The Unsafety in Java Regression Test Selection and its Occurrence in the Wild

Lukas Döllerer

Seminar: Software Quality

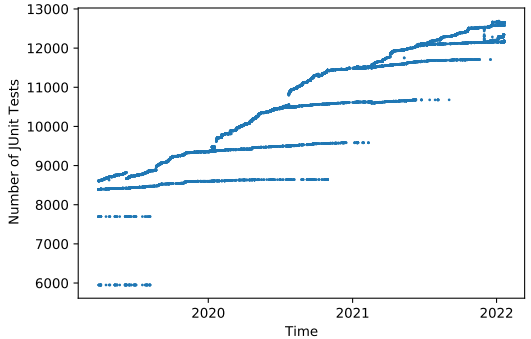
Informatics 4 - Software and Systems Engineering

June 12, 2022

The Unsafety in Java Regression Test Selection and its Occurrence in the Wild

What is the problem?

Over time, projects accumulate a lot of regression tests.



Too many regression tests.

→ *RetestAll* not feasible

→ ⚠ Bugs / regression may be introduced.

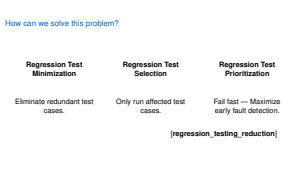
Figure: Spring Boot JUnit tests in the “master” branch.

- Approx. number of JUnit Tests in the Spring Boot Repository.
- From < 9000 in Beginning 2019 to approx. 13000 now.
- We are assuming that these are all regression tests
- Test suits tend to get too big.
- executing all tests (called retest all) requires too many computing resources.
- leads to a higher probability of occuring Regression faults

How can we solve this problem?

Regression Test Minimization	**Regression Test Selection**	**Regression Test Prioritization**
Eliminate redundant test cases.	Only run affected test cases.	Fail fast — Maximize early fault detection.

[**regression_testing_reduction**]



- Yoo and Harman identified 3 ways to tackle the problem of growing regression test suites.
- Minimization -> Eliminate redundancy
- Selection -> Only execute affected tests
- Prioritization -> Intelligently reorder tests
-
- For today's presentation, I'm going to focus on **Regression Test Selection**.

How can we solve this problem?

Regression Test Minimization	**Regression Test Selection**	Regression Test Prioritization
Eliminate redundant test cases.	Only run affected test cases.	Fail fast — Maximize early fault detection.

[regression_testing_reduction]

- Yoo and Harman identified 3 ways to tackle the problem of growing regression test suites.
- Minimization -> Eliminate redundancy
- Selection -> Only execute affected tests
- Prioritization -> Intelligently reorder tests
-
- For today's presentation, I'm going to focus on **Regression Test Selection**.

Regression Test Selection (RTS)

1. **Dependency collection:** Which parts of the source code affect the test's result?
2. **Identification of affected tests:** Were any of these parts changed?

[**regression_testing_reduction**]

Regression Test Selection (RTS)

1. **Dependency collection:** Which parts of the source code affect the test's result?
2. **Identification of affected tests:** Were any of these parts changed?

[**regression_testing_reduction**]

- Regression test selection is a 2-step process
- 1. Dependency Collection -> Associate tests with their tested source code. (this could be done asynchronously)
- 2. Identification of affected tests -> Use set of source code changes to determine affected tests.

S:4:00

Safe Regression Test Selection

“[...] they select every test from the original test suite that can expose faults in the modified program.” [**safety_paper**].

Additionally, we require that **safe** RTS does not otherwise interfere with testing.

Since 1997, a lot of RTS tools claimed to be **safe**. However, this *safety* is often only semi-formally proven for code-only changes.

Safe Regression Test Selection

“[...] they select every test from the original test suite that can expose faults in the modified program.” [**safety_paper**].

Additionally, we require that **safe** RTS does not otherwise interfere with testing.

Since 1997, a lot of RTS tools claimed to be **safe**. However, this *safety* is often only semi-formally proven for code-only changes.

- In 1997, two researchers defined **safe** RTS as the ability of your tool to ...
- We add to this definition that the rts tools must not otherwise interfere with testing.
- Over the last 20 years, many supposedly safe rts tools were published.
- Most of them do not supply perfect formal proofs of their safety. And they often only proof safety for code changes.
- But is this a problem?

RTS Safety — Not so easy?

Zhu and others presented RTSCHECK, a tool for testing RTS-Tools, in 2019.

They found **9 sources of unsafety** in 3 RTS-Tools that were expected to act safe in most conditions.

[**unsafety_eval**]

RTS Safety — Not so easy?
Zhu and others presented RTSCHECK, a tool for testing RTS-Tools, in 2019.
They found **9 sources of unsafety** in 3 RTS-Tools that were expected to act safe in most conditions.
[unsafety_eval]

- In 2019, a research group published a tool called RTScheck for testing RTS-tools.
- First project to find empirical evidance for unsafety in supposedly safe rts tools.
- They found multiple sources of unsafety in popular RTS-tools.
- With my work, I want to extend this catalog of sources of unsafety.

S:4:00

RTS Safety — Not so easy?

Zhu and others presented RTSCHECK, a tool for testing RTS-Tools, in 2019.

They found **9 sources of unsafety** in 3 RTS-Tools that were expected to act safe in most conditions.

Supposedly **safe** RTS-Tools act unsafe in certain conditions.

[**unsafety_eval**]

RTS Safety — Not so easy?
Zhu and others presented RTSCHECK, a tool for testing RTS-Tools, in 2019.
They found **9 sources of unsafety** in 3 RTS-Tools that were expected to act safe in most conditions.
Supposedly **safe** RTS-Tools act unsafe in certain conditions.
[**unsafety_eval**]

- In 2019, a research group published a tool called RTScheck for testing RTS-tools.
- First project to find empirical evidance for unsafety in supposedly safe rts tools.
- They found multiple sources of unsafety in popular RTS-tools.
- With my work, I want to extend this catalog of sources of unsafety.

S:4:00

Research Questions & Study Objects

Research Questions & Study Objects

Study Objects

Name	*Ekstazi*	*GIBstazi*	*OpenClover*	*STARTS*	*HyRTS*
Venue	ISSTA	ISSRE	—	ASE	ICSE
Dependency Collection Technique	Dynamic Runtime Instrumentation	Dynamic with Static Analysis	Static Instrumentation	Static Analysis	Dynamic Runtime Instrumentation

Table: Study Objects

Study Objects

Name	*Ekstazi*	*GIBstazi*	*OpenClover*	*STARTS*	*HyRTS*
Venue	ISSTA	ISSRE	—	ASE	ICSE
Dependency Collection Technique	Dynamic Runtime Instrumentation	Dynamic with Static Analysis	Static Instrumentation	Static Analysis	Dynamic Runtime Instrumentation

Table: Study Objects

- My study objects: A mix of common Java RTS-tools
- **Ekstazi**: Most commonly seen tool. -> International Symposium on Software Testing and Analysis
- **GIBstazi**: Wrapper around Ekstazi. -> International Symposium on Software Reliability Engineering
- **OpenClover**: Previously developed by Atlassian.
- **STARTS**: The only purely static tool. -> International Conference on Automated Software Engineering
- **HyRTS**: A hybrid RTS-tool. -> International Conference on Software Engineering

S:6:00

Research Questions

RQ1 What sources of unsafety exist for the examined RTS-tools?

RQ2 What are the differences in safety of the examined tools, in the context of the previously identified sources of unsafety?

RQ3 How can potential for unsafety be automatically identified in code changes without dynamic program analysis?

RQ4 In which quantities do the identified sources of unsafety occur in real world software projects?

Research Questions

RQ1 What sources of unsafety exist for the examined RTS-tools?
RQ2 What are the differences in safety of the examined tools, in the context of the previously identified sources of unsafety?
RQ3 How can potential for unsafety be automatically identified in code changes without dynamic program analysis?
RQ4 In which quantities do the identified sources of unsafety occur in real world software projects?

- Part 1: Sources of unsafety in the laboratory.

RQ1 What sources of unsafety exist for the examined RTS-tools?

RQ2 What are the differences in safety of the examined tools, in the context of the previously identified sources of unsafety?

-

- Part 2: Sources of unsafety in the wild -> look at actual software projects.

RQ3 How can potential for unsafety be automatically identified in code changes without dynamic program analysis?

RQ4 In which quantities do the identified sources of unsafety occur in real world software projects?

S:6:00

Evaluation of Sources of Unsafety

RQ1 & RQ2

S:10:00

Color Code

Conflicted with the proposed source of unsafety, the RTS-tool. . .

✓	. . . acts **safe**.
×	. . . acts **unsafe**.
⚠	. . . was unable to execute the tests.

Example:

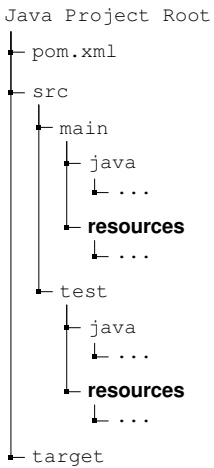
Ekstazi	✓
GIBstazi	✓
OpenClover	×
STARTS	×
HyRTS	⚠

Color Code
Conflicted with the proposed source of unsafety, the RTS-tool. . .
✓ . . . acts safe.
× . . . acts unsafe.
⚠ . . . was unable to execute the tests.
Example:
Ekstazi ✓
GIBstazi ✓
OpenClover ×
STARTS ×
HyRTS ⚠

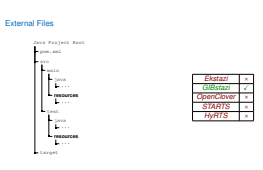
- **Important**: To find sources of unsafety, I looked at existing research, skimmed through the tool's source codes and thought about complications with the tool's dependency collection.
- In the following slides, you will only see the sources of unsafety that actually make one of the examined tools act unsafe.
-
- Color-Code for the following slides. -> Table on the right side of the screen. -> Table shows response of the examined tools.
- General notice: In order to save time, I'm not going to discuss the information contained in these tables in detail.

S:10:00

External Files



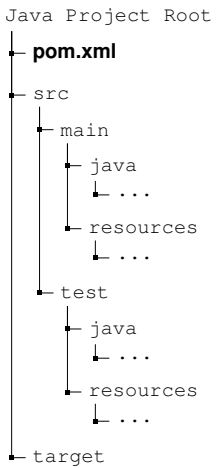
Ekstazi	×
GIBstazi	✓
OpenClover	×
STARTS	×
HyRTS	×



- File resources explicitely loaded via program code.
- Common location for external files in maven directory structure is the resources folder.
- One big difference to the next source of unsafety (configuration files)

S:10:00

Configuration Files




```
<version>2.11.4</version>
```

⇓

```
<version>2.12.0</version>
```

Ekstazi	×
GIBstazi	✓
OpenClover	×
STARTS	✓
HyRTS	×

- Implicitely loaded
- Change behavior of libraries and external tools
- Here example: changing a library version
- “As we can see from the response of the tools, this is actually a different source of unsafety than external files.”

S:10:00

Reflections

```
Class.forName("ModifiedClass");
```

Ekstazi	✓
GIBstazi	✓
OpenClover	✓
STARTS	×
HyRTS	×

Reflections
Class.forName("ModifiedClass");
Ekstazi ✓
GIBstazi ✓
OpenClover ✓
STARTS ×
HyRTS ×

- A well known problem for static tools: reflections
- “We focused on the `Class.forName(...)` method, because it allows the programmer to access features of classes that are not explicitly imported.”
- Class.forname method source for most reflection problems.

S:10:00

Runtime Instrumentation

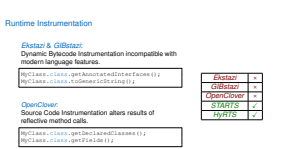
Ekstazi & *GIBstazi*:
Dynamic Bytecode Instrumentation incompatible with modern language features.

```
MyClass.class.getAnnotatedInterfaces();
MyClass.class.toGenericString();
```

OpenClover:
Source Code Instrumentation alters results of reflective method calls.

```
MyClass.class.getDeclaredClasses();
MyClass.class.getFields();
```

Ekstazi	×
GIBstazi	×
OpenClover	×
STARTS	✓
HyRTS	✓



- Special source of unsafety:
- A rather common problem - Runtime instrumentation that changes the code's behavior
- Invasive or buggy runtime instrumentation can cause unsafety.
- Ekstazi's and GIBstazi's (Dynamic Bytecode Instrumentation) can't execute all reflective method calls, which is causing runtime errors
- OpenClover's source code instrumentation adds fields and classes and therefore secretly changes method call results.

S:10:00

Dependency Injection (1/3)

Dependency Source Code Changes - Spring Framework

```
@Bean
public BeanInterface beanName() {
   // change this implementation
   return new Implementation();
}
```

Ekstazi	✓
GIBstazi	⚠
OpenClover	⚠
STARTS	×
HyRTS	✓

Dependency Injection (1/3)
Dependency Source Code Changes - Spring Framework

- Probably one of the most underestimated sources of unsafety.
- Once dependency injection is applied, the examined tools start to act unsafely
- Focused research on Spring and Guice frameworks as they are the most commonly used libraries.
-
- For starters, this is a straight forward source of unsafety:
- Declare an injected dependency, create a matching implementation and change this implementation. (here with Spring)
- -> next slide for Guice

S:10:00

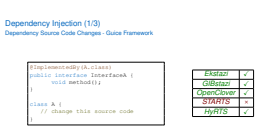
Dependency Injection (1/3)

Dependency Source Code Changes - Guice Framework

```
@ImplementedBy(A.class)
public interface InterfaceA {
      void method();
}

class A {
   // change this source code
}
```

Ekstazi	✓
GIBstazi	✓
OpenClover	✓
STARTS	×
HyRTS	✓



- Change the implementation of an injected dependency with Guice.

S:10:00

Dependency Injection (2/3)

Collection Injection - Spring Framework

```
@Autowired
Collection<BeanInterface> injected;
```

⇒ Multiple Beans packed into a collection.

Ekstazi	×
GIBstazi	⚠
OpenClover	⚠
STARTS	×
HyRTS	×

- Big problem with the Spring framework: Automatic classpath scanning (we'll come back to that in a moment)
- Define multiple implementations for the same dependency (Bean) -> Spring *implictly* packs all into a collection
- Adding new implementations is undetected

S:10:00

Dependency Injection (2/3)

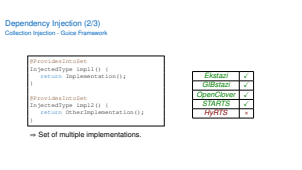
Collection Injection - Guice Framework

```
@ProvidesIntoSet
InjectedType impl1() {
   return Implementation();
}

@ProvidesIntoSet
InjectedType impl2() {
   return OtherImplementation();
}
```

⇒ Set of multiple implementations.

Ekstazi	✓
GIBstazi	✓
OpenClover	✓
STARTS	✓
HyRTS	×



- Collection injection is also available for the Guice framework
- way more *explicit*
- Most tools are quite good at detecting Guice collection injection.

S:10:00

Dependency Injection (3/3)

Classpath Scanning - Spring Framework only

```
@SpringBootApplication
public class MainSpringAppClass { }

@Configuration
@ComponentScan("example.beanConfig")
public class MainConfiguration { }
```

Ekstazi	×
GIBstazi	× / ⚠
OpenClover	× / ⚠
STARTS	×
HyRTS	×

Dependency Injection (3/3)
Classpath Scanning - Spring Framework only

Ekstazi ×
GIBstazi × / ⚠
OpenClover × / ⚠
STARTS ×
HyRTS ×

- Classpath scanning - the big source of unsafety of the spring framework
- The code only uses annotations and doesn't explicitely call an injector or configuration class.
- Spring handles dependency management in the background
- -> Dependencies are unclear, changes may not be detected

S:10:00

Occurence of Unsafety in the Wild

RQ3 & RQ4

S:13:00

Occurence in the Wild

But, are these scenarios actually relevant?

Do these sources of unsafety occur in real-world software projects?

⇒ Search through the last 100 commits of 100 open source projects.

Occurence in the Wild
But, are these scenarios actually relevant?
Do these sources of unsafety occur in real-world software projects?
⇒ Search through the last 100 commits of 100 open source projects.

- You may ask yourselfs: “Are these purely hypothetical sources of unsafety that occur under laboratory conditions?”
- I askes myself the same question.
- Performed a broad search.

S:13:00

Occurence in the Wild
Study Object Selection

Choose the 100 public projects with the most “stars” from GitHub that. . .

- . . . have the majority of their code written in Java
- . . . use Maven as their build system
- . . . were recently updated (at least once after the 06/01/2020)
- . . . have at least 100 JUnit test cases (approximated)
- . . .

Occurence in the Wild
Study Object Selection
Choose the 100 public projects with the most “stars” from GitHub that. . .
- . . . have the majority of their code written in Java
- . . . use Maven as their build system
- . . . were recently updated (at least once after the 06/01/2020)
- . . . have at least 100 JUnit test cases (approximated)
- . . .

- Filtered the most popular open source repositories from github
- They had to be mostly written in Java, use Maven und were recently updated.

\+ They need to have at least 100 Junit tests

-> we want to make sure that the projects are actual candidates for using an RTS solution in the future

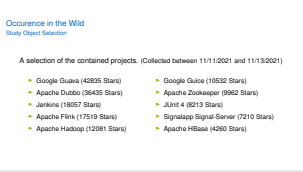
- We sorted all these repositories by stars and picked the top 100 to be our study objects.

S:13:00

Occurence in the Wild
Study Object Selection

A selection of the contained projects. (Collected between 11/11/2021 and 11/13/2021)

- Google Guava (42835 Stars)
- Apache Dubbo (36435 Stars)
- Jenkins (18057 Stars)
- Apache Flink (17519 Stars)
- Apache Hadoop (12081 Stars)
- Google Guice (10532 Stars)
- Apache Zookeeper (9962 Stars)
- JUnit 4 (8213 Stars)
- Signalapp Signal-Server (7210 Stars)
- Apache HBase (4260 Stars)



- Only a quick overview over some of the well known projects in the study object group.

S:13:00

Occurence in the Wild

Commit Scanners

Scan of the last 100 commits of each project. Each source of unsafety was evaluated based on these questions.

External Files	**Dependency Injection**	**Runtime Instrumentation**	**Reflections**
Did the commiter change any external files?	Was any keyword related to dependency injection added / removed?	Does the source code contain any problematic methods?	1. Were reflective accesses introduced? 2. Were reflectively accessed classes changed?

Table: Commit Scanner Objectives

Occurence in the Wild
Commit Scanners
Scan of the last 100 commits of each project. Each source of unsafety was evaluated based on these questions.
External Files | Dependency Injection | Runtime Instrumentation | Reflections
Did the commiter change any external files? | Was any keyword related to dependency injection added / removed? | Does the source code contain any problematic methods? | 1. Were reflective accesses introduced? 2. Were reflectively accessed classes changed?
Table: Commit Scanner Objectives

- To detect said sources of unsafety, I created, what I call, "commit scanners".
-
- External files scanner: detectes commit changes to files in the resources folder (maven, previously shown)
- Dependency Injection scanner: Searches for keywords that are related to spring or guice dependency injection and might introduce unsafety
- Runtime Instrumentation scanner: Detects keywords that cause unsafety with the runtime instrumentation of some of the examined tools.
- Reflections Scanner: Detects occurences of the `Class.forName(...)` method call
- Reflections Scanner: Also searches for changes on classes that are reflectively accessed
- Reflections Scanner: Detected reflections also trigger static initializers -> more yield
- I'm not going to go into technical details of the scanning and jump right to the results.

S:13:00

Commit Scanners — Pseudocode

Main Repository Scanner:

```
for commit in repo.traverse_commits(to=100):
   for scanner in commit_scanners:
         scanner.scan(commit)
```

Commit Scanner Module:

```
def scan(self, commit):
   for change in commit:
      if change_is_source_of_unsafety(change):
         self.unsafety_storage.add(commit)
```


Commit Scanners — Pseudocode
Main Repository Scanner:
for commit in repo.traverse_commits(to=100):
for scanner in commit_scanners:
scanner.scan(commit)
Commit Scanner Module:
def scan(self, commit):
for change in commit:
if change_is_source_of_unsafety(change):
self.unsafety_storage.add(commit)

Results

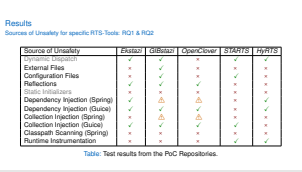
S:17:00

Results

Sources of Unsafety for specific RTS-Tools: RQ1 & RQ2

Source of Unsafety	*Ekstazi*	*GIBstazi*	*OpenClover*	*STARTS*	*HyRTS*
Dynamic Dispatch	✓	✓	×	✓	✓
External Files	×	✓	×	×	×
Configuration Files	×	✓	×	✓	×
Reflections	✓	✓	✓	×	×
Static Initializers	×	×	×	×	×
Dependency Injection (Spring)	✓	⚠	⚠	×	✓
Dependency Injection (Guice)	✓	✓	✓	×	✓
Collection Injection (Spring)	×	⚠	⚠	×	×
Collection Injection (Guice)	✓	✓	✓	✓	×
Classpath Scanning (Spring)	×	×	×	×	×
Runtime Instrumentation	×	×	×	✓	✓

Table: Test results from the PoC Repositories.



- Full evaluation of the discovered sources of unsafety. (same color code as before)
- The gray sources weren't previously discussed
-
- Few things to note:
  - External files are only safe with GIBstazi
  - static initialization, spring dependency injection and classpath scanning is unsafe with all examined tools
- following are the results of the scans of real world software projects.

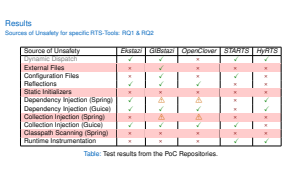
S:17:00

Results

Sources of Unsafety for specific RTS-Tools: RQ1 & RQ2

Source of Unsafety	*Ekstazi*	*GIBstazi*	*OpenClover*	*STARTS*	*HyRTS*
Dynamic Dispatch	✓	✓	×	✓	✓
External Files	×	✓	×	×	×
Configuration Files	×	✓	×	✓	×
Reflections	✓	✓	✓	×	×
Static Initializers	×	×	×	×	×
Dependency Injection (Spring)	✓	⚠	⚠	×	✓
Dependency Injection (Guice)	✓	✓	✓	×	✓
Collection Injection (Spring)	×	⚠	⚠	×	×
Collection Injection (Guice)	✓	✓	✓	✓	×
Classpath Scanning (Spring)	×	×	×	×	×
Runtime Instrumentation	×	×	×	✓	✓

Table: Test results from the PoC Repositories.


Results
Sources of Unsafety for specific RTS-Tools: RQ1 & RQ2

Source of Unsafety	*Ekstazi*	*GIBstazi*	*OpenClover*	*STARTS*	*HyRTS*
Dynamic Dispatch	✓	✓	×	✓	✓
External Files	×	✓	×	×	×
Configuration Files	×	✓	×	✓	×
Reflections	✓	✓	✓	×	×
Static Initializers	×	×	×	×	×
Dependency Injection (Spring)	✓	⚠	⚠	×	✓
Dependency Injection (Guice)	✓	✓	✓	×	✓
Collection Injection (Spring)	×	⚠	⚠	×	×
Collection Injection (Guice)	✓	✓	✓	✓	×
Classpath Scanning (Spring)	×	×	×	×	×
Runtime Instrumentation	×	×	×	✓	✓

Table: Test results from the PoC Repositories.

- Full evaluation of the discovered sources of unsafety. (same color code as before)
- The gray sources weren't previously discussed
-
- Few things to note:
  - External files are only safe with GIBstazi
  - static initialization, spring dependency injection and classpath scanning is unsafe with all examined tools
- following are the results of the scans of real world software projects.

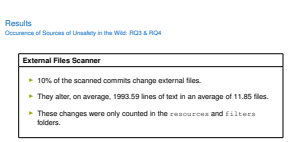
S:17:00

Results

Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4

External Files Scanner

- 10% of the scanned commits change external files.
- They alter, on average, 1993.59 lines of text in an average of 11.85 files.
- These changes were only counted in the `resources` and `filters` folders.



- starting with the external files scanner:
- 989 scanned commits have potential for unsafety
- We detected big changes to a lot of external files, making this an important factor to safety.

S:17:00

Results

Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4

Dependency Injection

- 53% of the projects use the Spring framework, 17% use Guice libraries.
- Detected 484 commits with Spring-related changes
- . . . and 297 commits with Guice-related changes.

Results
Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4
Dependency Injection
- 53% of the projects use the Spring framework, 17% use Guice libraries.
- Detected 484 commits with Spring-related changes
- . . . and 297 commits with Guice-related changes.

- Dependency injection is very important: the majority of the scanned projects uses the spring framework and 17% use guice
- Detected potential for unsafety in 484 commits for spring and 297 commits for guice
- This has to be evaluated even further, but automatically classifying these is hard.

S:17:00

Results

Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4

Runtime Instrumentation

- 63% of the programs are possibly affected by dynamic or source code instrumentation.
- *Ekstazi* cannot execute code from 14 projects.
- *OpenClover* changes the behavior of code in 52 projects.

Results
Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4
Runtime Instrumentation
- 63% of the programs are possibly affected by dynamic or source code instrumentation.
- *Ekstazi* cannot execute code from 14 projects.
- *OpenClover* changes the behavior of code in 52 projects.

- 63% of the projects cannot be tested with either the ekstazi / GIBstazi or OpenClover tools
- Ekstazi prevents execution (because of unsupported method calls) of 14 projects => unexpected runtime errors
- OpenClover changes the behavior of method calls in 52 projects => no obvious errors, just wrong test results
-
- 3 projects are affected by both tools' runtime instrumentation.
- On average 14.80 incompatible files per affected repository.

S:17:00

Results

Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4

Reflections

- 56 projects use the `Class.forName(...)` method.
- A total of 333 occurrences of the `Class.forName(...)` method call.
- 72 changes on reflectively accessed classes were found.
  (⚠ High false negative rate.)

Results
Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4
Reflections
- 56 projects use the Class.forName(...) method.
- A total of 333 occurrences of the Class.forName(...) method call.
- 72 changes on reflectively accessed classes were found.
  (⚠ High false negative rate.)

- More than half of the projects use reflective class access
- 333 references to classes that are not detected by *STARTS* & *HyRTS*
- 72 changes that were (most likely) not detected by *STARTS* & *HyRTS*

S:17:00

Results

Occurence of Sources of Unsafety in the Wild

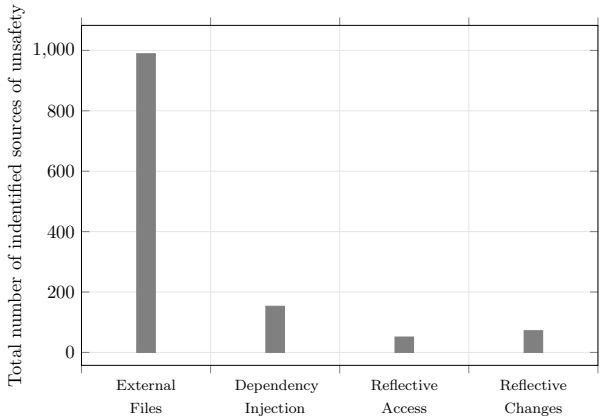
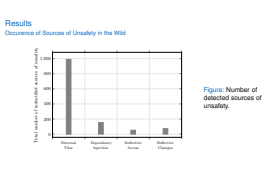


Figure: Number of detected sources of unsafety.



- chart to show the composition of the detected sources of unsafety
- Majority of detected unsafety comes from external files
- Next biggest part is associated with dependency injection

S:17:00

Conclusions

S:20:00

Conclusion
The Alarming State of RTS for Java

- All examined RTS-tools act in an unsafe manner.
- Latest versions of the tools cannot be used as **safe** RTS-tools.
- **Safe** RTS for Java is possible though, but it is not easy.

Conclusion
The Alarming State of RTS for Java
- All examined RTS-tools act in an unsafe manner.
- Latest versions of the tools cannot be used as **safe** RTS-tools.
- **Safe** RTS for Java is possible though, but it is not easy.

- We couldn't identify a suitable tool for safe RTS for Java.
- Although safety is theoretically possible, faulty implementations or methodological limitations can make RTS-tools unsafe.
-

S:20:00

Conclusion
Maintenance of Research Projects

- Tools developed for research purposes are not maintained properly.
- Further research has to focus on improving existing tools.

Conclusion
Maintenance of Research Projects
- Tools developed for research purposes are not maintained properly.
- Further research has to focus on improving existing tools.

- We identified the common pattern that RTS-tools from research papers are not well maintained.
- => Research should focus on improving existing solutions, rather than creating new, unmaintained solutions.

S:20:00

Thank you for your attention

Thank you for your attention

Time for Questions

Appendix

Bibliography I

Backup Slides

Sources of Unsafety

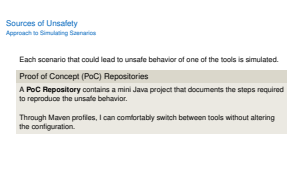
Approach to Simulating Szenarios

Each scenario that could lead to unsafe behavior of one of the tools is simulated.

Proof of Concept (PoC) Repositories

A **PoC Repository** contains a mini Java project that documents the steps required to reproduce the unsafe behavior.

Through Maven profiles, I can comfortably switch between tools without altering the configuration.



- Sources of unsafety are simulated using self-sufficient Java Maven projects.
- They also act as a kind of documentation.
- For the next slides: Really quick overview over some of my identified sources of unsafety.

S:10:00

Dynamic Dispatch

Runtime selection of the most specific method in the inheritance chain.

Ekstazi	✓
GIBstazi	✓
OpenClover	×
STARTS	✓
HyRTS	✓

Dynamic Dispatch
Runtime selection of the most specific method in the inheritance chain.
Ekstazi ✓
GIBstazi ✓
OpenClover ×
STARTS ✓
HyRTS ✓

- Question: "Does the RTS-tool act unsafe in the case of a dynamic dispatch?"
- As we can see, only OpenClover is not able to detect a dynamic dispatch

S:10:00

Static Initializers

```
class A {
   static {
      // Code with side-effects
   }
}

/* [...] */

Class.forName("A");
```

Ekstazi	×
GIBstazi	×
OpenClover	×
STARTS	×
HyRTS	×

Static Initializers
class A { static { // Code with side-effects } }
/* [...] */
Class.forName("A");
Ekstazi ×
GIBstazi ×
OpenClover ×
STARTS ×
HyRTS ×

S:10:00

Runtime Instrumentation

Keywords that cause unsafety.

Ekstazi	*OpenClover*
`getAnnotatedInterfaces()`	`getDeclaredClasses()`
`getAnnotatedSuperclass()`	`getDeclaredFields()`
`toGenericString()`	`getClasses()`
	`getFields()`

Table: Methods whose behavior changes with the corresponding RTS-tools runtime instrumentation. All methods are called on the meta class object of type `Class`.

Runtime Instrumentation
Keywords that cause unsafety.
Ekstazi | OpenClover
getAnnotatedInterfaces() | getDeclaredClasses()
getAnnotatedSuperclass() | getDeclaredFields()
toGenericString() | getClasses()
getFields()
Table: Methods whose behavior changes with the corresponding RTS-tools runtime instrumentation. All methods are called on the meta class object of type Class.

Dependency Injection

Examined Frameworks:

- Spring
- Guice

Scenarios:

- Configuration in External File
- Configuration in Code
  - Dependency Source Code Changes
  - Collection Injection
  - Implicit Configuration
    ⇒ Class Path Scanning (Spring-only)

S:10:00

Dependency Injection
Configuration in External Files

When placing the configuration of dependencies into an external file, the primary cause of unsafety is the tool's inability to detect changes to this external file.

Ekstazi	×
GIBstazi	✓
OpenClover	×
STARTS	×
HyRTS	×

S:10:00

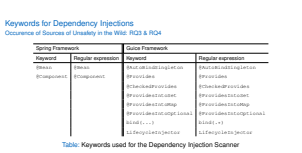
Dependency Injection
Configuration in External Files
When placing the configuration of dependencies into an external file, the primary cause of unsafety is the tool's inability to detect changes to this external file.
Ekstazi ×
GIBstazi ✓
OpenClover ×
STARTS ×
HyRTS ×

Keywords for Dependency Injections

Occurence of Sources of Unsafety in the Wild: RQ3 & RQ4

Spring Framework		Guice Framework	
Keyword	Regular expression	Keyword	Regular expression
`@Bean`	`@Bean`	`@AutoBindSingleton`	`@AutoBindSingleton`
`@Component`	`@Component`	`@Provides`	`@Provides`
		`@CheckedProvides`	`@CheckedProvides`
		`@ProvidesIntoSet`	`@ProvidesIntoSet`
		`@ProvidesIntoMap`	`@ProvidesIntoMap`
		`@ProvidesIntoOptional`	`@ProvidesIntoOptional`
		`bind(...)`	`bind(.*)`
		`LifecycleInjector`	`LifecycleInjector`

Table: Keywords used for the Dependency Injection Scanner



Results

Occurence of Sources of Unsafety in the Wild

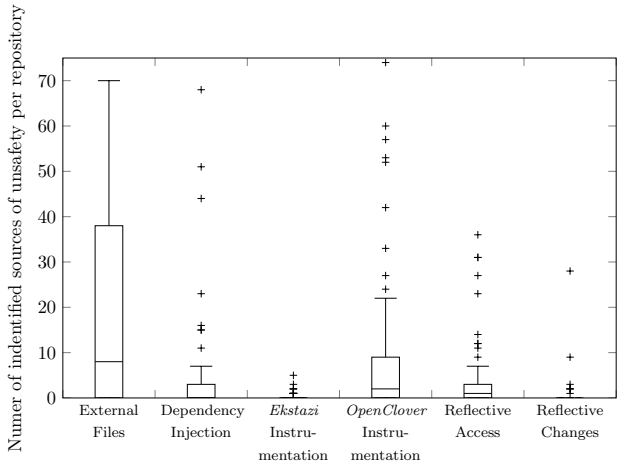


Figure: Sources of unsafety per repository (Removed outliers above 75 sources of unsafety. The full diagram is shown on the next slide.).

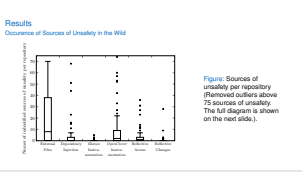


Figure: Sources of unsafety per repository (Removed outliers above 75 sources of unsafety. The full diagram is shown on the next slide.).

Results

Occurence of Sources of Unsafety in the Wild

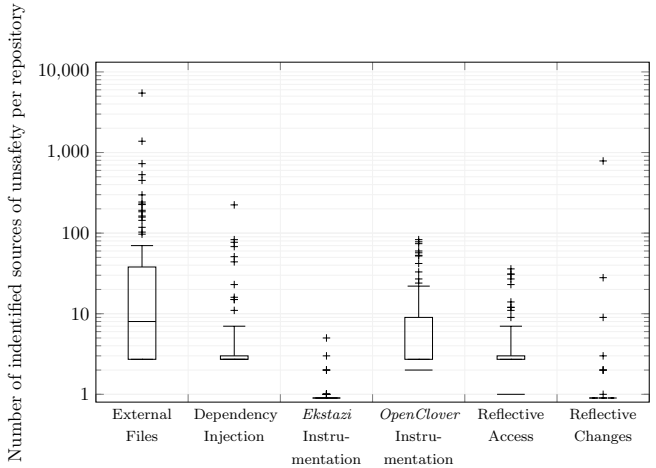
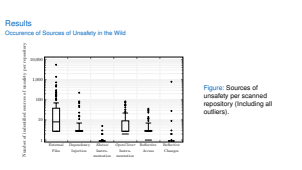


Figure: Sources of unsafety per scanned repository (Including all outliers).



- Box plot showing the number of sources of unsafety per repository
- Y-scale is logarithmic
- Big outliers for external files and reflective accesses

This work is licensed under a Creative Commons “Attribution 4.0 International” license.