

The Checker Framework: pluggable static analysis for Java

<http://CheckerFramework.org/>



Werner Dietl
University of Waterloo
<https://ece.uwaterloo.ca/~wdietl/>



Joint work with Michael D. Ernst and many others.

Bug Evolution

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

92

9/9

0800 Antran started
1000 " stopped - antran ✓ { 1.2700 9.037 847 025
13"uc (032) MP - MC ~~1.30476415~~ 9.037 846 995 conduct
033) PRO 2 2.130476415
conduct 2.130476415

Relays 6-2 in 033 failed special speed test
in Relay 10.000 test.

Relay
2145
Relay 3376

1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

1600 Antran started.
1700 closed down.

Bug Evolution

Photo # NH 96566-KN (Color) First Comp

92

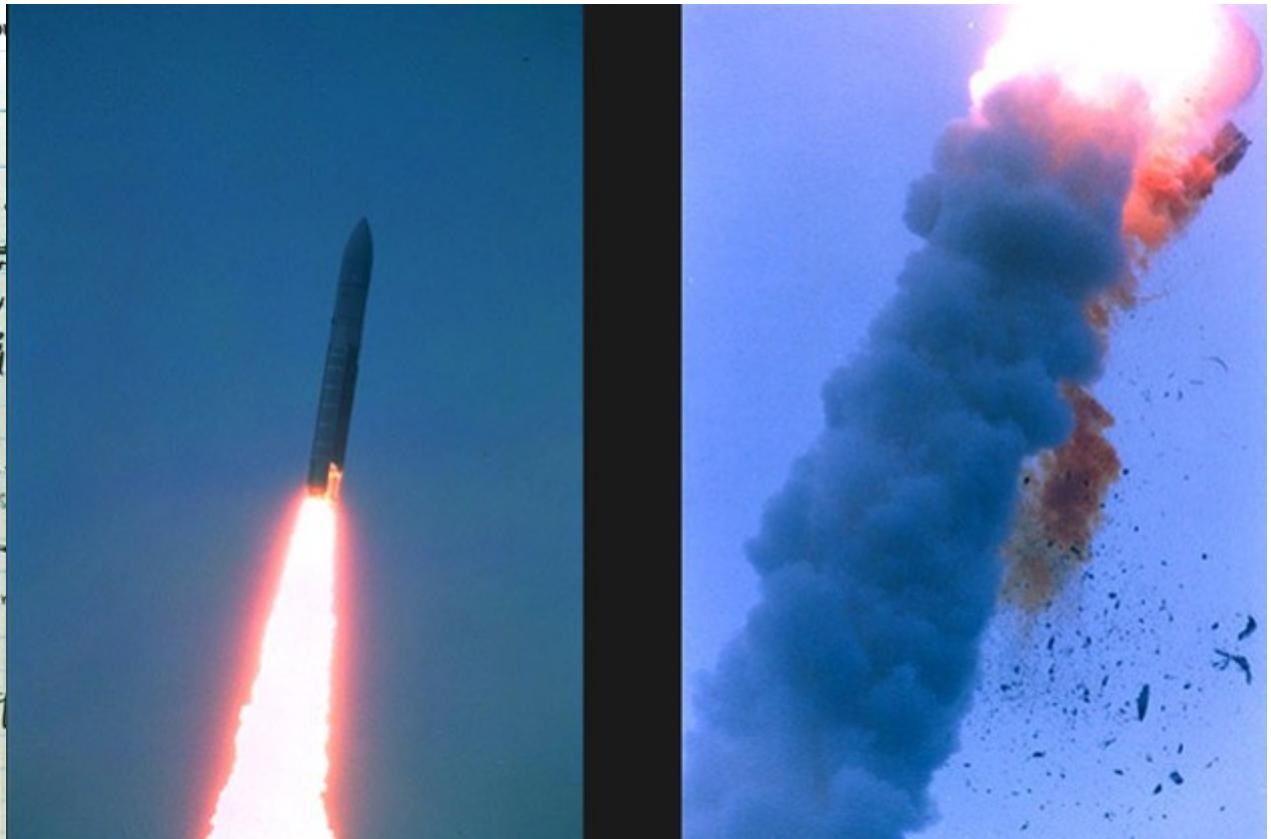
9/9

0800 Antennas started
1000 " stopped - antenna ✓
13" w/c (032) MP - MC ~~± 1304~~
033 PRO 2 2.1304
conduct 2.1306
Relays 6-2 in 033 failed
in Relay

1100 Started Cosine Tape (Sin
1525 Started Multi Adder Tes

1545 Relay
(moth).

1630 Antennas started.
1700 closed down.



Cost of software failures

\$312 billion per year global cost of software bugs (2013)

\$300 billion dealing with the Y2K problem

\$650 million loss by NASA Mars missions in 1999; unit conversion bug

\$500 million Ariane 5 maiden flight in 1996; 64 bit to 16 bit conversion bug

\$440 million loss by Knight Capital Group Inc. in 30 minutes in August 2012

Outline

- Pluggable type-checking
- Architecture overview
 - Java 8 syntax for Type Annotations
 - Dataflow Framework
 - Checker Framework
- Our experience
 - Case Studies
 - SPARTA: Android Security
 - VeriGames: Crowd-sourced Verification Games
- Writing your own type system

Java's type system is too weak

Type checking prevents many errors

```
int i = "hello"; // error
```

Type checking doesn't prevent enough errors

```
System.console().readLine();
```

```
Collections.emptyList().add("one");
```

```
dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = "hello"; // error
```

Type checking can prevent errors

NullPointerException

```
System.console().readLine();
```

```
Collections.emptyList().add("one");
```

```
dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = "hello"; // error
```

Type checking doesn't prevent enough errors

```
System.UnsupportedOperationException
```

```
Collections.emptyList().add("one");
```

```
dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = "hello"; // error
```

Type checking doesn't prevent enough errors

```
System.console().readLine();
```

Collections SQL Injection Attacks!

```
dbStatement.executeQuery(userData);
```

Static types: not expressive enough

Null pointer exceptions

```
String op(Data in) {  
    return "transform: " + in.getF();  
}  
  
...  
String s = op(null);
```

Many other properties can't be expressed

Prevent null pointer exceptions

Type system that statically guarantees that the program only dereferences known non-null references

Types of data

`@NonNull` reference is never null

`@Nullable` reference may be null

Null pointer exception

```
String op(Data in) {  
    return "transform: " + in.getF();  
}  
...  
String s = op(null);
```

Where is the error?

Solution 1: Restrict use

```
String op(@NotNull Data in) {  
    return "transform: " + in.getF();  
}  
  
...  
String s = op(null);      // error
```

Solution 2: Restrict implementation

```
String op(@Nullable Data in) {  
    return "transform: " + in.getF();  
    // error  
}  
  
...  
String s = op(null);
```

Benefits of type systems

- **Find bugs** in programs
- Guarantee the **absence of errors**
- **Improve documentation**
- Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
- Reduce number of run-time checks
- Possible negatives:
 - Must write the types (or use type inference)
 - False positives are possible (can be suppressed)

Extended type systems

- Null pointer exceptions (ICSE SEIP'11)
- Energy consumption (PLDI'11)
- Unwanted side effects (OOPSLA'04, '05, '12, ECOOP'08, ESEC/FSE'07)
- Unstructured heaps (ECOOP'07, '11, '12)
- Malformed input (FTfJP'12)
- UI actions not on event thread (ECOOP'13)
- Information leakage (CCS'14)

Extended type systems

- Null pointer exceptions (ICSE SEIP'11)
- Energy consumption (PLDI'11)
- Unwanted side effects (OOPSLA'04, '05, '12, ECOOP'08, ESEC/FSE'07)
- Unstructured heaps (ECOOP'07, '11, '12)
- Malformed input (FTfJP'12)
- UI actions not on event thread (ECOOP'13)
- Information leakage (CCS'14)

Theory

Practice

Decades!

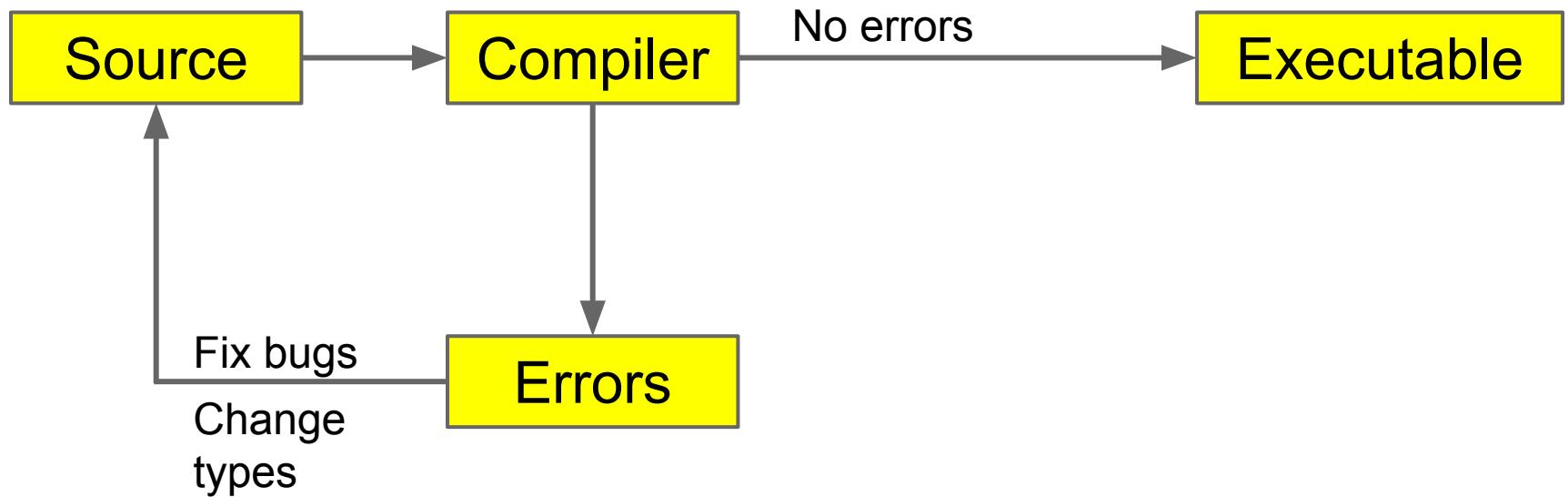
The Checker Framework

A framework for pluggable type checkers
“Plugs” into the OpenJDK compiler

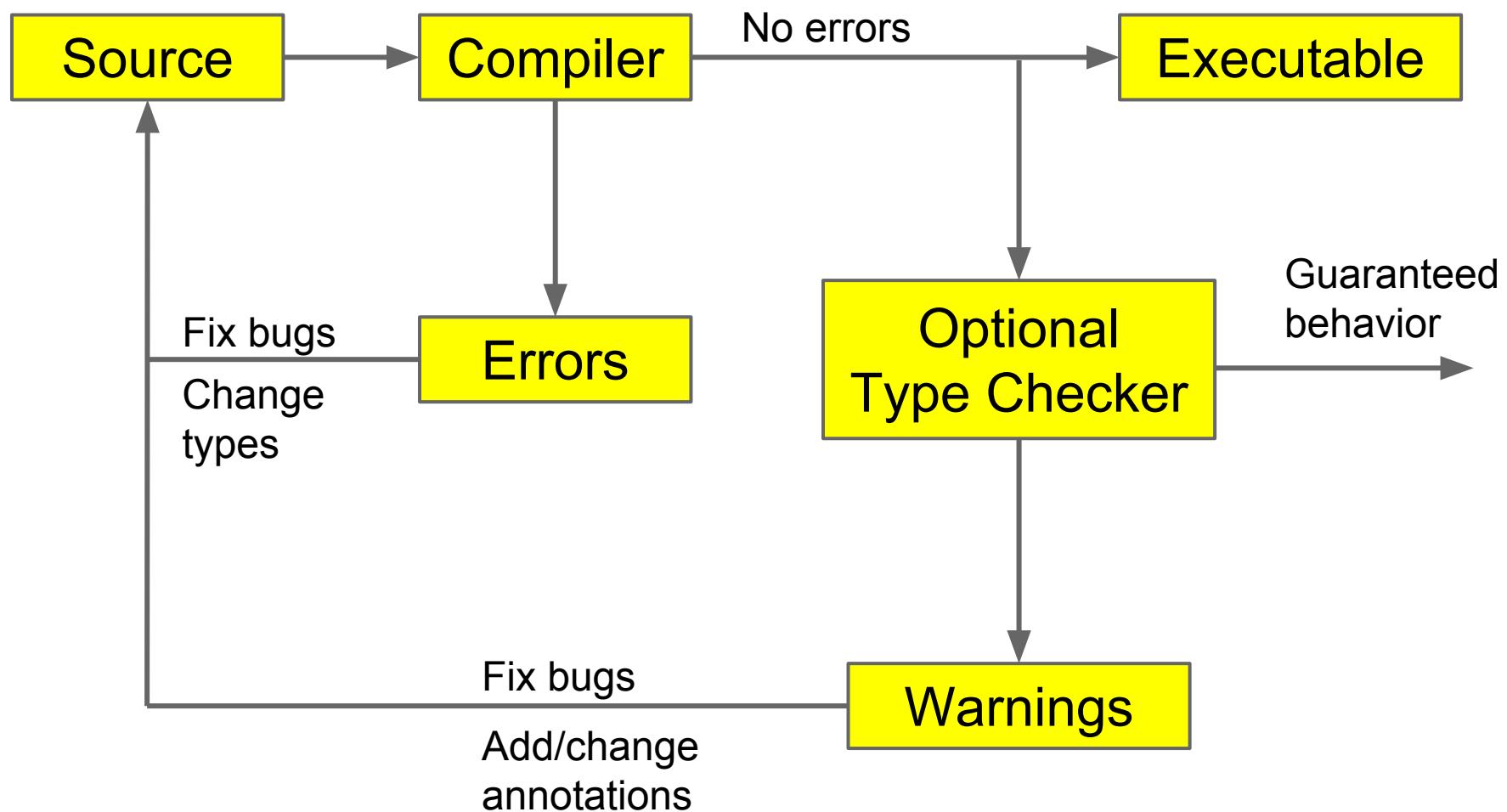
```
javac -processor MyChecker ...
```

Eclipse plug-in, Ant and Maven integration

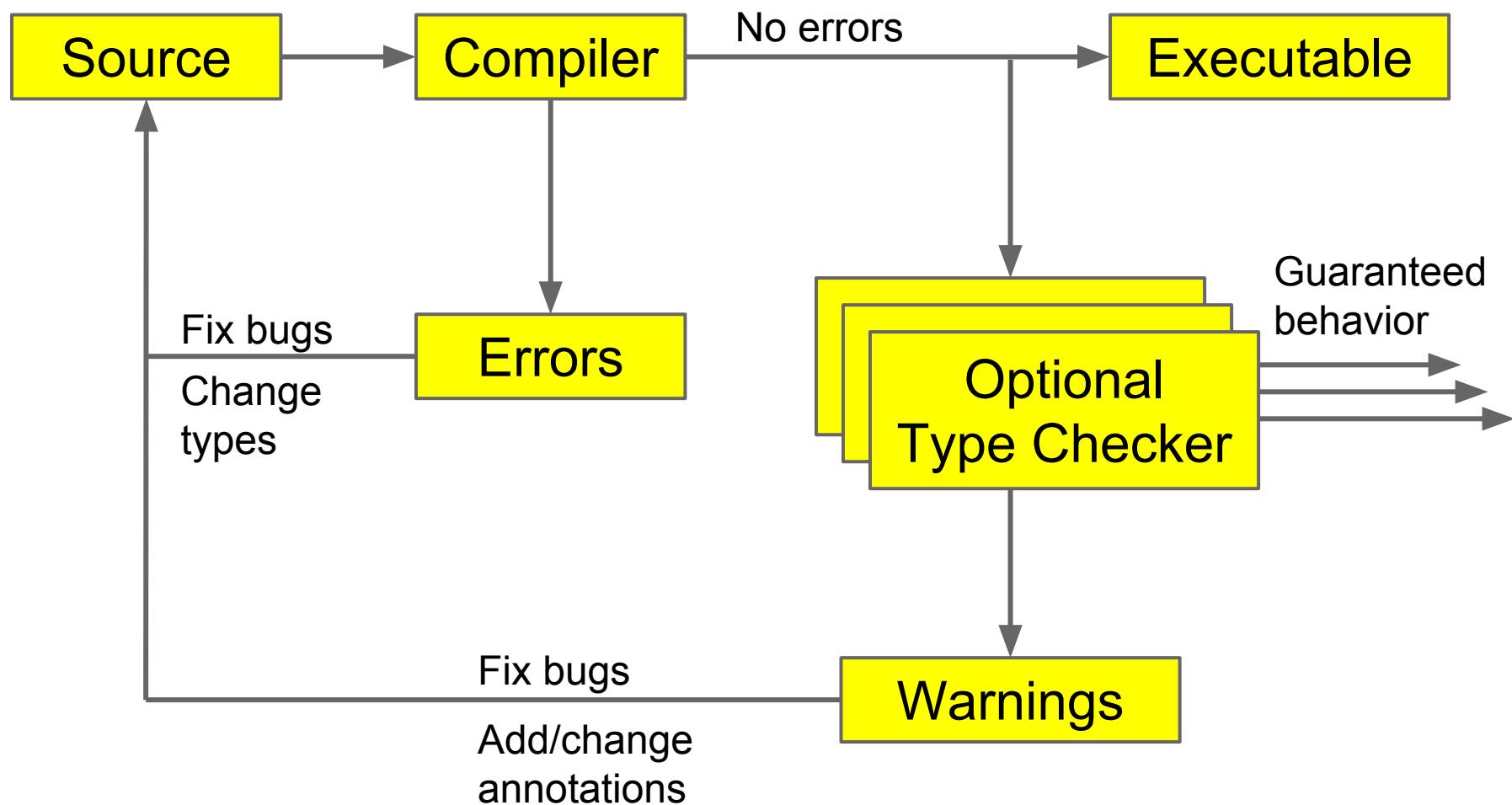
Type Checking



Optional Type Checking



Optional Type Checking



What bugs can you detect & prevent?

The property you care about:

Null dereferences

Mutation and side-effects

Concurrency: locking

Security: encryption,
tainting

Aliasing

Equality tests

Strings: localization,

Regular expression syntax

Signature format

Enumerations

Typestate (e.g., open/closed files)

The annotation you write:

@NotNull

@Immutable

@GuardedBy

@Encrypted

@Untainted

@Linear

@Interned

@Localized

@Regex

@FullyQualified

@Fenum

@State

Users can write their own checkers!

Checker Framework experience

Type checkers reveal important latent defects

Ran on >3 million LOC of real-world open-source code as of early 2011

Found hundreds of user-visible failures

Annotation overhead is low

Mean 2.6 annotations per kLOC

20 annotations per kLOC for Nullness Checker

[Dietl et al. ICSE'11]

Formalizations

$P \in \text{Program}$	$::=$	$\overline{\text{Class}}, \text{ClassId}, \text{Expr}$	$rT \in rType$
$\text{Cls} \in \text{Class}$	$::=$	$\text{class ClassId} < \overline{\text{TVarId}}$ $\text{extends ClassId} < \overline{sType}$ $\{ \overline{\text{FieldId}} \ sType; \overline{\text{Met}}$	$Fs \in \text{Fields}$ $\iota \in \text{OwnerI}$ $r\Gamma \in rEnv$
$sT \in sType$	$::=$	$sNType \mid \overline{\text{TVarId}}$	
$sN \in sNType$	$::=$	$\text{OM ClassId} < \overline{sType}$	
$u \in \text{OM}$	$::=$		$h, r\Gamma, e_0 \rightsquigarrow h_0, \iota_0$
$mt \in \text{Meth}$	$::=$		$\iota_0 \neq \text{null}_a$
	$\text{MethSig} ::=$		$h_0, r\Gamma, e_2 \rightsquigarrow h_2, \iota$
$w \in \text{Purity}$	$::=$		$h' = h_2[\iota_0.f := \iota]$
$e \in \text{Expr}$	$::=$	OS-Upd	$h, r\Gamma, e_0.f = e_2 \rightsquigarrow h'$,
			$\text{Expr.MethId} < \overline{sType}(\text{Expr}) \mid$ $\text{new } sType \mid (sType) \text{ Expr}$
$s\Gamma \in sEnv$	$::=$	$\overline{\text{TVarId}} \ sNType; \overline{\text{ParId}} \ sType$	
			$\frac{\Gamma \vdash e_0 : N_0 \quad N_0 = _- \text{GT-} -}{\Gamma \vdash e_0.f : N_0 \triangleright fType(C_0, f)}$
$h \vdash r\Gamma : s\Gamma$			
$h \vdash \iota_1 : dyn(sN, h, \iota_1)$			
$h \vdash \iota_2 : dyn(sT, \iota_1, h(\iota_1) \downarrow_1)$			
$sN = u_N \ C_N < \rangle$			
$u_N = \text{this}_u \Rightarrow r\Gamma(\text{this})$			
$free(sT) \subseteq \text{dom}(C_N)$			
			$\left. \begin{array}{l} \Rightarrow h \vdash \iota_2 : dyn(sN \triangleright sT, h, \iota_2) \\ rT = \iota' _ \langle \rangle \quad \iota \vdash rT \ r \langle : \iota' \ C \langle \overline{rT} \rangle \end{array} \right\} \text{DYN} \quad \text{dom}(C) = \bar{X}$
			$dyn(sT, \iota, rT, (\overline{X'} \ \overline{rT'}; _)) = sT[\iota' / \text{this}]$

- = $\text{Addr} \rightarrow \text{Obj}$
- = Set of Addresses $\cup \{\text{null}_a\}$
- = $\text{rType}, \text{Fields}$
- = OwnerAddr ClassId< $\overline{\text{rType}}$ >
- = FieldId \rightarrow Addr
- = $\text{Addr} \cup \{\text{any}_a\}$
- = $\overline{\text{TVarId rType}}; \text{ParId Addr}$

$$\text{OS-Read} \frac{\begin{array}{c} h, {}^r\Gamma, e_0 \rightsquigarrow h', \iota_0 \\ \iota_0 \neq \text{null}_a \\ \iota = h'(\iota_0) \downarrow_2 (f) \end{array}}{h, {}^r\Gamma, e_0.f \rightsquigarrow h', \iota}$$

$$\frac{\Gamma \vdash e_0 : N_0 \quad N_0 = u_0 \ C_0 \leftarrow T_1 \quad \Gamma \vdash e_2 : N_0 \triangleright T_1 \quad u_0 \neq \text{any}}{\Gamma \vdash e_2 : N_0 \triangleright T_1}$$

$$\text{GT-Read} \frac{\Gamma \vdash e_0 : N_0 \quad N_0 = _}{\Gamma \vdash e_0.f : N_0 \triangleright fType(c_0, f)} \text{GT-Upd} \frac{u_0 \neq \text{any}}{\Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1} \frac{rp(u_0, T_1)}{}$$

$$h \vdash {}^r\Gamma : {}^s\Gamma \quad \text{G-T-Read} \quad \Gamma \vdash e_0.f : N_0 \triangleright fType(C_0, f)$$

$\mathbf{h} \vdash \iota_1 : dyn(\mathbf{sN}, \mathbf{h}, \mathbf{^1}_\perp)$

$$h \vdash \iota_2 : dyn(\text{``}T, \iota_1, h(\iota_1) \downarrow_1)$$

$$^sN = u_N \ C_N < >$$

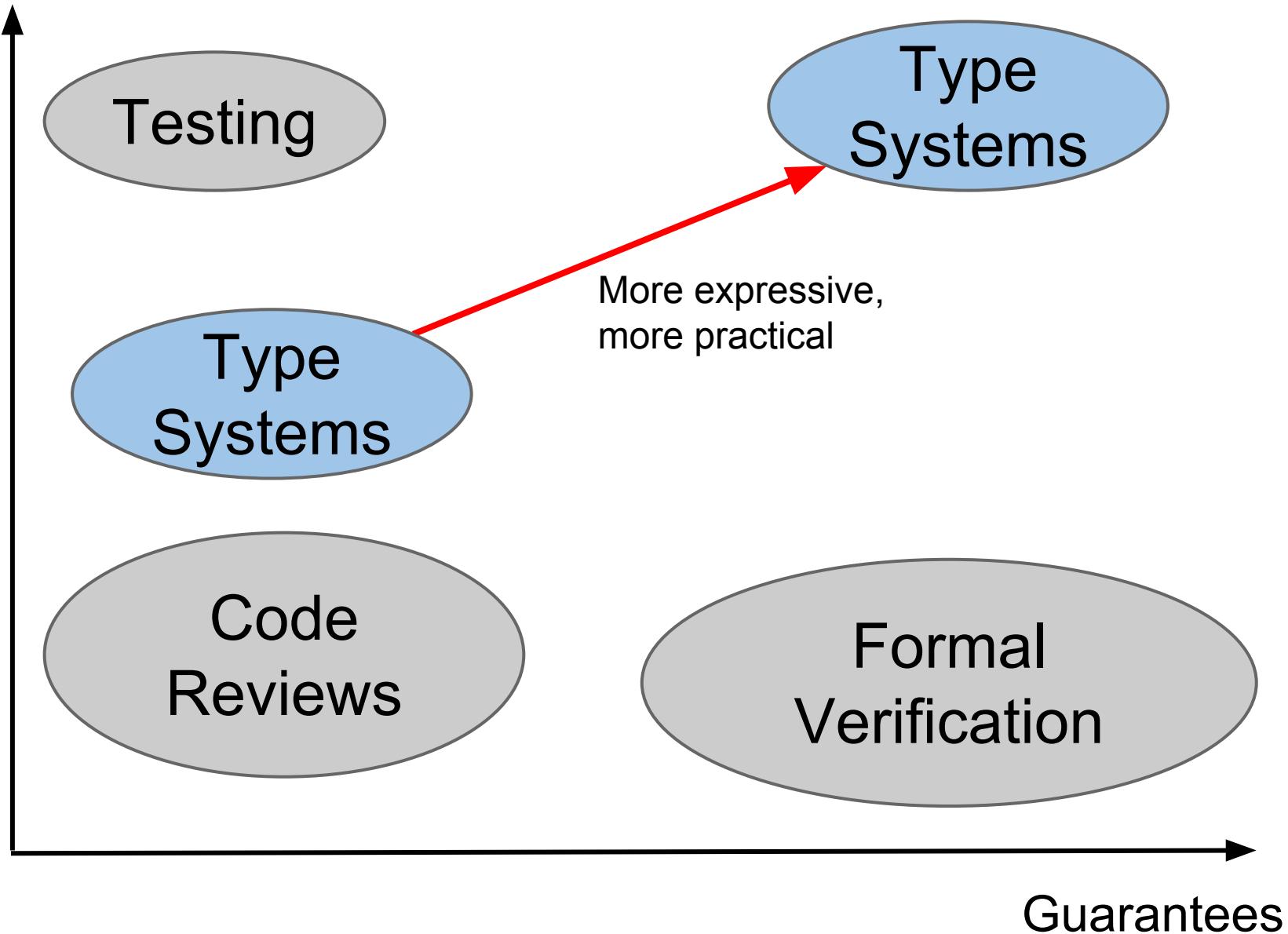
$u_N = \text{this}_u \Rightarrow {}^r\Gamma(\text{this})$

$$free(\mathbf{s}T) \subseteq dom(C_N)$$

$$\left\{ \begin{array}{l} \implies h \vdash \iota_2 : dyn({}^s N \triangleright {}^s T, h, {}^r \Gamma) \\ {}^r T = \iota' _ < _ > \quad \iota \vdash {}^r T \ {}^r < : \iota' C < \overline{{}^r T} > \quad \iota \vdash {}^r T \ {}^r < : \iota' C < \overline{{}^r T_a} > \Rightarrow \iota \vdash \overline{{}^r T} \ {}^r < : \overline{{}^r T}_a \\ \text{dom}(C) = \overline{X} \qquad \qquad \qquad \text{free}({}^s T) \subseteq \overline{X} \circ \overline{X}' \end{array} \right.$$

Practicality

Allow reliable and secure
programming in practice



Outline

- Pluggable type-checking
- Architecture overview
 - Java 8 syntax for Type Annotations
 - Dataflow Framework
 - Checker Framework
- Our experience
 - Case Studies
 - SPARTA: Android Security
 - VeriGames: Crowd-sourced Verification Games
- Writing your own type system

Project overview

Java 8 type annotation feature

jsr308-langtools extensions

annotation-tools

checker-framework

javacutils

dataflow

stubparser

framework

checker: 18 default type systems

checker-framework-inference

Project overview

Java 8 type annotation feature
jsr308-langtools extensions

annotation-tools

checker-framework

javacutils

dataflow

stubparser

framework

checker: 18 default type systems

checker-framework-inference

Java 8 extends annotation syntax

Annotations on all occurrences of types:

```
@Untainted String query;  
List<@NotNull String> strings;  
myGraph = (@Immutable Graph) tmp;  
class UnmodifiableList<T>  
    implements @ReadOnly List<T> {}
```

Stored in classfile

Handled by javac, javap, javadoc, ...

Explicit method receivers

```
class MyClass {  
    int foo(@TParam String p) {...}  
    int foo(@TRecv MyClass this,  
            @TParam String p) {...}
```

No impact on method binding and overloading

Constructor return & receiver types

Every constructor has a return type

```
class MyClass {  
    @TReturn MyClass(@TParam String p) {...}
```

Inner class constructors also have a receiver

```
class Outer {  
    class Inner {  
        @TReturn Inner(@TRcv Outer Outer.this,  
                        @TParam String p) {...}}
```

Array annotations

A read-only array of non-empty arrays of English strings:

```
@English String @ReadOnly [] @NonEmpty [] a;
```

CF has invariant arrays:

`@NotNull String[]` is unrelated to
`@Nullable String[]`

Compare to Java's unsound covariant arrays:

`String[]` is a subtype of `Object[]`

CF: Java 6 & 7 Compatibility

Annotations in comments

```
List</*@NotNull*/ String> strings;
```

Voodoo comments for arbitrary source code

```
/*>>> import myquals.TRecv; */
```

```
...
```

```
int foo(/*>>> @TRecv MyClass this,*/
        @TParam String p) {...}
```

CF: Annotating external libraries stub files and annotated JDK

Allows annotating external libraries

- As separate text file (stub file)
`checker-framework/checker/src/.../jdk.astub`
- Within its .jar file (from annotated partial source code)
`checker-framework/checker/jdk/...`

Dataflow Framework

Initially for flow-sensitive type refinement
Now a project of its own right

1. Translate AST to CFG
Standard multipass visitor over AST
2. Perform dataflow analysis over CFG with user-provided
 - a. Abstract value What are we tracking?
 - b. Transfer functions What do operations do?
 - c. Store What are intermediate results?
3. Allow queries about result

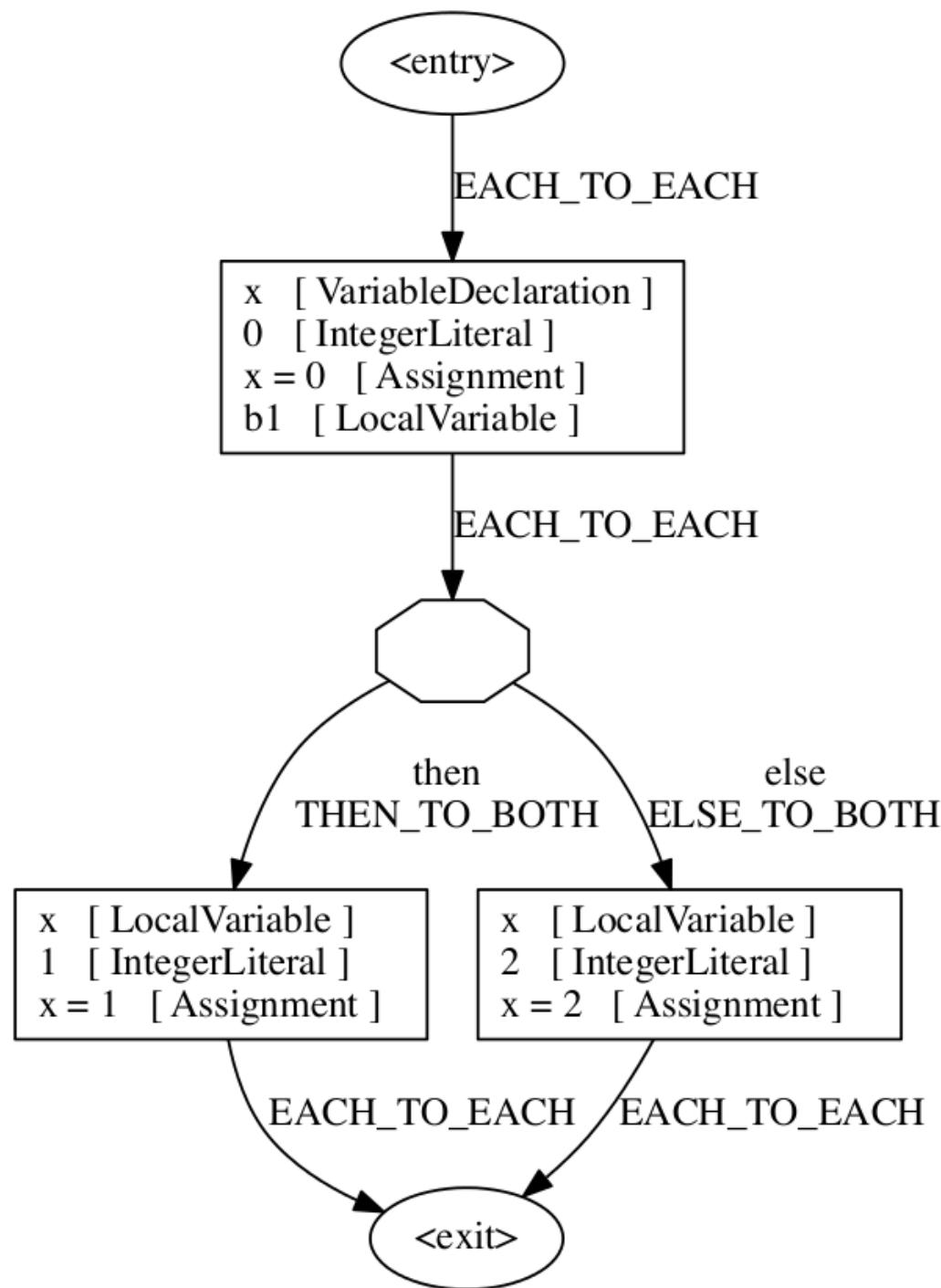
Control Flow Graph

CFG is a graph of basic blocks

- Conditional basic blocks to model conditional control flow
- Exceptional edges

Use type Node for all Java operations and expressions, e.g.,

- StringLiteralNode, FieldAccessNode, etc.
- Make up the content of basic blocks



Properties of the CFG

- Explicit representation of implicit Java constructs
 - Unboxing, implicit type conversions, etc.
 - Analyses do not need to worry about these things
 - All control flow explicitly modeled (e.g. exceptions on field access)
- High-level constructs
 - Close to source language
- Different from other approaches
 - Not three-address-form
 - Analysis is not performed over the AST

Checker Framework - “Framework”

- Full type systems: inheritance, overriding, ...
- Generics (type polymorphism)
 - Also qualifier polymorphism
- Flow-sensitive type qualifier inference
 - Infers types for local variables
 - reusable component
- Qualifier defaults
- Pre-/Post-conditions
- Warning suppression
- Testing infrastructure

Outline

- Pluggable type-checking
- Architecture overview
 - Java 8 syntax for Type Annotations
 - Dataflow Framework
 - Checker Framework
- Our experience
 - Case Studies
 - SPARTA: Android Security
 - VeriGames: Crowd-sourced Verification Games
- Writing your own type system

Evaluations

- Checkers reveal important latent bugs
 - Ran on >3 million LOC of real-world code
 - Found 40 user-visible bugs, hundreds of mistakes
- Annotation overhead is low
 - Mean 2.6 annotations per kLOC
- Learning their usage is easy
 - Used successfully by first-year CS majors
- Building checkers is easy
 - New users developed 3 new realistic checkers

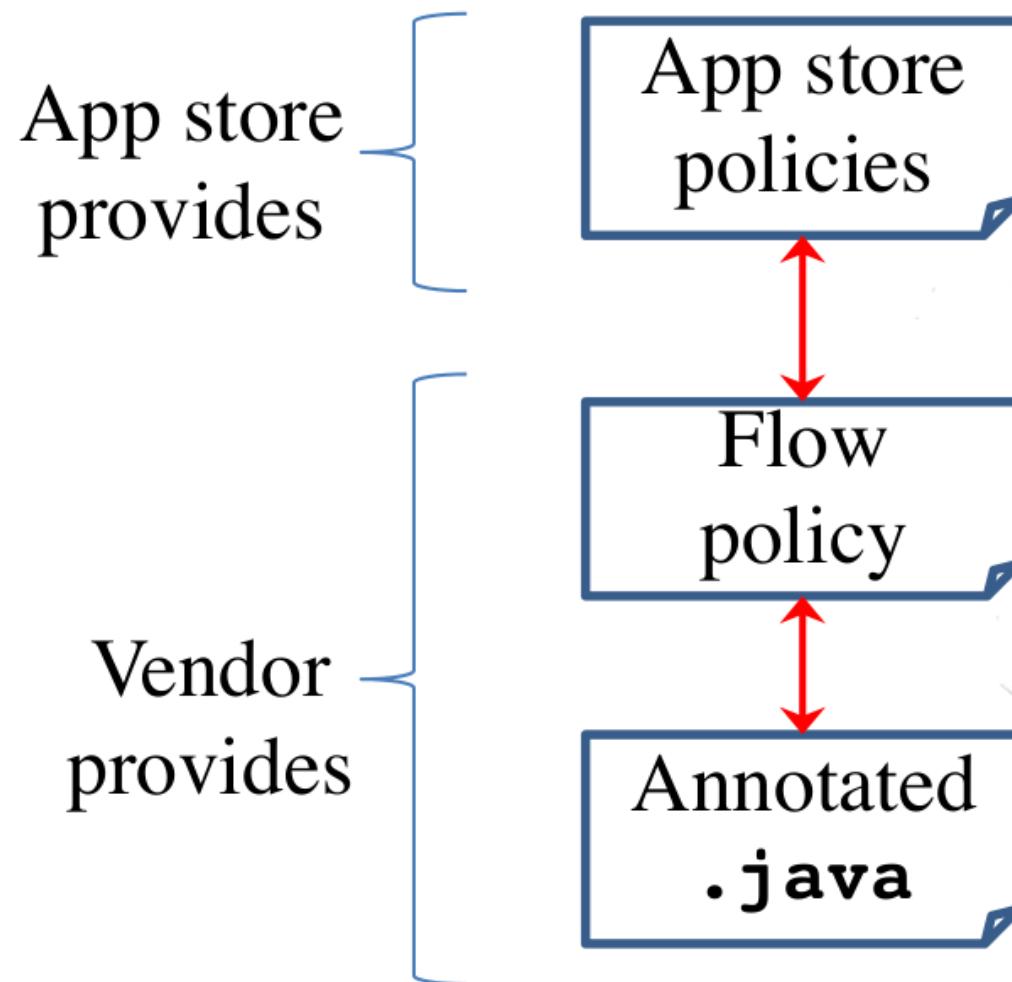
SPARTA: Static Program Analysis for Reliable Trusted Apps

Security type system for Android apps
Guarantees no leakage of private information

Part of DARPA's
Automated Program Analysis
for Cybersecurity (APAC)
program



Collaborative Verification



SPARTA: Static Program Analysis for Reliable Trusted Apps

- Software vendor and the app store auditor collaborate on verification task
- Flow-sensitive, context-sensitive information-flow type system
- Red Team provided 72 apps (576 kLOC)
- Detected 96% of information flow malware and 82% of all malware

See "Collaborative Verification of Information Flow for a High-Assurance App Store" paper at CCS'14

Crowd-sourced verification

Make software verification easy and fun

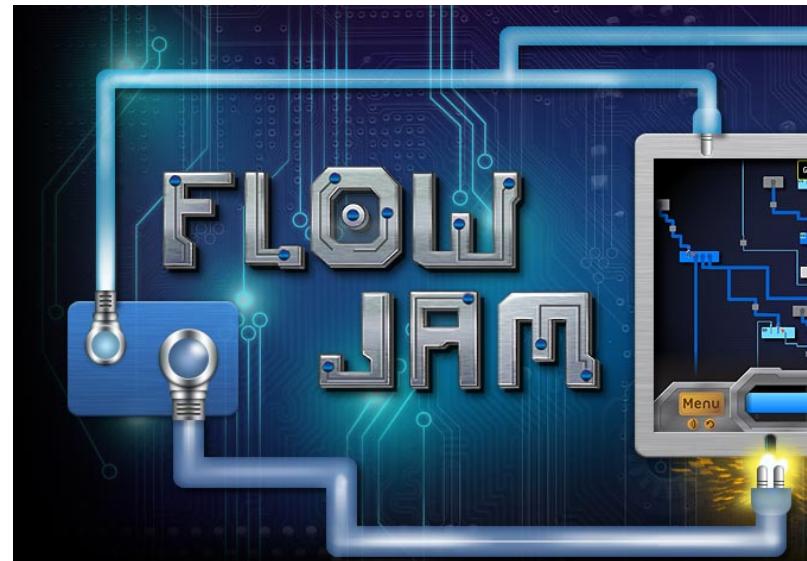
Make the game accessible to everyone

Harness the power of the crowd

Goal: Verify software while waiting

<http://verigames.com/>

FTfJP'12 paper



Building checkers is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {...}  
@Encrypted String msg1 = ...;  
send(msg1);    // OK  
String msg2 = ....;  
send(msg2);    // Warning!
```

Building checkers is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {...}  
@Encrypted String msg1 = ...;  
send(msg1);    // OK  
String msg2 = ....;  
send(msg2);    // Warning!
```

The complete checker:

```
@Target(ElementType.TYPE_USE)  
@SubtypeOf(Unqualified.class)  
public @interface Encrypted {}
```

Testing Infrastructure

jtreg-based testing as in OpenJDK

Light-weight tests with in-line expected errors:

```
String s = "%+s%";  
//::: error: (format.string.invalid)  
f.format(s, "illegal");
```

Defining a type system

1. Qualifier hierarchy
 - defines subtyping
2. Type introduction rules
 - types for expressions
3. Type rules
 - checker-specific errors
4. Flow-refinement
 - checker-specific flow properties

Conclusions

Java 8 syntax for type annotations

Dataflow Framework

Checker Framework for creating type checkers

- Featureful, effective, easy to use, scalable

Prevent bugs at compile time

Create custom type-checkers

Learn more, or download at:

<http://CheckerFramework.org/>