP function from σ to τ.*

Using Theorem 2.1, to synthesize a DP program, we simply need to find a program of type $\sigma \multimap_\epsilon \bigcirc\tau$ satisfying our correctness constraint. Type-checking DFuzz programs is undecidable [6], so simple guess-and-check is unsuitable. Fortunately, privacy is often enforced via a limited number of provably-correct *mechanisms*, which we can utilize as higher-order components in synthesis.

Mechanisms Differential privacy, even in complicated systems, is often enforced via the use of primitives called *privacy mechanisms*. The most-used mechanism is:

Theorem 2.2 (Laplace Mechanism)**.** *Let q be a k-sensitive real-valued query, and let $\epsilon > 0$ be a fixed privacy parameter. The program $\lambda x.q(x) + X$, where $X \sim \text{Lap}(k/\epsilon, 0)$, is a randomized program that is ϵ-differentially private.*

This transformation is referred to as the Laplace mechanism.

We can encode the above as a primitive in DFuzz:

$$\vdash \text{LapMech} : \forall k.(\sigma \multimap_k \mathbb{R}) \multimap_\infty \sigma \multimap_\epsilon \bigcirc\mathbb{R}$$

Observe that the amount of noise added scales with the sensitivity of the provided query. The interplay between sensitivity, utility, and privacy is a complicated one, and often reduces down to economic matters [11]. We assume a user is able to place an *upper-bound* on k, the sensitivity of the query.

Synthesis Our synthesis procedure maintains tuples $\langle\phi, p\rangle$ containing *i)* a partial solution p containing *wildcards* $\bullet^{\Omega}_{\sigma}$ representing unrefined sub-terms with type σ in context Ω, and *ii)* a constraint ϕ restricting when the partial solution p is well-typed. Synthesis is defined by a set of inference rules that refine wildcards and accumulate constraints, terminating when a closed program is found that is well-typed and satisfying the provided correctness constraint. These inference rules are derived by effectively inverting the typing rules for DFuzz.

Inverting Typing Rules Because typing contexts maintain sensitivities of variables, combining contexts is a linear operation, as seen in the rule for function application:

$$\frac{\Omega \vdash f : \sigma \multimap_r \tau \quad \Gamma \vdash e : \sigma}{\Omega + r \cdot \Gamma \vdash f\ e : \tau}\ (\multimap E)$$

Intuitively, because f is r-sensitive, we require r uses of context Γ to type the application $f\ e$. If we wish to invert the rule $(\multimap E)$ to synthesize function applications, we must *find a way to linearly decompose our initial context.* Unfortunately, there are possibly infinitely-many such decompositions, so we introduce *symbolic context constraints* (SCCs) to abstractly record the space of possible decompositions:

Definition 2.3 (Symbolic Context Constraint)**.** A *symbolic context* is a term C in the grammar

$$C := \emptyset \mid \{x :_R \tau\} \mid \Omega \mid C + R \cdot C,$$

where Ω is a context variable, R is a sensitivity term, τ is a type, and s is a sensitivity variable. A *symbolic context constraint* is a conjunction of equalities of symbolic contexts.

SCCs have a natural interpretation consistent with the expected notions of context equality and manipulations. Checking satisfiability of SCCs is done using off-the-shelf SMT solvers and a procedure for converting SCCs to *equisatisfiable* formulas in the theory of non-linear real arithmetic.

Using SCCs, we can invert the DFuzz rule $(\multimap E)$:

$$\frac{\langle\phi, p[\bullet^{\Omega}_{\tau}]\rangle \quad \Omega_1, \Omega_2, r \text{ fresh}}{\left\langle\phi \wedge \Omega = \Omega_1 + r \cdot \Omega_2, p[\bullet^{\Omega_1}_{\sigma \multimap_r \tau}\ \bullet^{\Omega_2}_{\sigma}]\right\rangle}\ \textsc{App}$$

Note the SCC in the consequent. Informally, this inference rule states that we can replace a wildcard with a function application, assuming we properly decompose the context variable Ω into the symbolic context $\Omega_1 + r \cdot \Omega_2$, and can find an r-sensitive function to apply.

Pruning and Heuristics The constraint ϕ in the state $\langle\phi, p\rangle$ becomes unsatisfiable when it is impossible for p to satisfy the typing or privacy constraints. As constraints are only ever extended, if we arrive at a state where the constraint ϕ is unsatisfiable, we are justified in excluding any state derivable from that synthesis state from the search, as no completion of p will be able to satisfy the privacy budget.

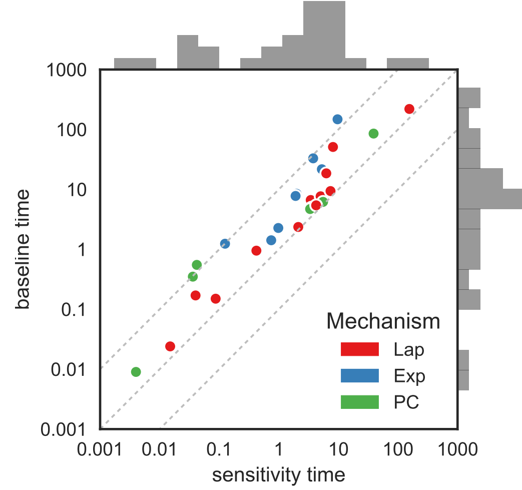

Figure 1. Each benchmark is a pair in log-space comparing sens.-directed and baseline. Top, middle, and bottom dotted lines are the 10x, 1x, and 0.1x levels of the efficiency gradient. Marginal distributions are projected on the borders.

Checking satisfiability at every stage of synthesis is prohibitively expensive, however. Instead, we repeatedly simplify constraints via *constant propagation*, and use the structure of the resulting formula as a heuristic to direct the search towards states that are *more likely* to be satisfiable.

Results After inverting the rest of the typing rules for DFuzz, which involves introducing an *abduction* judgement for handling subtyping, we evaluated a determinization of our synthesis procedure on benchmarks taken from several real-world datasets [5, 12, 14], which consists of 28 queries using the Laplace mechanism, parallel composition mechanism, and exponential mechanism [7]. To evaluate the effect of considering sensitivities during synthesis, we compare to a baseline type-directed synthesis tool that does not maintain constraints. The results are plotted in Figure 1.

Considering sensitivities results in an average improvement of 3.9x, and a maximum improvement of over 30x. All but 2 sensitivity-directed benchmarks terminate in under 10 seconds. These numbers strongly suggest that there are very tangible benefits to maintaining constraints over partial solutions and using them to direct the search.

3 Proving Accuracy of DP Programs

Often, an end user desires a program that is not only privacy-preserving, but *useful.* Common notions of utility for privacy mechanisms are given in terms of accuracy, and roughly correspond to the intuition that *usually, a close-enough answer is returned.* More precisely, we are concerned with statements that take the form of *probabilistic Hoare triples*:

Definition 3.1 (Probabilistic Hoare Triple). The judgement $\vdash_p \{\phi_{\text{pre}}\} P \{\phi_{\text{post}}\}$ is said to hold if, for all program states $s \models \phi_{\text{pre}}$, we have $\Pr_{s' \sim P(s)}[s' \models \phi_{\text{post}}] \geq 1 - p$.

Proving probabilistic Hoare triples is unfortunately difficult to automate in the case of accuracies of privacy mechanisms, as existing techniques are unable to handle the combination of complicated, continuous distributions and symbolic failure properties required. In this section, we summarize a technique from Smith et al. [15] that is able to automatically prove accuracy conditions in the face of the above challenges via reduction to *program synthesis*.

Proof Rule We rely on *trace abstraction* as a framework for our approach [10]. Trace abstraction is a technique for proving *deterministic* Hoare triples $\{\phi_{\text{pre}}\} P \{\phi_{\text{post}}\}$ hold, and is simple in concept: *i)* model P as an automaton whose language contains all traces in the control-flow graph of P, *ii)* sample a trace $\pi_i \in \mathcal{L}(P)$ and prove $\{\phi_{\text{pre}}\} \pi_i \{\phi_{\text{post}}\}$, *iii)* repeat step *ii* until no *new* traces can be sampled. If this procedure succeeds, the control-flow graph of P has been proven correct, and so the Hoare triple holds.

To extend trace abstraction to *randomized programs*, we need to modify our proof rule slightly:

Theorem 3.2 (Trace Abstraction for Randomized Programs). *The probabilistic Hoare triple* $\vdash_p \{\phi_{pre}\} P \{\phi_{post}\}$ *holds if*

1. $\vdash_{p_i} \{\phi_{pre}\} \pi_i \{\phi_{post}\}$ *for all* $\pi_i \in \mathcal{L}(P)$
2. $\sum_i p_i \leq p$

This proof rule is a simplification of the technique presented in Smith et al. [15], but is sufficient to prove simple programs correct. Checking condition 2 is straightforward using an SMT solver (as the p_is are likely symbolic), so the rest of this presentation focuses on checking condition 1.

Reduction to Synthesis To prove $\vdash_p \{\phi_{\text{pre}}\} \pi \{\phi_{\text{post}}\}$, we follow the classic strategy of encoding the semantics of π as a logical formula. However, as π is a trace in a randomized program, it very likely contains *sampling statements* of the form $x \sim \mathcal{D}(e)$, where x is a program variable, $\mathcal{D}$ a distribution function, and e an expression that parameterizes $\mathcal{D}$. To encode sampling statements, we turn to *distribution axioms*, which are statements of the form $\Pr_{x \sim \mathcal{D}}[\phi_{\text{ax}}] \leq p$, where p is a $[0, 1]$-valued expression and ϕ_{ax} is a formula. To use an axiom, we note that we can *assume* $\neg\phi_{\text{ax}}$ holds if we accumulate the chance of failure p.

Axioms can be *parameterized* by an uninterpreted function f whose inputs are the input variables to the program V. For example, concentration inequalities for the Laplace distribution give the following family of axioms:

$$\Pr_{x \sim \text{LAP}(s,m)}[|x - m| > (1/s) \log(1/f(V))] \leq f(V).$$

The use of axioms enables a natural encoding:

Table 1. Results on private algorithms. PA: # of proposed axioms (synthesis solutions); time is in sec.

Algorithm	Axiom(s) synthesized	PA	Time
RResponse	$priv \iff \text{Snd}(r)$	162	2
NoisySum	$\lvert Q\rvert /p$	5	98
RNM	$\lvert Q\rvert /p$	4	33
ExpMech	$\lvert R\rvert /p$	3	27
AboveT	$2/p$ and $2\lvert Q\rvert /p$	22	23
SparseVec	$3/p$, $3\lvert Q\rvert /p$, and $3/p$	941	97

Theorem 3.3. *The triple* $\vdash_p \{\phi_{pre}\} \pi \{\phi_{post}\}$ *holds iff the following formula is satisfiable:*

$$\forall V, \omega_i.(\phi_{pre} \wedge \omega_0 = 0 \wedge \phi_{sem}) \Rightarrow (\omega_n \leq p \wedge \phi_{post}),$$

where ω_i *tracks the cumulative chance of failure at statement* i *and* ϕ_{sem} *encodes the statement semantics [15].*

Note, however, that any use of an axiom family can introduce a *free* uninterpreted function f to the encoding. For a trace with one uninterpreted function in its logical encoding, the above formula is more precisely written as:

$$\exists f \forall V, \omega_i.(\phi_{\text{pre}} \wedge \omega_0 = 0 \wedge \phi_{\text{sem}}) \Rightarrow (\omega_n \leq p \wedge \phi_{\text{post}})$$

Checking satisfiability therefore becomes a *synthesis* problem, where we must find interpretations for each f such that the rest of the formula is satisfiable. Our implementation solves these synthesis problems using a standard bottom-up term enumeration strategy [1, 2].

Results A prototype implementation of the above procedure, with the necessary extensions from [15], was able to automatically prove accuracy properties for 6 real-world DP mechanisms. Results are summarized in Table 1.

Synthesized axioms, in general, are quite simple. This observation, combined with the low correlation between the number of proposed axioms and the synthesis time suggest the performance bottleneck lies elsewhere in the implementation. This is a promising revelation: we have likely not yet reached the limits of synthesis in proof automation.

4 Conclusion

This abstract presents two lines of work towards the automation of data analytics in the presence of differential privacy: the efficient synthesis of privacy-aware queries, and the automated proof of accuracy of randomized programs. Both contributions make a reasonable step forwards in lowering the barrier-of-entry for accessing data protected by formal privacy guarantees by leveraging synthesis techniques.

As it stands, these two developments are disjoint. We imagine a system that incorporates both privacy *and* utility into synthesis, so that highly efficient answers are routinely generated. Furthermore, we believe there is more for synthesis to automate in the realm of DP by considering frameworks such as *adaptive differential privacy* [16].

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950.

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

[3] Apple. Accessed 11-11-2017. Differential privacy. https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

[4] US Census Bureau. Accessed 11-11-2017. On The Map. https://onthemap.ces.census.gov/.

[5] Paulo Cortez and Alice Silva. 2008. Using data mining to predict secondary school student performance. (01 2008).

[6] Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014*. 5:1–5:12.

[7] Cynthia Dwork and Aaron Roth. 2013. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3-4 (2013), 211–407. https://doi.org/10.1561/0400000042

[8] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. ACM, 1054–1067.

[9] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2429069.2429113

[10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[11] Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin Pierce, and Aaron Roth. 2014. Differential Privacy: An Economic Method for Choosing Epsilon. *Proceedings of the Computer Security Foundations Workshop* 2014. https://doi.org/10.1109/CSF.2014.35

[12] Lauren Kirchner Jeff Larson, Surya Mattu and Julia Angwin. [n. d.]. How We Analyzed the COMPAS Recidivism Algorithm. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm/ Accessed: 2017-11-15.

[13] Noah M. Johnson, Joseph P. Near, and Dawn Xiaodong Song. 2018. Practical Differential Privacy for SQL Queries Using Elastic Sensitivity. *VLDB*. http://arxiv.org/abs/1706.09479

[14] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[15] Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace Abstraction Modulo Probability. *Proc. ACM Program. Lang.* 3, POPL, Article 39 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290352

[16] Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A Framework for Adaptive Differential Privacy. *Proc. ACM Program. Lang.* 1, ICFP, Article 10 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110254