Sensitivity Analysis using Type-Based Constraints

Loris D'Antoni[⋆] Marco Gaboardi[⋆‡] Emilio Jesús Gallego Arias[⋆] Andreas Haeberlen[⋆]
Benjamin C. Pierce[⋆]

[⋆]University of Pennsylvania [‡]Università di Bologna–INRIA EPI Focus

Abstract

Function *sensitivity*—how much the result of a function can change with respect to linear changes in the input—is a key concept in many research areas. For instance, in *differential privacy*, one of the most common mechanisms for turning a (possibly privacy-leaking) query into a differentially private one involves establishing a bound on its sensitivity.

One approach to sensitivity analysis is to use a type-based approach, extending the Hindley-Milner type system with functional types capturing statically the sensitivity of a functional expression. This approach — based on *affine* logic — has been used in *Fuzz*, a language for differentially private queries.

We describe an automatic typed-based analysis that infers and checks the sensitivity annotations for simple functional programs. We have implemented a prototype in *Fuzz*'s compiler. The first component of the analysis extends the typechecker to generate nonlinear constraints over the positive real numbers extended with infinity, which are then checked by the Z3 SMT solver; a solution for them will provide an upper bound on the sensitivity annotations and ensure the correctness of the annotations. We also present a simple sensitivity minimization procedure and demonstrate the effectiveness of the approach by analyzing several examples.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; F.3.3 [*Theory of computation*]: Studies of Program Constructs—Type structure

General Terms Design, Languages, Theory

Keywords Sensitivity Analysis, Differential Privacy, Linear Types, Special Purpose Language, SMT solver

1. Introduction

The *sensitivity* of a function $f(x)$ is an upper bound on how much $f(x)$ can change in response to a change to x—in other words, if f has sensitivity k, then $|f(x+\delta) - f(x)| \leq k \cdot |\delta|$ for all x and δ. This property, also known as *Lipschitz continuity*, can be extended to entire programs with multiple inputs, and it has important applications in many parts of computer science, including control theory [26], dynamic systems [7], program analysis [9], and data privacy [13].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPCDSL '13, September 22, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2380-2/13/09…$15.00.
http://dx.doi.org/10.1145/2505351.2505353

In these applications, it is often necessary to *verify* a claim that a given function or program has a particular sensitivity. For instance, consider the following scenario from the differential privacy literature [13]: A querier Q asks the owner of a private database D to run some program f on D and then report the result $f(D)$ back to Q. It is known that this can be done safely, as long as a) the program f has a finite sensitivity, and b) the answer $f(D)$ is perturbed with a bit of random "noise", the amount of which depends on the sensitivity of f. Note that f can be an arbitrary program, and that the safety of the response depends on knowing the correct sensitivity; if the owner of the database underestimates the sensitivity of f, the result can be a serious privacy breach.

A related and equally important problem is to infer a *minimal* sensitivity for a given program. A program with sensitivity k also has sensitivity k' for any $k' > k$; thus, it is not incorrect to assign a program a sensitivity that is larger than necessary. However, high sensitivities come at a cost: for instance, a highly sensitive control function may be deemed less stable than it actually is, or a highly sensitive query on a private database may be perturbed with more noise than is strictly necessary, reducing the accuracy of the result. Thus, it is useful to be able to infer sensitivities that are as small as possible.

One promising approach to these problems is to use an specialized functional language featuring a *linear type system* to reason about sensitivity [11, 15, 22]. In this approach, the program types are decorated with annotations describing the sensitivity of each function; the typing rules are used to reason about the sensitivity of larger and larger subprograms in a compositional way. Thus, the sensitivity of the overall program can be verified from the known sensitivities of the language primitives. This approach is attractive because it produces a formal proof that a given program has a certain sensitivity; once the type-based verification succeeds, the program may be compiled and run by standard functional tools. However, requiring the programmer to make extensive sensitivity annotations throughout the program is tedious, error-prone, and hard to maintain, since a small change in some part may require the user to redo many annotations.

In this paper, we present a method that can infer and verify the sensitivity annotations in a completely automated way, even in the case where none are present. The key insight is to split the type-checking process into two steps: 1) a symbolic step that is largely analogous to standard type checking in functional programming languages, and 2) a numerical step that infers suitable values for the sensitivity annotations. The first step produces a set of constraints, such that each satisfying assignment corresponds to a valid upper bound on the program's sensitivity; the second step then finds solutions for these constraints by handing them to a standard SMT solver.

To demonstrate that our approach is practical, we have applied it to Fuzz [22], a programming language for differential privacy. In Fuzz, the generated constraints are nonlinear inequalities over the

$$
\begin{array}{llll}
\tau, \sigma & ::= & \mathbb{R} \,|\, !_r\sigma \multimap \tau \,|\, \sigma \oplus \tau & \text{(types, } r \in \mathbb{R}^\infty) \\
e & ::= & x \,|\, \mathrm{r} \,|\, \mathsf{f} \,|\, \lambda x : !_r\sigma.e \,|\, e_1\ e_2 \,| & \text{(expressions)} \\
 & & \mathbf{fix}\, e \,|\, \mathbf{inj}_i\ e \,| & \\
 & & \mathbf{case}\, e\, \mathbf{of}\, x \to e_l \mid y \to e_r & \\
\Gamma & ::= & \emptyset \,|\, \Gamma, x : !_r\sigma & \text{(environments)}
\end{array}
$$

Figure 1. *μFuzz* syntax

positive extended real numbers (the set $\{r \in \mathbb{R} \mid r \geq 0\} \cup \{\infty\}$, written $\mathbb{R}^\infty$ here). Such constraints are difficult to handle for most SMT solvers; fortunately, Z3 [12] is able to handle this theory thanks to its built-in support for nonlinear real arithmetic and its ability to integrate multiple theories. We have integrated our approach with the Fuzz compiler and measured running times on five sample queries from the differential privacy literature. Our results show that Z3 can solve all of the generated constraint systems in less than 200 milliseconds each and that it can find a good approximation to the minimal sensitivity in less than a second. In each case, the compiler produces a full set of sensitivity annotations automatically, without programmer intervention.

In summary, our contributions are: first, a language-based approach for inferring and/or verifying program sensitivity in a completely automatic way, based on generating constraints on possible sensitivity annotations that can then be solved by an SMT solver (§2 and §3); second, an application of our approach to Fuzz [22], a programming language for differential privacy, and the corresponding set of concrete constraints, which are nonlinear inequalities over the real numbers (§4); third, an implementation of our approach in the Fuzz compiler, which is based on the SMT solver Z3 (§5); and fourth, an experimental evaluation with five concrete programs from the differential privacy literature (§6).

2. The *μFuzz* language

As a vehicle for explaining our approach, we introduce *μFuzz*, a simple functional programming language with a linear type system able to reason about sensitivity. *μFuzz* is a proper subset of the language *Fuzz* [22], including just the features needed to illustrate how sensitivity analysis works. Extending the analysis to the full Fuzz language is straightforward and completely modular as witnessed by the implementation.

Types and terms. Figure 1 shows the formal grammar of *μFuzz*, a simple functional language with real-number constants. Real numbers appear in *μFuzz* in two places: both as constants and as annotations. In the examples, we will also use the arithmetic operations of addition and multiplication by a scalar, but we don't bother formalizing them here. A shown in [22], we can extend our language and distance definition to standard algebraic data types.

The types are a refinement of the classical simply typed lambda-calculus with some extra information about sensitivity. Specifically, in a function with type $!_r\sigma \multimap \tau$, the annotation r (drawn from $\mathbb{R}^\infty$) gives an upper bound on the function's sensitivity. When r is equal to ∞, it means that the sensitivity is not bounded. We define $r + \infty = \infty$, and $r \cdot \infty = \infty$. We write $\sigma \to \tau$ as a shorthand for $!_\infty\sigma \multimap \tau$. For the moment, sensitivity annotations on lambda-abstractions are explicit; we explain how to infer them in §3.

Typing. Figure 2 shows the rules for type assignment in *μFuzz*. The judgment $\Gamma \vdash e : \sigma$ can be read as "the expression e has type σ under the assumptions in environment Γ," where Γ records both the type of each free variable x appearing in e and an upper bound on the sensitivity of e to changes x; the binding $x : !_r\tau$ in Γ means that e can be assigned type σ assuming that x has type τ and that e is r-sensitive on x. Given $r \in \mathbb{R}^\infty$ and two environments Γ and Δ binding the same set of variables, we define the r-scaled sum operation $\Gamma + r \cdot \Delta$ as follows:

$$
\begin{array}{rcll}
\Gamma + r \cdot \Delta & = & \{x : !_{r_1 + r \cdot r_2}\sigma & \mid x : !_{r_1}\sigma \in \Gamma \wedge x : !_{r_2}\sigma \in \Delta\} \\
 & \cup & \{x : !_{r_1}\sigma & \mid x : !_{r_1}\sigma \in \Gamma \wedge x \notin \mathrm{dom}(\Delta)\} \\
 & \cup & \{x : !_{r \cdot r_2}\sigma & \mid x \notin \mathrm{dom}(\Gamma) \wedge x : !_{r_2}\sigma \in \Delta\}.
\end{array}
$$

The (Prim) rule assigns to the primitives in our language the corresponding predefined types. The (Const) rule assigns the type $\mathbb{R}$ to a real number constant r; since the expression r does not depend on the variables in Γ the rule does not require further assumptions. The (Var) rule says that the variable x has type τ, if x is assigned the type τ by the environment, and the sensitivity annotation on x is at least 1; this rule is motivated by the fact that the value of the expression x is indeed 1-sensitive to changes in x. The $(\multimap I)$ rule says that if e is r-sensitive in the free variable x, and if e yields a value of type σ when x is of type τ, then the lambda-abstraction $\lambda x : !_r\tau.e$ is an r-sensitive function of type $!_r\tau \multimap \sigma$. The most interesting rule is $(\multimap E)$: if e_1 is an r-sensitive function from τ to σ, and if e_2 has type τ, then 1) the application $e_1\ e_2$ has type σ, and 2) the sensitivity of $e_1\ e_2$ in each free variable x is the sum of r times the sensitivity of e_2 in x, and the sensitivity of e_1 in x (because each use of x in e_2 is "magnified" r times by the use that e_1 makes of its argument). A similar policy for sensitivities is used by the rule $(Case)$. A further aspect specific to this rule is that the expressions e_l and e_r are required to have the same sensitivities in their free variables. For this reason they are required to be typed in the same environment Γ.

The remaining constructors for expressions are introduced as typed combinators in Fig. 3. Note the fixpoint combinator **fix** for every type $((!_r\tau \multimap \sigma) \to (!_r\tau \multimap \sigma)) \to (!_r\tau \multimap \sigma)$. While it has unbounded sensitivity, its return type is an r-sensitive function $!_r\tau \multimap \sigma$. In the same table, we also present the types of some primitives[1] that are used in our examples. Other primitives can be added as described by the rule (Prim).

Semantics. To connect the type system with the operational semantics and justify the way the rules propagate sensitivities, we equip each type τ with a *metric* that defines a "distance" between expressions. The metric on the base type $\mathbb{R}$ is the standard distance metric on reals. Metrics for type constructors like function types are defined in terms of the metrics on their components; see [22]. A metric judgment of the form $\vdash e_1 \approx_m e_2 : \tau$ indicates that the expressions e_1 and e_2 are related at type τ, and that they are no more than distance m apart with respect to the metric on τ (where $m \in \mathbb{R}^{\geq 0} \cup \{\infty\}$ and $\infty + m = \infty$, $0 \cdot \infty = 0$ and $m \cdot \infty = \infty$ for $m \neq 0$). The type system is *sound* with respect to the metric if every expression of type $!_r\tau \to \sigma$ actually computes a r-sensitive function from τ to σ under a standard operational semantics for expressions. Formally (writing $e \hookrightarrow v$ to mean that expression e evaluates to value v):

Definition 1 (Metric Preservation [22])**.** *Let* $\vdash e : !_r\tau \to \sigma$ *and* $\vdash v_1 \approx_m v_2 : \tau$ *such that* $r \cdot m$ *is finite. If* $e\ v_1 \hookrightarrow w_1$*, then* $e\ v_2 \hookrightarrow w_2$ *and* $\vdash w_1 \approx_{r \cdot m} w_2$*. Note that* $\approx$ *is symmetric.*

Other approaches. A slightly different type system for sensitivity analysis is presented in [15]; any k-sensitive function is also semantically k'-sensitive for all $k' \geq k$; this induces a (fairly standard) subtyping relation on types.

The practical impact for type-checking an inference purposes comes from the different application rules. In a subtyping-based approach, inferring the minimal sensitivity of a function is enough.

[1] From a type checking perspective, combinators and primitives are handled similarly. However, from the operational point of view, primitives reduce to values in one step, while combinators reduce to expressions that may need further evaluation.

$$\frac{}{\Gamma \vdash \mathsf{r} : \mathbb{R}} \ (\text{Const}) \qquad \frac{\mathsf{f}\ r\text{-sensitive primitive} \quad \sigma \to \tau}{\Gamma \vdash \mathsf{f} : !_r\sigma \multimap \tau} \ (\text{Prim}) \qquad \frac{r \geq 1}{\Gamma, x : !_r\tau \vdash x : \tau} \ (\text{Var}) \qquad \frac{\Gamma, x : !_r\tau \vdash e : \sigma}{\Gamma \vdash \lambda x : !_r\tau . e : !_r\tau \multimap \sigma} \ (\multimap I)$$

$$\frac{\Gamma \vdash e_1 : !_r\tau \multimap \sigma \qquad \Delta \vdash e_2 : \tau}{\Gamma + r \cdot \Delta \vdash e_1\ e_2 : \sigma} \ (\multimap E) \qquad \frac{\Delta \vdash e : \sigma \oplus \tau \qquad x : !_r\sigma, \Gamma \vdash e_l : \mu \qquad y : !_r\tau, \Gamma \vdash e_r : \mu}{\Gamma + r \cdot \Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ x \to e_l \mid y \to e_r : \mu} \ (\text{Case})$$

Figure 2. Type assignment rules for *μFuzz*

$\mathbf{inj}_1$	$: \sigma \multimap \sigma \oplus \tau$		$(\forall \sigma\tau \text{ types})$
$\mathbf{inj}_2$	$: \tau \multimap \sigma \oplus \tau$		
$\mathbf{fix}$	$: (F \to F) \to F$	$F \equiv !_r\tau \multimap \sigma$	$(\forall r \in \mathbb{R}^{\geq 0})$
$(+)$	$: !_1\mathbb{R} \multimap !_1\mathbb{R} \multimap \mathbb{R}$		
$(r\ \cdot)$	$: !_r\mathbb{R} \multimap \mathbb{R}$		$(\forall r \in \mathbb{R}^{\geq 0})$

Figure 3. Types of combinators and primitive expressions

However, in a subtyping-free system all the uses of a given function must be taken into account in order to find the exact sensitivity, leading to global-scope constraints.

In the rest of this paper we explore the challenges generated by the subtyping-free version.

3. An Overview of Sensitivity Analysis

A well-typed function definition in the explicitly annotated language described above can thus be viewed as a proof that this function has a particular sensitivity. For instance, if we assume the primitive $2\cdot$ (with typing rule $\vdash 2\cdot : !_2\mathbb{R} \multimap \mathbb{R}$), then the expression $\lambda x : !_2\mathbb{R}\ .\ 2 \cdot x$ can be given the type $!_2\mathbb{R} \multimap \mathbb{R}$, expressing the fact that the underlying function $\lambda x.\ 2 \cdot x$ is 2-sensitive in its argument x. The same underlying function can be annotated differently to certify other (less precise) claims about its sensitivity. Indeed, for every $k \geq 2$ we can show $x : !_k\mathbb{R} \vdash 2 \cdot x : \mathbb{R}$ by using the $(\multimap E)$ rule with premises $\vdash 2\cdot : !_2\mathbb{R} \multimap \mathbb{R}$ and $x : !_i\mathbb{R} \vdash x : \mathbb{R}$, where $k = 2 \cdot i$ and $i \geq 1$ (the latter coming from the Var rule).

Finding such a proof—i.e., checking the well-typedness of a fully annotated program—requires some nontrivial reasoning on the part of the typechecker. For example, suppose we want to check that the following is a valid typing judgment:

$$x : !_5\mathbb{R} \vdash (\lambda f : !_1(!_3\mathbb{R} \multimap \mathbb{R}).\ f\,x)\ (\lambda y : !_3\mathbb{R}.\ 3 \cdot y + 0.5 \cdot x) : \mathbb{R}$$

The first step is to type the subexpressions. This requires splitting the annotation on x—i.e., finding two environments $x : !_{r_1}\mathbb{R}$ and $x : !_{r_2}\mathbb{R}$ that respectively allow us to prove

$$x : !_{r_1}\mathbb{R} \ \vdash\ \lambda f : !_1(!_3\mathbb{R} \multimap \mathbb{R}).\ f\,x \ :\ !_1(!_3\mathbb{R} \multimap \mathbb{R}) \multimap \mathbb{R}$$

and

$$x : !_{r_2}\mathbb{R} \ \vdash\ \lambda y : !_3\mathbb{R}.\ 3 \cdot y + 0.5 \cdot x \ :\ !_3\mathbb{R} \multimap \mathbb{R}$$

However, without further analyses on the subexpressions we do not know anything about the values r_1 and r_2 except that they should be such that $5 \geq r_1 + 1 \cdot r_2$, as required by the rule $(\multimap E)$. This example shows how even typechecking fully annotated expressions requires reasoning about constraints. A similar situation arises for languages like ML, whose polymorphic type systems require unification-based constraint solving, except that here the constraints are numerical.

In this particular example, the generated constraints are local and can be solved efficiently by a simple special-purpose algorithm (as was done in the original *Fuzz* implementation, for example). However, this form of constraint solving is not powerful enough to enable convenient programming with sensitivities. In particular, the task of providing complete annotations can be difficult and error-prone, so we would also like to be able to check the correctness of "partial programs" where some of the explicit sensitivity annotations are replaced by variables. We might, for example, want to verify a judgment like the following:

$$x : !_5\mathbb{R} \vdash (\lambda f : !_{i_1}(!_{i_2}\mathbb{R} \multimap \mathbb{R}).\ f\,x)\ (\lambda y : !_{i_3}\mathbb{R}.\ 3{\cdot}y{+}0.5{\cdot}x) : \mathbb{R}$$

This requires not only finding the split environments for x as described above but also finding values for i_1, i_2, and i_3 with the condition that i_2 must be equal to i_3. Note that now the only information we have from the rule $(\multimap E)$ is that $5 \geq r_1 + i_1 r_2$; unlike the previous case, this constraint is nonlinear and thus harder to solve. Similar constraints can arise when a program contains **case** expressions.

A natural further step is to ask not just for some set of annotations that makes an expression well typed, but for a set of *minimal* annotations. Consider a similar judgment:

$$x : !_j\mathbb{R} \vdash (\lambda f : !_{i_1}(!_{i_2}\mathbb{R} \multimap \mathbb{R}).2{+}f\,x)\ (\lambda y : !_{i_3}\mathbb{R}.3{\cdot}y{+}0.5{\cdot}x) : \mathbb{R}$$

To know the minimal value that j can take on, we have to find the least j satisfying the equation $j \geq r_1 + i_1 r_2$. (It turns out to be $\frac{7}{2}$.)

To formally define these problems, we enrich the possible annotations in expressions, types, and environments to include *sensitivity variables* $i, j, k, \ldots$ from a set $\mathcal{X}$. We will call any object containing such sensitivity variables *open*. Assigning values to variables corresponds to applying a *substitution*, a total function $\rho : \mathcal{X} \to \mathbb{R}^\infty$. As usual, substitutions on variables can be extended to open expressions, types and environments. A *constraint* C containing free sensitivity variables is *satisfiable* if there is a substitution ρ such that $\rho(C)$ is valid in the structure $\mathbb{R}^\infty$; we write $[\![\rho(C)]\!]$ in this case. Now, the two problems we address can be described formally as follows.

Problem 1 (Sensitivity Checking)**. Input:** *an open expression (or annotated program) e, and an open type σ.* **Output:** yes*, if there exists a substitution ρ such that* $\vdash \rho(e) : \rho(\sigma)$*;* no*, otherwise.*

Problem 2 (Sensitivity Minimization)**. Input:** *an open expression e (or annotated program), an open type σ, and a set of sensitivity variables* $\vec{i}$*.* **Output:** ρ sat*, if ρ is the minimal substitution with respect to* $\vec{i}$ *such that* $\vdash \rho(e) : \rho(\sigma)$*;* unsat *if there is no such substitution.*

Our approach to these problems is outlined in Figure 4. The input of the analysis is a *μFuzz* program where the sensitivity of every function expression is either provided by the user or left unspecified. The program is processed using a sensitivity analysis extending a standard typechecking. The result is a set of *sensitivity constraints* over $\mathbb{R}^\infty$ capturing all the program's sensitivity information. The resulting set of constraints is then submitted to an SMT solver. If the formula is satisfiable, the solver supplies the values of the unknown sensitivities (*solution set*). Otherwise, the solver provides a negative answer meaning that the user annotations are incorrect. In the case of a positive answer, a minimization engine can be used to generate a solution that is minimal (*minimal solution set*). This minimal solution provides the values of the missing

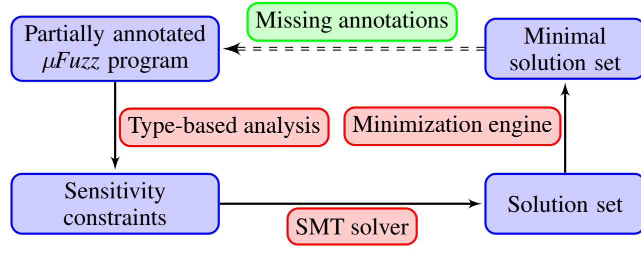

Figure 4. Our approach

annotations. If the minimal solution for a variable doesn't exist, the engine will stop at a value which is minimal with respect to a user-defined delta.

4. Constraint Solving and Generation

We describe our method for extracting constraints from a given μ *Fuzz* program, such that (1) the constraints are satisfiable iff the program is typable, and (2) every satisfying assignment directly corresponds to an upper bound on the program's sensitivity. For clarity, we present our method in two steps. First, we present a type-based constraint generation algorithm which, given a program, produces a set of constraints over $\mathbb{R}^\infty$; the minimal solutions to such constraints characterize the program sensitivities (§4.1). Second, we transform the constraints over $\mathbb{R}^\infty$ into a different representation that can be solved using SMT solvers such as Z3 (§4.2). We conclude the section with a discussion on complexity (§4.3).

4.1 Type-based constraint generation

Our constraint generation algorithm is shown in Figure 5. The algorithm takes as input an environment without sensitivity annotations and an open expression ($\Gamma \vdash e$). It returns a type σ, an updated environment Γ' containing the annotations $!_k$ for the original variables in Γ, and a constraint C. We write $\Gamma \vdash e \Rightarrow \sigma \dashv \Gamma' \mid C$ for an algorithm call. We write $\overline{\Gamma}$ for the environment resulting from removing all the sensitivity annotations from the bindings in Γ. Formally $\overline{\Gamma} = \{x : \sigma \mid (x : !_i\sigma) \in \Gamma\}$. The algorithm operates by traversing the expression syntax tree and accumulating information about the sensitivities in the set of constraints C; at the same time, it updates the environments to reflect the newly computed information about sensitivities.

The first two rules are standard and do not generate any interesting constraint. The rule (Var) , given a variable x, introduces a new sensitivity variable and a constraint to record that the sensitivity of x is greater or equal than one. The rule ($\multimap I$), given an abstraction $\lambda x : !_k\sigma.e$, runs the analysis on the inner expression e; such analysis will return a type σ and a sensitivity r which is checked against the annotations k.

The rule ($\multimap E$), given an application $e_1 \; e_2$, runs the analysis on e_1 and e_2. These two calls to the algorithm return two environments Γ and Δ containing different sensitivities annotations which need to be combined into a single environment representing $\Gamma + r \cdot \Delta$. To compute such an environmnet, we use the procedure $\mathrm{fresh}(\Sigma)$ that introduces a fresh sensitivity variable for each element of the initial environment Σ, and the procedure $\mathrm{merge}(\Gamma, r, \Delta)$ that produces a set of arithmetic constraints that link the new variables to the ones from Γ and Δ. Since the inferred types σ, σ' may contain different sensitivities, we impose the constraint $\sigma = \sigma'$, which assuming structural equality of σ and σ' requires all the sensitivities appearing in the types to be equal. If the types are not structurally equal, the term is not typable.

The rule ($\oplus E$) generalizes the approach described above. Given an expression **case** e **of** $x \to e_l \mid y \to e_r$, it runs the analysis on e, e_l and e_r. The first step to obtain the output environment is to find an upper bound for the two environments Δ_l and Δ_r inferred for the two branches of the case. This is again achieved by using the procedure $\mathrm{fresh}_{\vec{n}}(\Sigma)$ that introduces a fresh sensitivity variable in $\vec{n}$ for each element of the initial environment Σ, and the procedure $\sup_{\vec{n}}(\Delta_l, \Delta_r)$ that produces a set of arithmetic constraints that link the new variables in $\vec{n}$ to the ones from Δ_l and Δ_r. The second step is to find an upper bound for the sensitivities of x and y. This is obtained by introducing a new sensitivity variable r and by imposing the constraints $r \geq r_l$ and $r \geq r_r$. Using the results of these two steps, the environments $\mathrm{fresh}_{\vec{n}}(\Sigma)$ and Γ can be combined into a single one representing $\mathrm{fresh}_{\vec{n}}(\Sigma) + r \cdot \Gamma$ using the same technique described for the rule ($\multimap E$). Finally, we need to make sure that the return type in the two branches is the same. To obtain this we impose the constraint $\mu_l = \mu_r$ which assuming structural equality of μ_l and μ_r requires all the sensitivities appearing in the types to be equal.

Sensitivity annotations are only required to be greater or equal than the value of the actual sensitivity. For example a 1-sensitive function can be annotated with any sensitivity greater than 1, as any k-sensitive function is k'-sensitive for all $k' \geq k$.

Next, we relate the constraint generation algorithm with the type system presented in §2, which does not contemplate the use of sensitivity variables in the judgments. The algorithm in Figure 5 is *sound* in the sense that, if it generates a set of constraints C for a given expression e, then any satisfying assignment of C corresponds to a valid type for e. The algorithm is also *complete* in the sense that it always generates a satisfiable set of constraints C if the input is a well typed expression e. Moreover, for every sound sensitivity annotation of e there exists a solution of C with values corresponding to those of such annotation.

Definition 2 (Soundness)**.** *For all e and Γ, if $\Gamma \vdash e \Rightarrow \sigma \dashv \Gamma' \mid C$, then for all ρ such that $[\![\rho(C)]\!]$, we have $\rho(\Gamma') \vdash \rho(e) : \rho(\sigma)$.*

Definition 3 (Completeness)**.** *If $\Gamma \vdash e : \sigma$, then $\overline{\Gamma} \vdash e \Rightarrow \sigma \dashv \Gamma' \mid C$ and exists ρ such that $[\![\rho(C)]\!]$ and $\Gamma \geq \rho(\Gamma')$.*

4.2 Constraint Satisfiability and Minimization

The constraint generation algorithm in Figure 5 generates equalities and inequalities over a set of terms containing variables, constants in $\mathbb{R}^\infty$ and functions $\{+, *\}$. The structure of the generated constraints is described by the following grammar:

$$\mathsf{r} ::= \mathcal{X} \mid \mathbb{R}^\infty \qquad \mathsf{c} ::= \mathsf{r} \geq \mathsf{e} \mid \mathsf{r} = \mathsf{r} \qquad \mathsf{e} ::= \mathsf{r} \mid \mathsf{e} + \mathsf{r} * \mathsf{e} \tag{1}$$

Given the above constraints, we study how to solve them and how to minimize the value of a solution. The constraints generated in this paper follow a particular pattern, however they are not trivial to implement, making sense to use an already existing constraint solver. Not being tailored to a particular constraint solver algorithm also allowed to experiment more easily with different variations of the type system.

4.2.1 Constraint solving over the extended real line:

Our constraints are quite common in nonlinear real arithmetic with one important exception: we use real arithmetic extended to real line with ∞. This raises a difficulty, as most SMT solvers are restricted to solving over $\mathbb{R}$.

To overcome this limitation we encode terms r over $\mathbb{R}^\infty$ as a pair $(\mathbb{B} \times \mathbb{R})$. Assume a set of variables $\mathcal{X}_\mathbb{B}$ and $\mathcal{X}_\mathbb{R}$ for variables over $\mathbb{B}$ and $\mathbb{R}$. The translation I from terms over $\mathbb{R}^\infty$ to terms over $(\mathbb{B} \times \mathbb{R})$ is presented in Figure 6. Predicates over $\mathbb{R}^\infty$ are encoded using the same principles, using the translation function P. The boolean component determines when a term is infinity, and variables over $\mathbb{R}^\infty$ are represented as a pair of variables over $(\mathbb{B} \times \mathbb{R})$. Note that the translation is not a bijection, although it would be easy to do so.

$$\frac{\vec{m}\text{ fresh}}{\Gamma \vdash \mathsf{r} \Rightarrow \mathbb{R} \dashv \text{fresh}_{\vec{m}}(\Gamma) \mid \vec{m} \geq 0}\ (\text{Const}) \qquad \frac{\mathsf{f}\text{ typed }\sigma \qquad \vec{m}\text{ fresh}}{\Gamma \vdash \mathsf{f} \Rightarrow \sigma \dashv \text{fresh}_{\vec{m}}(\Gamma) \mid \vec{m} \geq 0}\ (\text{Prim})$$

$$\frac{r\text{ fresh} \qquad \vec{m}\text{ fresh}}{x:\sigma,\Gamma \vdash x \Rightarrow \sigma \dashv x : !_r\sigma, \text{fresh}_{\vec{m}}(\Gamma) \mid r \geq 1 \land \vec{m} \geq 0}\ (\text{Var}) \qquad \frac{x:\sigma,\Gamma \vdash e \Rightarrow \tau \dashv x : !_r\sigma, \Gamma' \mid C}{\Gamma \vdash \lambda x : !_k\sigma.e \Rightarrow !_k\sigma \multimap \tau \dashv \Gamma' \mid k \geq r \land C}\ (\multimap I)$$

$$\frac{\Sigma \vdash e_1 \Rightarrow !_r\sigma \multimap \tau \dashv \Gamma \mid C_1 \qquad \Sigma \vdash e_2 \Rightarrow \sigma' \dashv \Delta \mid C_2 \qquad \vec{m}\text{ fresh}}{\Sigma \vdash e_1\ e_2 \Rightarrow \tau \dashv \text{fresh}_{\vec{m}}(\Sigma) \mid C_1 \land C_2 \land \text{merge}_{\vec{m}}(\Gamma, r, \Delta) \land \sigma = \sigma'}\ (\multimap E)$$

$$\frac{\Sigma \vdash e \Rightarrow \sigma \oplus \tau \dashv \Gamma \mid C_e \qquad x:\sigma,\Sigma \vdash e_l \Rightarrow \mu_l \dashv x : !_{r_l}, \Delta_l \mid C_l \qquad y:\tau,\Sigma \vdash e_r \Rightarrow \mu_r \dashv y : !_{r_r}, \Delta_r \mid C_r}{\begin{array}{l}\Sigma \vdash \textbf{case}\, e\, \textbf{of}\ x \to e_l \Rightarrow \mu \dashv \text{fresh}_{\vec{m}}(\Sigma) \mid C_e \land C_l \land C_r \land r \geq \{r_l, r_r\} \\ \quad\ \mid y \to e_r \qquad \land\, \text{merge}_{\vec{m}}(\Gamma, r, \text{fresh}_{\vec{n}}(\Sigma)) \land \text{sup}_{\vec{n}}(\Delta_l, \Delta_r) \land \mu_l = \mu_r\end{array}}\ (\oplus E)$$

$$\begin{array}{rcl}
\text{fresh}_{\vec{m}}(\Gamma) & \equiv & \{x_i : !_{m_i}\sigma_i \mid (x_i : \sigma_i) \in \Gamma\} \\
\text{sup}_{\vec{m}}(\Gamma, \Delta) & \equiv & \{m_i \geq j_i \land m_i \geq k_i \mid (x_i : !_{j_i} : \sigma, x_i : !_{k_i}\sigma) \in (\Gamma, \Delta)\} \\
\text{merge}_{\vec{m}}(\Gamma, r, \Delta) & \equiv & \{m_i \geq j_i + r \cdot k_i \mid (x_i : !_{j_i}\sigma_i, x_i : !_{k_i}\sigma_i) \in (\Gamma, \Delta)\}
\end{array}$$

Figure 5. Constraint generation algorithm

$$\begin{array}{lcll}
I(\mathsf{r}) & = & (\mathsf{r.inf}, \mathsf{r.val}) & \mathsf{r} \in \mathcal{X} \quad (\mathsf{r.inf}, \mathsf{r.val}) \in (\mathcal{X}_{\mathbb{B}}, \mathcal{X}_{\mathbb{R}}) \\
I(\mathsf{r}) & = & (\mathsf{false}, \mathsf{r}) & \mathsf{r} \neq \infty \\
I(\infty) & = & (\mathsf{true}, _) & \\
I(\mathsf{r}_1 + \mathsf{r}_2) & = & (\mathsf{r}_1.\mathsf{inf} \lor \mathsf{r}_2.\mathsf{inf}, \mathsf{r}_1.\mathsf{val} + \mathsf{r}_2.\mathsf{val}) & (\mathsf{r_i.inf}, \mathsf{r_i.val}) = I(\mathsf{r_i}), \quad i \in \{1, 2\} \\
I(\mathsf{r}_1 * \mathsf{r}_2) & = & (\mathsf{r}_1.\mathsf{inf} \lor \mathsf{r}_2.\mathsf{inf}, \mathsf{r}_1.\mathsf{val} * \mathsf{r}_2.\mathsf{val}) &
\end{array}$$

$$\begin{array}{lcll}
P[\mathsf{r}_1 \Box \mathsf{r}_2] & = & P_\Box[I(\mathsf{r}_1) \Box I(\mathsf{r}_2)] & \Box \in \{=, \geq\} \\
P_\Box[(\mathsf{b}_1, \mathsf{r}_1) = (\mathsf{b}_2, \mathsf{r}_2)] & = & (\mathsf{b}_1 \land \mathsf{b}_2) \lor (\neg\mathsf{b}_1 \land \neg\mathsf{b}_2 \land \mathsf{r}_1 = \mathsf{r}_2) & \\
P_\Box[(\mathsf{b}_1, \mathsf{r}_1) \geq (\mathsf{b}_2, \mathsf{r}_2)] & = & \mathsf{b}_1 \lor (\neg\mathsf{b}_1 \land \neg\mathsf{b}_2 \land \mathsf{r}_1 \geq \mathsf{r}_2) &
\end{array}$$

Figure 6. Translation of the extended real numbers

The constraints resulting from the encoding are still challenging to solve because they mix boolean constraints with nonlinear real arithmetic, but, as we will show in §5, there is at least one current solver (Z3) that can handle them.

A *constraint satisfiability problem* for a set of constraints C consists in finding a substitution ρ such that $\rho(C)$ is a valid set of (in)equalities in $\mathbb{R}^\infty$ or alternatively in saying that no such substitution exists. It is easy to show that the encoding is correct and complete with respect to constraint solving, that is to say, an encoded constraint will have a solution with respect to the theory of the booleans and reals iff it has a solution over the theory of the extended positive real line, or formally $\forall C.(\exists \rho.[\![\rho(C)]\!] \iff \exists \rho_{(\mathbb{B}\times\mathbb{R})}[\![\rho_{(\mathbb{B}\times\mathbb{R})}(P[C])]\!])$.

4.2.2 Sensitivity minimization:

A *constraint minimization problem* for a set of constraints C and a variable x consists on finding the substitution ρ that satisfies C such that for every other valid substitution ρ', $\rho'(x) \geq \rho(x)$. Assume a formula $C(x)$ parametric on x, then we can express the minimization problem as the following first-order formula: $\exists x.(C(x) \land \forall y.(C(y) \to y \geq x)$.

This definition can be extended to several variables by extending the $\geq$ operation to take a variable set:

$$f(\vec{x_{min}}) = \exists \vec{x_{min}}.(C(\vec{x_{min}}) \land \forall \vec{x'_{min}}(C(\vec{x'_{min}}) \to \vec{x'_{min}} \vec{\geq} \vec{x_{min}}))$$

A formula may have no minimal solution.

The constraints generated by the sensitivity algorithm — characterized in Eq. (1) — feature some properties crucial for the minimization algorithm to work:

Definition 1. *Assume a constraint $C(\vec{x})$ with variables among $\vec{x}$ and a solution ρ for it. For every variable $x \in \vec{x}$, if $C(\vec{x}) \land x < \rho(x)$ is satisfiable, exists a solution ρ' such that $\rho'(\vec{x}) \leq \rho(\vec{x})$.*

Trying a lower value for a variable will never force other sensitivity variables to be assigned a higher value if the solver is complete. By this fact, we can use a one-variable-at-time minimization strategy, and the order in which we pick the variables to be minimized will not affect the final result. Indeed, in this version of the type system, the generated constraints will always have a minimal solution.[2]

4.3 Theoretical Complexity Bounds

First-order constraints (with conjunction and negation) over $\mathbb{R}$ with inequalities, equalities, sum and multiplication are generally referred to as Existential Theory of the Reals (ETR). Solving ETR constraints is a PSPACE problem with an NP-HARD lower bound [23]. This complexity result can be applied to our problem using the following variant of the reduction in 4.2. Every variable x over $\mathbb{R}^\infty$ can be represented by two variables $x.\mathsf{inf}$ and $x.\mathsf{val}$, this time both of type $\mathbb{R}$, where if $x.\mathsf{inf} = 0$ then x represents ∞ else x represents the real number $x.\mathsf{val}$. Operations such as sum and multiplication are as in §4.2. The resulting number of variables and constraints is polynomial in the size of the initial constraint set, so the PSPACE bound still holds.

The constraint minimization problem can be solved via *quantifier elimination*. This procedure takes a formula $f(x)$ and com-

[2] It should be noted that in general no minimal solution exists for constraints over reals.

Algorithm 1 Constraint Minimization

Input: $(x, lower, upper)$ variable to minimize, lower and upper bounds
(cs) set of satisfiable constraints
(IsDELTA) binary predicate that returns true iff its two arguments reached the desired distance/precision.

Output: minimal solution for the variable x in cs with precision depending on the predicate IsDELTA

```
1: function MIN(x, lower, upper, cs, IsDELTA)
2:     if IsDELTA(lower, upper) then
3:         return VALUEOF(x, cs)
4:     ADDCONSTRAINT(x ≤ (upper+lower)/2, cs)
5:     if IsSATISFIABLE(cs) then
6:         return MIN(x, lower, VALUEOF(x, cs), cs, IsDELTA)
7:     else
8:         REMOVELASTCONSTRAINT(cs)
9:         return MIN(x, (upper+lower)/2, upper, cs, IsDELTA)
```

putes an equivalent formula $f'(x)$ that does not contain quantifiers. The first order theory of reals (ETR with quantifiers) admits quantifier elimination, and, when the number of quantifier alternations is bounded, has complexity EXPTIME [4]. We can apply this technique to solve our constraints. The formula $f(\vec{x_{min}})$ of §4.2 can be encoded, as in the previous paragraph, as a formula $f_1(\vec{x_{min}}.\mathsf{val})$ of size polynomial in that of $f(\vec{x_{min}})$. After this encoding we can use the quantifier elimination and obtain an equivalent formula $f'(\vec{x_{min}}.\mathsf{val})$, where $\vec{x_{min}}.\mathsf{inf}$ is forced to have value greater than 0, of size exponential in that of $f_1(\vec{x_{min}}.\mathsf{val})$. Finally, $f'(\vec{x_{min}}.\mathsf{val})$ can be solved using the PSPACE procedure of [23], with final complexity EXPSPACE. If $f'(\vec{x_{min}}.\mathsf{val})$ does not admit any solution, the special case in which $\vec{x_{min}}.\mathsf{inf} = 0$ (the minimum value of $\vec{x_{min}}$ is ∞) is checked separately (this does not change the complexity).

5. Implementation: *Fuzz* and Z3

We have applied our sensitivity analysis method to *Fuzz* [22], a programming language for differential privacy that uses a sensitivity analysis to determine how much private information a given program could release. Our implementation is now part of the public release of Fuzz, available from the project web page [25].

After some experimenting with SMT solvers we decided to use Z3, which recently incorporated a complete solver (`nlsat`) for the existential theory of the reals [17]. This solver seamlessly integrates with boolean formulas; thus, Z3 supports the encoding of $\mathbb{R}^\infty$ we have described in §4.2.

Our implementation is based on Z3's Python bindings. The compiler applies some basic optimizations to the generated constraints. For instance, it removes trivial constraints like $\infty \geq i$ or $4 \geq 3$. The encoding of $\mathbb{R}^\infty$ interferes with Z3 own optimizations, as constraints like $i = 3$, which Z3 would eliminate using substitution propagation, are seen as a conditional boolean formula. There is room to experiment with compiler-side constraint optimization, for instance, compiler-side substitution propagation would halve the number of sensitivity variables finally sent to Z3, with the consequent runtime improvement.

As discussed in §4.2, the minimization problem can be expressed using quantifiers; however, Z3 does not yet support quantifier elimination for the minimization problem. Instead, we have implemented a simple minimization procedure based on binary search (see pseudocode in Algorithm 1). The basic idea is to use the constraint solver to successively narrow the range of the variable that is being optimized until the desired precision has been reached. Our procedure supports both relative and absolute precision parameters.

Problem	SC (ms)	SM (ms)	DB sens	Iterations ($p = 0.01$)	LoC
over40	33	266	1	10	47
income	50	424	4	11	81
age-histo	48	438	1	10	84
ipquery	53	368	1	10	74
k-means	67	662	6	12	126

Table 1. Running times for Sensitivity Checking (SC) and Sensitivity Minimization (SM) for DP algorithms. The Iterations column denotes the number of binary search iterations necessary to find the minimal sensitivity with error ± 0.01. The lines of code (LoC) omit those in the libraries for lists and bags (about 250 lines).

6. Evaluation

In this section we report results from an experimental evaluation of our approach, based on the extension to full Fuzz described in §5. The key metric we focus on is performance: in order to be practical for interactive applications (such as developing Fuzz programs), the analysis must complete within a reasonable amount of time. Since the time needed to generate the constraints is dwarfed by the time needed to solve them, we only report the latter, unless specified otherwise. Our experiments were performed on a commodity laptop—specifically, an Apple MacBook with a 1.7 GHz processor and 4 GB of memory.

6.1 Performance for realistic programs

To get a first impression of the performance for realistic Fuzz programs, we ran our analysis on the five example programs that are distributed with Fuzz. Each of these programs is motivated by a concrete application from the differential privacy literature. The programs take a database with private information (e.g., census data) as their single argument, and they return some aggregate result (e.g., the number of individuals in the database who have certain attributes).

We separately measured the time needed for sensitivity checking and for sensitivity minimization. For the former, we removed all sensitivity annotations except for the one on the database argument; for the latter, we removed *all* annotations and asked for the sensitivity of the database argument to be minimized, with a precision of ± 0.001. We additionally report the number of iterations for the binary search, as well as the number of lines of code (LoC) in each program.

Table 1 shows our results. All experiments completed within one second (in the case of sensitivity checking even within 100 ms), which is fast enough for interactive applications, such as program development. As a sanity-check for our implementation, we also verified that the minimum sensitivities matched the (known) sensitivities of the programs, as expected.

6.2 Scalability Analysis

Next, we measured how the analysis scales with respect to three key parameters: the sensitivity of the program, the size of the program in LoC, and the requested precision for minimization.

Sensitivity values. The cost of the binary search depends partly on the absolute value of the program's sensitivity—if this value is very big or very small, the search takes more time. To quantify this, we ran the analysis on a series of small programs with sensitivity values ranging between compute 0.3^9 and 2^9. We found that the running times were slightly affected by the sensitivitity values, althouth remaining below one second. Thus, at least for the small programs we tested, the runtime remains practical even for extreme sensitivities.

Program size. We expect the analysis time to increase with the program size as well, since larger programs tend to generate more constraints. To generate a number of comparable programs of dif-

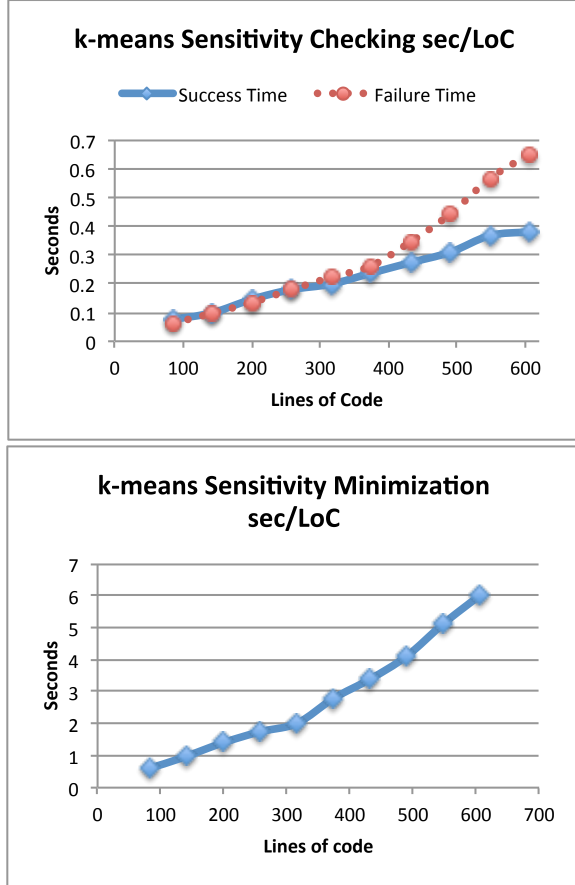

Figure 7. Running times for Sensitivity Checking and Sensitivity Minimization on k-means benchmark varying lines of code. We consider an absolute error of 0.01. The graph on the left also shows (dotted line) the running time required by the solver to say that the constraints are not satisfiable when inserting the wrong sensitivity annotation.

ferent sizes, we varied the number of iterations in the `k-means` program; the main loop calls 11 auxiliary functions, so each extra iteration adds 58 LoC once the loop is unrolled. We thus obtained ten programs $K_1, \ldots, K_{10}$ with sizes between 84 LoC (K_1) and 606 LoC (K_{10}). Figure 7 shows our results; the dotted line on the left is for the unsatisfiable constraints. For programs of 600 LoC or less, sensitivity checking takes at most 1 seconds until the check either succeeds or fails. For sensitivity minimization, at 600 LoC, the running time is of 6 seconds.

Precision. Finally, we quantified the impact of the requested precision on the runtime of the analysis. We re-ran all the tests from §6.1 with absolute precisions between 0.01 and 0.000001 and relative precisions between 1% and 0.0001%. We found that, the maximum runtime was 1350 ms, and therefore still practical.

Summary and discussion. The main conclusions from our evaluation results are as follows. *Sensitivity checking is usable:* In all of our experiments, the tool produced a (positive or negative) answer to the sensitivity checking problem in less than a second, for programs of up to 600 LoC. For the realistic programs (§6.1), the answer never took more than one second. *Higher-order functions are supported well:* Since complex types cause the application rule to generate more equality constraints, we initially expected the analysis to slow down as the order of functions increases (i.e., the depth of functions taking functions as arguments which themselves take functions...). However, in our experiments we did not find a significant slowdown, even with sixth-order functions. *Precision is not expensive:* The binary search provides a way to get high precision at a relatively small cost: increasing the precision by an order of magnitude adds only two extra iterations. *Big and small numbers take time:* Our minimization procedure is affected by the values of the sensitivity: when large programs have large sensitivity values, the minimization procedure can be slowed down. However, once SMT solvers add native support for minimization, performance should improve. *Large programs take time:* The minimization procedure takes longer than 5 seconds when the size of the program exceeds 500 LoC, even for relatively low precisions. However, practical queries are often smaller than this, and minimization need only be run once (after that, the identified values can be plugged into the program as annotations), so this may still be acceptable. Also, performance should improve once native support for minimization becomes available.

7. Related Work

Sensitivity analyses. Chaudhuri et al. in [8] and [9] study an automatic program analysis that can verify robustness of imperative programs. Their notion of robustness is same as program sensitivity: a program is K-robust if an ε-variation of the input can cause the output to vary by at most $\pm K\varepsilon$. In [9] they further extend this notion by parametrizing it over the "size" of the input (e.g. the number of elements in an array). Their technique performs a numerical analysis of the function computed by the program and then reasons about such function. Our approach differs from the one in [9] in several aspects. (1) Their analysis only considers terminating programs for which bounds on the number of loop iterations are known a priori. Our analysis does not impose a priori restrictions on the program that can be studied. (2) Their analysis allows to prove robustness of programs for which our analysis fails. Moreover, they are able to express the robustness in terms of sizes, rational numbers, and input values. This also entails the capability of reasoning in a more refined way about conditional branching. While Fuzz is not able to do this, the extended type system proposed in [15] can capture such variations to some extent. We plan on extending our analysis to this more complex type system. (3) Their analysis is able to verify but not infer a minimal sensitivity. And (4) their analysis is "mostly automated" due to proof obligations that are not always automatically solvable, while ours is fully automated.

Palamidessi and Stronati [20] recently proposed a constraint-based approach to compute the sensitivity of relational algebra queries. In particular, their analysis is able to compute the minimal sensitivity of wide range of queries. In contrast, the goal of our approach is to provide an upper bound on the sensitivity not of relational queries but for higher order functional programs.

Type systems and constraint solvers. Many previous works have reduced type inference and type checking to constraint satisfiability (see [19, 21] for an introduction). A recent language that uses an SMT solver for typechecking is Dminor [5]. This typed language uses subtyping and refinement types to process relational data. The SMT solver is used to check logical constraints corresponding to the refinement types and the subtyping relation. Similarly, the SAGE language [16] uses an SMT solver to check the validity of refinement types in a language with dynamic typing. This approach differs from ours since it considers symbolic constraints rather than numeric ones. Linear types are a key tool to support fine grained reasoning about resource management. Type checking and type inference for linear types usually involve some type decoration problem that can be reduced to a corresponding problem on integer constraints [1, 2]. These constraints can usually be solved efficiently using constraint solvers for integer programming. Dal Lago

and Petit [11] have recently proposed an inference algorithm for a type system for implicit complexity that combines linear indexed types with dependent types. Their algorithm generates integer constraints that can be solved using Why3 [6], a platform combining automatic and interactive solvers. ATS [10] is a language that combines linear types with automatic and interactive solvers for integer constraints. ATS helps reasoning about memory and pointers properties. However, linear types as used in ATS are not enough to reason about the sensitivity of programs. Our approach differs from all of these in the use of an SMT solver to decide constraints over the reals.

Differential privacy. Differential privacy [14] is one of the strongest privacy guarantees that has been proposed to date. Other linguistic tools besides Fuzz have been proposed to ensure differential privacy. PINQ [18] is an SQL-like differentially private query language embedded in C#; Airavat [24] is a MapReduce-based solution using a modified Java VM. CertiPriv [3]. is a machine-assisted framework—built on top of the Coq proof assistant—for reasoning about differential privacy from first principles. None of these tools provides a static sensitivity analysis method.

8. Conclusion

We have presented an approach for checking and inferring the sensitivity of functional programs. The approach is based on the type system of Fuzz, a higher-order functional programming language for differential privacy. Our technique generates constraints over positive real numbers extended with infinity so that each constraint solution is an upper bound for the program sensitivity. We used the SMT solver Z3 to solve such constraints and adopted our analysis to infer and verify the correct sensitivity of real world differential privacy programs.

We were able to scale our analysis for programs with more than 500 lines of code. We identified a few limitations in our method related to the minimization procedure. The use of an SMT solver makes our approach modular and extensible whenever new SMT solving techniques are introduced and we believe that future advances in the SMT community will address our analysis limitations.

We plan to extend our method to deal with the type system proposed in [15]. This extension would enable analysis of programs for which the sensitivity depends on some input values, for example array sizes.

Acknowledgments. We thank Leonardo de Moura for providing important insights and feedback on the use of Z3 in our tool. This research was supported by ONR grants N00014-09-1-0770 and N00014-12-1-0757, and NSF grant CNS-1065060. Loris D'Antoni was supported by the NSF Expeditions in Computing award CCF 1138996. Marco Gaboardi was supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 272487.

References

[1] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of ptime reducibility for system F terms: Type inference in dual light affine logic. *LMCS*, 3(4), 2007.

[2] Patrick Baillot and Martin Hofmann. Type inference in intuitionistic linear logic. In *PPDP*, pages 219–230, 2010.

[3] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2012.

[4] Saugata Basu. New results on quantifier elimination over real closed fields and applications to constraint databases. *J. ACM*, 46(4):537–555, July 1999.

[5] Gavin M. Bierman, Andrew D. Gordon, Ctlin Hricu, and David Langworthy. Semantic subtyping with an SMT solver. *Journal of Functional Programming*, 22:31–105, 0 2012.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011*, pages 53–64, 2011.

[7] Olivier Bournez, Daniel S. Graça, and Emmanuel Hainry. Robust computations with dynamical systems. In *MFCS'10*, LNCS, pages 198–208. Springer-Verlag, 2010.

[8] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. *SIGPLAN Not.*, 45(1):57–70, 2010.

[9] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving programs robust. In *SIGSOFT FSE*, 2011.

[10] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proc. ICFP*, pages 66–77, 2005.

[11] Ugo Dal Lago and Barbara Petit. Geometry of types. In *Proc. ACM POPL*, 2013.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[13] C. Dwork. Differential privacy: A survey of results. *Proc. TAMC*, 2008.

[14] Cynthia Dwork. Differential privacy. In *Proc. ICALP*, 2006.

[15] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proc. ACM POPL*, 2013.

[16] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *In Scheme and Functional Programming Workshop*, 2006.

[17] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In *IJCAR*, pages 339–354, 2012.

[18] Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD*, 2009.

[19] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[20] Catuscia Palamidessi and Marco Stronati. Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. In *QAPL*, pages 92–105, 2012.

[21] Franois Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[22] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, September 2010.

[23] James Renegar. A faster PSPACE algorithm for deciding the existential theory of the reals. In *Proc. FOCS'88*, pages 291–295. IEEE, 1988.

[24] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[25] `http://privacy.cis.upenn.edu`.

[26] G. Zames. Input-output feedback stability and robustness, 1959-85. *Control Systems, IEEE*, 16(3):61–66, June 1996.