Really Naturally Linear Indexed Type Checking

Arthur Azevedo de Amorim\textsuperscript{1}, Marco Gaboardi\textsuperscript{2}, Emilio Jesús Gallego Arias\textsuperscript{1}, Justin Hsu\textsuperscript{1}

\textsuperscript{1}University of Pennsylvania
\textsuperscript{2}University of Dundee

October 2, 2014
Check properties via types

- Type safety
- Parametricity
- Non-interference
In the beginning...

Check properties via types
• Type safety
• Parametricity
• Non-interference
In the beginning...

Check properties via types

- Type safety
- Parametricity
- Non-interference
Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy
Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy
Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy

- how robust?
- how likely?
Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy
More recently

Properties model quantitative information

- Numerical robustness: how robust?
- Probabilistic assertions: how likely?
- Differential privacy: how private?

Properties not just true or false
Typechecking quantitative languages is tricky

- May need to solve numeric constraints
- Typechecking may not be decidable
- May need heuristics to make typechecking practical
Typechecking quantitative languages is tricky

- May need to solve numeric constraints
- Typechecking may not be decidable
- May need heuristics to make typechecking practical

Our goal

- Design and implement a typechecking algorithm for DFuzz, a language for verifying differential privacy
The plan today

- A DFuzz crash course
- The problem with standard approaches
- Modifying the DFuzz language to ease typechecking
- Decidability and heuristics
Differential privacy [DMNS06]

- Rigorous definition of privacy for randomized programs
- Idea: random noise should “conceal” an individual’s data
- Quantitative: measure how private a program is
- Close connection to sensitivity analysis

The quantitative property
$R$-sensitive function
$R$-sensitive function
$R$-sensitive function
$R$-sensitive function
R-sensitive function
$R$-sensitive function

$f()$
A language for differential privacy

DFuzz [GHHNP13]

- Type system for differentially private programs
- Use linear logic to model sensitivity
- Combine with (lightweight) dependent types
Types

\[ \tau ::= \mathbb{N} \mid \tau \oplus \tau \mid \tau \otimes \tau \mid ! \tau \rightarrow \tau \mid \forall i. \tau \]
In a little more detail...

Types

\[ \tau ::= \mathbb{N} \; [R] \mid \tau \oplus \tau \mid \tau \otimes \tau \mid ! \; R \; \tau \rightarrow \tau \mid \forall i. \; \tau \]

Contexts

\[ \Gamma ::= \cdot \mid \Gamma, x : [R] \; \tau \]
In a little more detail...

Types

\[ \tau ::= \mathbb{N} \ [R] \ | \ \tau \oplus \tau \ | \ \tau \otimes \tau \ | \ !_{R} \ \tau \rightarrow \tau \ | \ \forall i. \ \tau \]

Contexts

\[ \Gamma ::= \cdot \ | \ \Gamma, x : [R] \ \tau \]

Typing judgment

\[ \Gamma \vdash e : \tau \]
Sensitivity reading

- Functions \( R \tau_1 \rightarrow \tau_2 \): \( R \)-sensitive functions
- Changing input by \( d \) changes output by at most \( R \cdot d \)
Sensitivity analysis

$R$-sensitive function

$R d < f d$
Sensitivity reading

- Functions $!_{R \tau_1} \circ \tau_2$: $R$-sensitive functions
- Changing input by $d$ changes output by at most $R \cdot d$
Sensitivity reading

- Functions $!_{R \tau_1} \to \tau_2$: $R$-sensitive functions
- Changing input by $d$ changes output by at most $R \cdot d$

Subtyping

- “A 1-sensitive function is also a 2-sensitive function”
- Subtyping: weaken sensitivity bound

\[
!_{R \tau} \to \tau_2 \sqsubseteq!_{R' \tau_1} \to \tau_2 \quad \text{if} \quad R \leq R'
\]
In a little more detail...

Types

\[ \tau ::= \mathbb{N} \mid [R] \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_R \tau \rightarrow \tau \mid \forall i. \tau \]

Contexts

\[ \Gamma ::= \cdot \mid \Gamma, x : [R] \tau \]

Typing judgment

\[ \Gamma \vdash e : \tau \]
Types

\[ \tau ::= \mathbb{N} \mid \tau \oplus \tau \mid \tau \otimes \tau \mid ! \tau \rightarrow \tau \mid \forall i. \tau \]

Contexts

\[ \Gamma ::= \cdot \mid \Gamma, x : \mathbb{N} \]

Typing judgment

\[ \Gamma \vdash e : \tau \]
The sensitivity language

Grammar

\[ R ::= \ i_R \mid i_N \mid \mathbb{R} \mid R + R \mid R \cdot R \]
The sensitivity language

Grammar

\[ R ::= i_R | i_N | \mathbb{R} | R + R | R \cdot R \]

variables over real/naturals

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)

What does this mean for typechecking?

- Sensitivities are polynomials over reals and naturals
- How to check subtyping?
The sensitivity language

Grammar

\[ R ::= i_R \mid i_N \mid \mathbb{R} \mid R + R \mid R \cdot R \]

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)
Types

\[ \tau ::= \mathbb{N}^{[R]} | \tau \bigoplus \tau | \tau \otimes \tau | !^R \tau \rightarrow \tau | \forall i. \tau \]

Contexts

\[ \Gamma ::= \cdot | \Gamma, x :^{[R]} \tau \]

Typing judgment

\[ \Gamma \vdash e : \tau \]
The sensitivity language

Grammar

\[ R ::= i_R | i_N | R | R + R | R \cdot R \]

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)
The sensitivity language

Grammar

\[ R ::= i_\mathbb{R} \mid i_\mathbb{N} \mid \mathbb{R} \mid R + R \mid R \cdot R \]

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)

What does this mean for typechecking?

- Sensitivities are **polynomials** over reals and naturals
- How to check subtyping?
Sensitivity reading

- Functions $!_{R \tau_1} \to \tau_2$: $R$-sensitive functions
- Changing input by $d$ changes output by at most $R \cdot d$

Subtyping

- “A 1-sensitive function is also a 2-sensitive function”
- Subtyping: weaken sensitivity bound

\[ !_{R \tau} \to \tau_2 \sqsubseteq !_{R' \tau_1} \to \tau_2 \quad \text{if} \quad R \leq R' \]
Sensitivity reading

- Functions \( !_{R\tau_1} \rightarrow \tau_2 \): \( R \)-sensitive functions
- Changing input by \( d \) changes output by at most \( R \cdot d \)

Subtyping

- “A 1-sensitive function is also a 2-sensitive function”
- Subtyping: weaken sensitivity bound

\[
!_{R\tau} \rightarrow \tau_2 \sqsubseteq !_{R'\tau_1} \rightarrow \tau_2 \quad \text{if} \quad R \leq R'
\]
Assume

- Can extract type skeleton from term
- Given annotated term, compute best type
Assume

- Can extract type **skeleton** from term
- Given annotated term, compute **best** type

**Annotations**
- We need: fully annotated argument types of all functions
- Other more minor annotations
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type w.r.t. subtyping
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type w.r.t. subtyping
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

\[
\begin{align*}
\text{!} & \quad \tau_1 \quad \text{⇒} \quad \tau_2 \\
\end{align*}
\]
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

\[ \tau_1 \rightarrow \tau_2 \]

annot.
Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions
The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

\[ \tau_1 \rightarrow^o \tau_2 \]

! ?? no annot. annot. no annot.
The typechecking problem

**Assume**

- Can extract type skeleton from term
- Given annotated term, compute best type

**Annotations**

- We need: fully annotated argument types of all functions

\[
! \ \ ?? \quad \tau_1 \quad \rightarrow \quad \tau_2
\]

- Other more minor annotations
The typechecking problem

Input

- Annotated term $e$
- Annotated context skeleton $\Gamma^\bullet$:

\[
\begin{align*}
  x : & \quad ?? \quad \tau
\end{align*}
\]
The typechecking problem

Input

• Annotated term $e$
• Annotated context skeleton $\Gamma^\bullet$:

\[ x : \text{??} \quad \tau \]

no annot.
The typechecking problem

Input

- Annotated term $e$
- Annotated context skeleton $\Gamma^\bullet$:

\[
x : \text{??} \quad \tau \quad \text{annotate.}
\]

\[
\text{no annotate.}
\]
The typechecking problem

Input

• Annotated term $e$
• Annotated context skeleton $\Gamma^\bullet$:

Output

• Type $\tau^*$ and context $\Gamma$ with $\Gamma \vdash e : \tau^*$
• Most precise context and type (with respect to subtyping)
“Bottom-up” typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type
"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

\[ \Gamma \vdash e_1 : !_{R \sigma} \rightarrow \tau \quad \Delta \vdash e_2 : \sigma \]

\[ \Gamma + R \cdot \Delta \vdash e_1 \ e_2 : \tau \]

1. Given \((e_1 \ e_2, \Gamma^\bullet)\)
“Bottom-up” typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

\[ \Gamma \vdash e_1 : !R \sigma \to \tau \quad \Delta \vdash e_2 : \sigma \]
\[ \Gamma + R \cdot \Delta \vdash e_1 e_2 : \tau \]

1. Given \((e_1, e_2, \Gamma^\bullet)\)
2. Call typechecker on \((e_1, \Gamma^\bullet)\), get \((!R \sigma \to \tau, \Gamma)\)
“Bottom-up” typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

\[
\begin{array}{c}
\Gamma \vdash e_1 : !R\sigma \rightarrow \tau \\
\Delta \vdash e_2 : \sigma \\
\hline
\Gamma + R \cdot \Delta \vdash e_1 e_2 : \tau
\end{array}
\]

1. Given \((e_1, e_2, \Gamma^\bullet)\)
2. Call typechecker on \((e_1, \Gamma^\bullet)\), get \((!R\sigma \rightarrow \tau, \Gamma)\)
3. Call typechecker on \((e_2, \Delta^\bullet)\), get \((\sigma', \Delta)\)
“Bottom-up” typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

\[
\Gamma \vdash e_1 : ! R \sigma \rightarrow \tau \\
\Delta \vdash e_2 : \sigma
\]

\[
\Gamma + R \cdot \Delta \vdash e_1 \, e_2 : \tau
\]

1. Given \((e_1 \, e_2, \Gamma^*)\)
2. Call typechecker on \((e_1, \Gamma^*)\), get \((! R \sigma \rightarrow \tau, \Gamma)\)
3. Call typechecker on \((e_2, \Delta^*)\), get \((\sigma', \Delta)\)
4. Check \(\sigma' \sqsubseteq \sigma\), output \((\tau, \Gamma + R \cdot \Delta)\)
A problem with the bottom-up approach

- Some DFuzz rules have form

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \\
\Gamma \vdash \ldots : \ldots
\]
A problem with the bottom-up approach

- Some DFuzz rules have form

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \\
\Gamma \vdash \cdots : \cdots
\]
A problem with the bottom-up approach

- Some DFuzz rules have form

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2
\]

\[
\Gamma \vdash \ldots : \ldots
\]
A problem with the bottom-up approach

- Some DFuzz rules have form
  \[ \Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \]
  \[ \Gamma \vdash \ldots : \ldots \]

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
“Minimal” types?

A problem with the bottom-up approach

- Some DFuzz rules have form

\[
\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \ldots : \ldots}
\]

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
- But what context do we output?
“Minimal” context?

First try

- Have $x : [R_1] \sigma$ and $x : [R_2] \sigma$
- Most precise context should be $x : [\max(R_1, R_2)] \sigma$
- But DFuzz doesn’t have $\max(R_1, R_2)$...
The sensitivity language

Grammar

\[ R ::= \ i_R \mid i_N \mid R \mid R + R \mid R \cdot R \]
First try

- Have $x : [R_1] \sigma$ and $x : [R_2] \sigma$
- Most precise context should be $x : [\text{max}(R_1, R_2)] \sigma$
- But DFuzz doesn’t have $\text{max}(R_1, R_2)$...
First try

- Have $x : [R_1] \sigma$ and $x : [R_2] \sigma$
- Most precise context should be $x : [\max(R_1, R_2)] \sigma$
- But DFuzz doesn’t have $\max(R_1, R_2)$...

Max of two polynomials may not be polynomial!
The idea: enrich DFuzz

EDFuzz: E(xtended) DFuzz

- Sensitivity language in DFuzz is “incomplete” for typechecking
- Add constructions like \( \text{max}(R_1, R_2) \) to sensitivity language
- Typecheck EDFuzz programs instead
EDFuzz: *E*(xtended) D*Fuzz*

- Sensitivity language in DFuzz is “incomplete” for typechecking
- Add constructions like \( \max(R_1, R_2) \) to sensitivity language
- Typecheck EDFuzz programs instead

**Relation with DFuzz**

- Extension: all DFuzz programs still valid EDFuzz programs
- Preserve metatheory
- Bottom-up typechecking simple, works
Previously problematic rule

\[
\begin{align*}
\Gamma &\vdash e_1 : \sigma_1 & \Gamma &\vdash e_2 : \sigma_2 \\
\hline
\Gamma &\vdash \ldots : \ldots
\end{align*}
\]
Previously problematic rule

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \\
\Gamma \vdash \ldots : \ldots
\]

Now: no problem

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
How does this fix the problem?

Previously problematic rule

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \\
\hline
\Gamma \vdash \ldots : \ldots
\]

Now: no problem

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
Previously problematic rule

\[
\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2
\]

\[
\Gamma \vdash \ldots : \ldots
\]

Now: no problem

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
How does this fix the problem?

Previously problematic rule

\[ \Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \]
\[ \Gamma \vdash \ldots : \ldots \]

Now: no problem

- Running algorithm gives \((\sigma_1, \Gamma_1)\) and \((\sigma_2, \Gamma_2)\)
- For
  \[ x : [R_1] \sigma \in \Gamma_1 \quad \text{and} \quad x : [R_2] \sigma \in \Gamma_2, \]
  put \( x : [\max(R_1, R_2)] \sigma \) in output context
How does this fix the problem?

Previously problematic rule

\[ \Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \]

\[ \Gamma \vdash \ldots : \ldots \]

Now: no problem

- Running algorithm gives (\(\sigma_1, \Gamma_1\)) and (\(\sigma_2, \Gamma_2\))
- For
  \[ x : [R_1] \sigma \in \Gamma_1 \quad \text{and} \quad x : [R_2] \sigma \in \Gamma_2, \]
  put \( x : [\max(R_1, R_2)] \sigma \) in output context
- Return \( \max(R_1, R_2) \) as context
Bad news

- Must check inequalities over reals and natural polynomials
- Subtype relation is undecidable
- Even checking validity of derivations is undecidable
- Problem for both DFuzz and EDFuzz
Bad news

- Must check inequalities over reals and natural polynomials
- Subtype relation is undecidable
- Even checking validity of derivations is undecidable
- Problem for both DFuzz and EDFuzz

Good news

- Constraint solvers are pretty good in practice
- Typical DFuzz programs rely on easy constraints
Checking the constraints

Special structure of constraints

- Allow standard (DFuzz) annotations only
- Subtyping only needs to check

\[ R \geq R^*, \]

where \( R \) is a DFuzz sensitivity and \( R^* \) is a EDFuzz sensitivity

- \( R \) understood by standard numeric solvers
- \( R^* \) has extended terms like \( \max(R_1, R_2), \ldots \)
Idea: eliminate extended terms

- Change $R \geq \max(R_1^*, R_2^*)$ to

$$R \geq R_1^* \land R \geq R_2^*$$

- Recursively eliminate comparisons $R \geq R^*$
- Similar technique for other new sensitivity constructions
It works!

- Dispatches numeric constraints to Why3
- Typechecks examples from the DFuzz paper with no problems
- Annotation burden light on these examples
Lessons learned

- Typechecking with quantitative constraints is tricky
- Numeric solvers are quite good, even for undecidable problems
- Minor details in original language can have huge effects on how easy it is to use standard solvers
- Keep typechecking in mind!
Lessons learned

- Typechecking with quantitative constraints is tricky
- Numeric solvers are quite good, even for undecidable problems
- Minor details in original language can have huge effects on how easy it is to use standard solvers
- Keep typechecking in mind!

Open questions

- Does this technique of “completing” a language to ease typechecking apply to other quantitative type systems?
- Can we remove the argument type annotation in functions?
Really Naturally Linear Indexed Type Checking

Arthur Azevedo de Amorim\textsuperscript{1}, Marco Gaboardi\textsuperscript{2}, Emilio Jesús Gallego Arias\textsuperscript{1}, Justin Hsu\textsuperscript{1}

\textsuperscript{1}University of Pennsylvania
\textsuperscript{2}University of Dundee

October 2, 2014
Problematic rule

\[
\Gamma \vdash e : \sigma \quad i \text{ fresh in } \Gamma
\]

\[
\Gamma \vdash \Lambda i : \kappa. \ e : \forall i : \kappa. \sigma
\]

Avoidance problem

- Running typechecker on \((e, \Gamma^\bullet)\) yields \((\sigma, \Gamma)\)
- For \(x : [R] \sigma \in \Gamma\), want smallest \(R^*\) bigger than \(R\) but independent of \(i\)
- Again: \(R^*\) may lie outside sensitivity language
- Add construction \(\text{sup}(R, i)\) to EDFuzz