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 \( \delta \). 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 [25], 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.

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.

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
\[ \tau, \sigma ::= \tau, \sigma \rightarrow \tau \]  
\[ e ::= x | e, f | \lambda x . e | e_1 \cdot e_2 \]  
\[ \text{fix } e \in \text{env} \]  
\[ \text{case } e \rightarrow e_1 | y \rightarrow e_r \]  
\[ \Gamma ::= 0 | \Gamma, x : \tau, \sigma \]  

Figure 1. \( \mu \text{Fuzz} \) 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 [3] and [1]; 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 [3]; 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 [9].

2. The \( \mu \text{Fuzz} \) language

As a vehicle for explaining our approach, we introduce \( \mu \text{Fuzz} \), a simple functional programming language with a linear type system able to reason about sensitivity. \( \mu \text{Fuzz} \) 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 \( \mu \text{Fuzz} \), a simple functional language with real-number constants. Real numbers appear in \( \mu \text{Fuzz} \) 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 \( !r, \sigma \rightarrow \tau \), the annotation \( r \) (drawn from \( \mathbb{R}^\infty \)) gives an upper bound on the function’s sensitivity. When \( r \) is equal to \( \infty \), it means that the sensitivity is not bounded. We define \( r + \infty = \infty \) and \( r \cdot \infty = \infty \). We write \( \sigma \rightarrow \tau \) as a shorthand for \( !r, \sigma \rightarrow \tau \). For the moment, sensitivity annotations on lambda-abstractions are explicit; we explain how to infer them in [4].

Typing. Figure 2 shows the rules for type assignment in \( \mu \text{Fuzz} \). The judgment \( \Gamma \vdash e : \sigma \) can be read as “the expression \( e \) has type \( \sigma \) under the assumptions in environment \( \Gamma \),” where \( \Gamma \) 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 \( \Gamma \) means that \( e \) can be assigned type \( \sigma \) assuming that \( x \) has type \( \tau \) and that \( e \) is \( r \)-sensitive on \( x \). Given \( r \in \mathbb{R}^\infty \) and two environments \( \Gamma \) and \( \Delta \) binding the same set of variables, we define the \( r \)-scaled sum operation \( \Gamma + r \cdot \Delta \) as follows:

\[ \Gamma + r \cdot \Delta = \{ x : !r_1 + r_2, \sigma \mid x : !r_1, \sigma \in \Gamma \land x : !r_2, \sigma \in \Delta \} \]
\[ \cup \{ x : !r_1, \sigma \mid x : !r_2, \sigma \not\in \Gamma \land \sigma \not\in \text{dom}(\Gamma) \} \]
\[ \cup \{ x : !r_2, \sigma \mid x : !r_1, \sigma \not\in \Gamma \land \sigma \not\in \text{dom}(\Gamma) \} \]

The (Prim) rule assigns to the primitives in our language the corresponding predefined types. The (Const) rule assigns the type \( \mathbb{R}^\infty \) to a real number constant \( r \); since the expression \( r \) does not depend on the variables in \( \Gamma \) the rule does not require further assumptions. The (Var) rule says that the variable \( x \) has type \( \tau \), if \( x \) is assigned the type \( \tau \) by the environment, and the sensitivity annotation on \( x \) at least is 1; this rule is motivated by the fact that the value of the expression \( x \) is indeed 1-sensitive to changes in \( x \). The (→) rule says that if \( e \) is \( r \)-sensitive in the free variable \( x \), and \( e \) yields a value of type \( \sigma \) when \( x \) of type \( \tau \), then \( \lambda x . e \) is an \( r \)-sensitive function of type \( !r, \tau \rightarrow \sigma \). The most interesting rule is (Case): if \( e_1 \) is an \( r \)-sensitive function from \( \tau \) to \( \sigma \), and if \( e_2 \) has type \( \tau \), then \( 1 \) the application \( e_1 \cdot e_2 \) has type \( \sigma \), and \( 2 \) the sensitivity of \( e_1 \cdot e_2 \) 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 \( e_1 \) and \( e_2 \) are required to have the same sensitivities in their free variables. For this reason they are required to be typed in the same environment \( \Gamma \).

The remaining constructors for expressions are introduced as typed combinators in Fig. 3. Note the fixpoint combinator \( \text{fix} \) for every type \( ((!r, \tau \rightarrow \sigma) 

\]  

Figure 2. Types and terms.

Semantic. To connect the type system with the operational semantics and justify the way the rules propagate sensitivities, we equip each type \( \tau \) with a metric that defines a “distance” between expressions. The metric on the base type \( \mathbb{R}^\infty \) 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 \( \tau \), and that they are no more than \( m \) apart with respect to the metric on \( \mathbb{R}^\infty \) (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 \( !r, \tau \rightarrow \sigma \) actually computes a \( r \)-sensitive function from \( \tau \) to \( \sigma \) under a standard operational semantics for expressions. Formally (writing \( e \leftrightarrow v \) to mean that expression \( e \) evaluates to value \( v \)):

Definition 1 (Metric Preservation [22]). Let \( \vdash e : !r, \tau \rightarrow \sigma \) and \( \vdash e_1 \approx_{m} e_2 \) : \( !r, \tau \rightarrow \sigma \) such that \( r \cdot m \) is finite. If \( e_1 \approx_{m} e_2 \), then \( e_2 \approx_{m} e_1 \). This is called 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 \( k \leq k' \); 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.

\[ \text{From a type checking perspective, combinators and primitives are handled similarly. However, from the operational point of view, primitives reduce to values in one step, while combinators reduce to expressions that may need further evaluation.} \]
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 (with typing rule $\Gamma : \tau \rightarrow \sigma$, then the expression $\lambda x : \sigma \rightarrow \tau \cdot \cdot \cdot$, where $\cdot \cdot \cdot$ is 2-sensitive in its argument $x$. The same underlying function can be annotated differently to make an expression well typed, but for a set of minimal annotations. Consider a similar judgment:

$$\Gamma \vdash x : \sigma \rightarrow \tau \rightarrow r \Rightarrow \cdot \cdot \cdot$$

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 ($\rightarrow 2$) 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:

$$\Gamma \vdash x : \sigma \rightarrow \tau \rightarrow r \Rightarrow \cdot \cdot \cdot$$

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 $\frac{5}{2}$.)

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 $X$. We will call any object containing such sensitivity variables open. Assigning values to variables corresponds to applying a substitution, a total function $\rho : X \rightarrow \mathbb{R}^\infty$. 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 $\rho$ 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 $\sigma$. *Output*: yes, if there exists a substitution $\rho$ such that $\Gamma \vdash \rho(e) : \rho(\sigma)$; no, otherwise.

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

Our approach to these problems is outlined in Figure 3.
4. Constraint Solving and Generation

We describe our method for extracting constraints from a given $\mu$ Fuzz program, such that (1) the constraints are satisfiable if 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 $\sigma$, an updated environment $\Gamma'$ containing the annotations $1_k$ for the original variables in $\Gamma$, and a constraint $C$. We write $\Gamma \vdash e \Rightarrow \sigma \vdash \Gamma' \mid C$ for an algorithm call. We write $\Gamma'$ for the environment resulting from removing all the sensitivity annotations from the bindings in $\Gamma$. Formally $\Gamma' := \{ x : \sigma \mid (x : 1) \sigma \in \Gamma \}$. 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 ($\text{Var}$), 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 ($\text{l"o}$), given an abstraction $\lambda x : \ell \sigma . e$, runs the analysis on the inner expression $e$; such analysis will return a type $\sigma$ and a sensitivity $r$ which is checked against the annotations $k$.

The rule ($\text{E}$), 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 $\Gamma$ and $\Delta$ containing different sensitivities annotations which need to be combined into a single environment representing $\Gamma + r \cdot \Delta$. To compute such an environment, we use the procedure $\text{fresh}(\Sigma)$ that introduces a fresh sensitivity variable for each element of the initial environment $\Sigma$, and the procedure $\text{merge}(\Gamma, r, \Delta)$ that produces a set of arithmetic constraints that link the new variables to the ones from $\Gamma$ and $\Delta$. Since the inferred types $\sigma, \sigma'$ may contain different sensitivities, we impose the constraint $\sigma = \sigma'$, which assuming structural equality of $\sigma$ and $\sigma'$ 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 $x \rightarrow e_1 \mid y \rightarrow e_r$, it runs the analysis on $e$, $e_1$ and $e_r$. The first step to obtain the output environment is to find an upper bound for the two environments $\Delta_i$ and $\Delta_r$ inferred for the two branches of the case. This is again achieved by using the procedure $\text{fresh}_{\sigma}(\Sigma)$ that introduces a fresh sensitivity variable in $\bar{n}$ for each element of the initial environment $\Sigma$, and the procedure $\text{sup}_{\sigma}(\Delta_i, \Delta_r)$ that produces a set of arithmetic constraints that link the new variables in $\bar{n}$ to the ones from $\Delta_i$ and $\Delta_r$. 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 $r \geq r_1$ and $r \geq r_2$. Using the results of these two steps, the environments $\text{fresh}_{\sigma}(\Sigma)$ and $\Gamma$ can be combined into a single one representing $\text{fresh}_{\sigma}(\Sigma) + r \cdot \Gamma$ using the same technique described for the rule ($\text{E}$). Finally, we need to make sure that the return type in the two branches is the same. To obtain this we impose the constraint $\mu_i = \mu_r$ which assuming structural equality of $\mu_i$ and $\mu_r$ 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 $\Gamma$, if $\Gamma \vdash e \Rightarrow \sigma \vdash \Gamma' \mid C$, then for all $\rho$ such that $[\rho(C)]$, we have $\rho(\Gamma') \vdash \rho(e) : \rho(\sigma)$.

Definition 3 (Completeness). If $\Gamma \vdash e : \sigma$, then $\Gamma \vdash e \Rightarrow \sigma \vdash \Gamma' \mid C$ and exists $\rho$ 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 $\{+, \ast\}$. The structure of the generated constraints is described by the following grammar:

$$ r ::= \mathcal{X} \mid \mathbb{R}^{\infty} \ c ::= r \geq e \mid r = r \ e ::= r + r \ast e \quad (1) $$

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 $\infty$. 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 $X_0$ and $X_0'$ 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
\[ \exists n \text{ fresh} \]
\[ \Gamma \vdash r \Rightarrow R \vdash \text{fresh}_n(\Gamma) \mid m \geq 0 \]  
(Cons)

\[ f \text{ typed } \sigma \vdash \text{fresh}_n(\Gamma) \mid m \geq 0 \]  
(Prim)

\[ r \text{ fresh} \quad \exists n \text{ fresh} \]
\[ x : \sigma, \Gamma \vdash x \Rightarrow \sigma + x : !\sigma, \text{fresh}_n(\Gamma) \mid r \geq 1 \land m \geq 0 \]  
(Var)

\[ x : \sigma, \Gamma \vdash e \Rightarrow r + x : !\sigma, \Gamma' \mid C \]
\[ \Gamma \vdash \lambda x : \sigma.e \Rightarrow \lambda x : \sigma \rightarrow r + \Gamma' \mid k \geq r \land C \]  
(\(\neg I\))

\[ \Sigma \vdash e_1 \Rightarrow \sigma \rightarrow r + \Gamma \mid C_1 \]
\[ \Sigma \vdash e_2 \Rightarrow r' + \Delta \mid C_2 \]
\[ \exists n \text{ fresh} \]
\[ \Sigma \vdash e_1, e_2 \Rightarrow r + \text{fresh}_n(\Sigma) \mid C_1 \land C_2 \land \text{merge}_n(\Gamma, r, \Delta) \land \sigma = \sigma' \]  
(\(\neg E\))

\[ \Sigma \vdash e \Rightarrow \sigma + r + \Gamma \mid C_e \]
\[ x : \sigma, \Sigma \vdash e_1 \Rightarrow \mu + x : !r_1, \Delta_1 \mid C_1 \]
\[ y : \tau, \Sigma \vdash e_r \Rightarrow \mu_r + y : !r_r, \Delta_r \mid C_r \]
\[ \exists n \text{ fresh} \]
\[ \Sigma \vdash \text{case } e \text{ of } x \to e_1 \Rightarrow \mu + \text{fresh}_n(\Sigma) \mid C_e \land C_1 \land C_r \land r \geq \{r_1, r_r\} \land y \to e_r \land \text{merge}_n(\Gamma, r, \Delta) \land \mu = \mu_r \]  
(\(\oplus E\))

\[ \text{fresh}_n(\Gamma) \equiv \{ x_i : !m_i, \sigma_i \mid (x_i : \sigma_i) \in \Gamma \} \]
\[ \text{sup}_n(\Gamma, \Delta) \equiv \{ m_i \geq j_i \land m_i \geq k_i \mid (x_i : !m_i, x_i : !k_i, \sigma) \in (\Gamma, \Delta) \} \]
\[ \text{merge}_n(\Gamma, r, \Delta) \equiv \{ m_i \geq j_i + r \cdot k_i \mid (x_i : !m_i, x_i : !k_i, \sigma) \in (\Gamma, \Delta) \} \]

**Figure 5.** Constraint generation algorithm

**Figure 6.** Translation of the extended real numbers

\[ \mathbb{R}^\infty \] are represented as a pair of variables over \((B \times \mathbb{R})\). Note that the translation is not a bijection, although it would be easy to do so.

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 \(\rho\) such that \(\rho(C)\) is a valid set of (in)equality in \(\mathbb{R}^\infty\), or alternatively in saying that no such substitution exists. It is easy to show that the encoding is correct with respect to arithmetic, but, as we will show in [5], there is at least one current solver (Z3) that can handle them.

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

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 [23]. Every variable \(x\) over \(\mathbb{R}^\infty\) can be represented by two variables \(x\text{.inf}\) and \(x\text{.val}\), this time both of type \(\mathbb{R}\), where if \(x\text{.inf} = 0\) then \(x\text{.val}\) represents \(\infty\) else \(x\text{.val}\) represents the real number \(x\text{.val}\). Operations such as sum and multiplication are as in [4.2]. The resulting number of variables and

---

\(^2\) It should be noted that in general no minimal solution exists for constraints over reals.
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 computes 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 \( \text{EXPTIME} \). We can apply this technique to solve our constraints. The formula \( f(x_{\text{min}}) \) of \( +2 \) can be encoded, as in the previous paragraph, as a formula \( f_1(x_{\text{min}}, \text{val}) \) of size polynomial in that of \( f(x_{\text{min}}) \). After this encoding we can use the quantifier elimination and obtain an equivalent formula \( f'(x_{\text{min}}, \text{val}) \), where \( x_{\text{min}}.\text{inf} \) is forced to have value greater than 0, of size exponential in that of \( f_1(x_{\text{min}}, \text{val}) \). Finally, \( f'(x_{\text{min}}, \text{val}) \) can be solved using the PSPACE procedure of \( +2 \), with final complexity \( \text{ExpSPACE} \). If \( f'(x_{\text{min}}, \text{val}) \) does not admit any solution, the special case in which \( x_{\text{min}}.\text{inf} = 0 \) (the minimum value of \( x_{\text{min}} \) is \( \infty \)) is checked separately (this does not change the complexity).

5. Implementation: Fuzz and Z3

We have applied our sensitivity analysis method to Fuzz \cite{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 \cite{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 \cite{17}. This solver seamlessly integrates with boolean formulas; thus, Z3 supports the encoding of \( \mathbb{R}^\infty \) we have described in \( +2 \).

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 \( \geq 0 \), of size exponential in that of \( f_1(x_{\text{min}}, \text{val}) \). Finally, \( f'(x_{\text{min}}, \text{val}) \) can be solved using the PSPACE procedure of \( +2 \), with final complexity \( \text{ExpSPACE} \). If \( f'(x_{\text{min}}, \text{val}) \) does not admit any solution, the special case in which \( x_{\text{min}}.\text{inf} = 0 \) (the minimum value of \( x_{\text{min}} \) is \( \infty \)) is checked separately (this does not change the complexity).

As discussed in \( +2 \), 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.

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.

## Algorithm 1 Constraint Minimization

**Input:** \((x, \text{lower}, \text{upper})\) 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\((x, \text{lower}, \text{upper}, \text{cs}, \text{IsDelta})\)
2. if IsDelta(\(\text{lower}, \text{upper}\)) then
3. return VALUEOF\((x, \text{cs})\)
4. ADDCONSTRAINT\((x \leq \frac{\text{upper} - \text{lower}}{2}, \text{cs})\)
5. if IsSATISFIABLE\((\text{cs})\) then
6. return MIN\((x, \text{lower}, \text{VALUEOF}(x, \text{cs}), \text{cs}, \text{IsDelta})\)
7. else
8. REMOVELASTCONSTRAINT\((\text{cs})\)
9. return MIN\((x, \frac{\text{upper} - \text{lower}}{2}, \text{upper}, \text{cs}, \text{IsDelta})\)

### 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 \( \pm 0.01 \). The lines of code (LoC) omit those in the libraries for lists and bags (about 250 lines).

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

6.1 Performance for realistic programs

To get a first impression of the performance of 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 \( \pm 0.001 \). 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 0.3\(^9\) and 2\(^9\). We found that the running times were slightly affected by the sensitivity values, although 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 different 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 the same as program sensitivity: a program is \(K\)-robust if an \(\varepsilon\)-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. Their 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.

**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 va-
lidity 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 [1] 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 [15] is an SQL-like differentially private query language embedded in C#. Airavat [24] is a MapReduce-based solution using a modified Java VM. CertiPriv [8], 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