Checking Memory Safety of Level 1 Safety-Critical Java Programs using Static-Analysis without Annotations

Chris Marriott

University of York, UK

Thesis Seminar - July 2014
Outline

1. Introduction
2. Memory safety of safety-critical Java programs
3. \textit{SCJ-mSafe}: An abstract language for memory-safety checking
4. \textit{SCJ-mSafe} program analysis
5. Checking \textit{SCJ-mSafe} programs
6. Experiments and results
7. Conclusion and future work
1. Introduction

2. Memory safety of safety-critical Java programs

3. SCJ-mSafe: An abstract language for memory-safety checking

4. SCJ-mSafe program analysis

5. Checking SCJ-mSafe programs

6. Experiments and results

7. Conclusion and future work
SCJ is a subset of Java designed for safety-critical applications.

Being developed by The Open Group (JSR-302).

Designed to allow programs to be certified to DO-178B Level A.

SCJ has a new programming paradigm based on missions and handlers:

- Level 0 - Cyclic executive with Periodic handlers
- Level 1 - Periodic and aperiodic handlers executing concurrently
- Level 2 - Concurrent missions and real-time threads

We are interested in memory safety of Level 1 SCJ programs (similar in complexity to Ravenscar Ada).
Hypothesis

Thesis hypothesis

*It is possible to produce an automatic static checking technique for valid Level 1 Safety-Critical Java programs to identify possible unsafe uses of memory at the source-code level, without the need for additional user-added annotations.*
1 Introduction

2 Memory safety of safety-critical Java programs

3 SCJ-mSafe: An abstract language for memory-safety checking

4 SCJ-mSafe program analysis

5 Checking SCJ-mSafe programs

6 Experiments and results

7 Conclusion and future work
RTSJ and SCJ

**RTSJ**

- Designed to address the limitations found in standard Java for real-time programs.
  - High-resolution time
  - Scheduling
  - Real-time threads
  - Asynchronous events
- Scoped memory model

**SCJ**

- Designed with certification in mind
- More restricted than RTSJ
- New programming paradigm based on missions and handlers
- Scoped memory model
Scoped memory model of RTSJ

• Immortal memory
  • Lasts for the entire length of program execution
  • Objects cannot be removed.
  • No garbage collection

• Scoped memory
  • Have a specific lifetime
  • Areas are cleared out when there are no more schedulable objects executing in them.
  • Cactus-stack structure with single parent
  • No garbage collection

• Usable heap
Scoping memory model of SCJ

- Immortal memory
  - Like in RTSJ
- Scoped memory
  - Have a specific lifetime (Mission, Handler, Temporary)
  - Areas are cleared out at the end of the execution of the paradigm component / runnable object.
  - No garbage collection
- No Heap
SCJ memory model (cont.)

Safelet

Mission Sequencer

Mission

initialize

enterPrivate Memory

Handler 1

handleEvent

enterPrivate Memory

enterPrivate Memory

Handler 2

handleEvent

IMem

MMem

PRMem

TPMem
Memory safety

A program is memory safe if there are no possible execution paths that lead to the dereference of a reference variable whose associated object resides in a memory area that has been cleared out.

- Strict memory model does not ensure memory safety by construction.
- Scoped memory introduces the possibility of dangling references.
- References must not point down the memory structure.
- Any reference down the memory structure is considered a memory-safety violation.

Our definition

A program is memory safe if there are no possible execution paths that may lead to a reference variable pointing to an object in a memory area that is lower in the hierarchy, that is, further away from the immortal memory area.
Verifying memory safety in RTSJ and SCJ

### RTSJ
- Restricted programming models [6]
- Type systems with memory property annotations [7]
- Type systems with ownership relations [2]
- Dynamic logic with invariant annotations [1]
- Bytecode analysis [9]

### SCJ
- SCJ annotation checking [10]
- Correctness by construction [3]
- Model checking [5]
- Bytecode analysis [4]
- Hardware-based run-time checking [8]
Current Section

1. Introduction

2. Memory safety of safety-critical Java programs

3. \textit{SCJ-mSafe}: An abstract language for memory-safety checking

4. \textit{SCJ-mSafe} program analysis

5. Checking \textit{SCJ-mSafe} programs

6. Experiments and results

7. Conclusion and future work
Our approach

SCJ Program → SCJmSafe Program → Method Properties → Environment

- Translation Strategy
- Analysis Strategy
- Memory Inference

Safe!

Possible Memory Safety Violations
public class MyHandler extends PeriodicEventHandler {
    
    public void handleAsyncEvent() {
        A a = new A();
        MyRunnable myRun = new MyRunnable(a);
        ManagedMemory.enterPrivateMemory(1, myRun);
    }

    class MyRunnable implements Runnable {
        A aField;

        public MyRunnable(A arg) {
            aField = arg;
        }

        public void run() {
            aField.field = new Object();
        }
    }
}
The **SCJ-mSafe** language

- Designed to ease verification
- Defines SCJ programs in a more abstract way
- Uniform structure
- Individual definitions for all SCJ paradigm components
- Simple commands and expressions
Overview of translation

- Input SCJ program must be well formed and type correct.
- No information relevant to memory safety is lost.
- Components that do not impact memory safety are abstracted away (side effects are still analysed).
  - Annotations
  - Unary/Binary operations
  - InstanceOf comparisons
  - Literals
  - Assert statements
- Each program is described in the same style.
  - Programs are easier to read and analyse.
  - Eases formalisation
- Translation formalised in Z
Expressions and Commands

- Not all SCJ expressions and commands are required in SCJ-mSafe.

Expressions
- Important expressions are left expressions.
- Expressions with side effects are translated separately.

Commands
- Additional commands are included in SCJ-mSafe:
  - Assignments
  - New instantiations
  - Method invocations
- These modify the value of program variables.
  - Better characterised as commands semantically
Example - *SCJ-mSafe*

```
handler MyHandler {
  fields { }
  init { }
  handleEvent {
    A a;
    NewInstance(a, Current, A, ());
    MyRunnable myRun;
    NewInstance(myRun, Current, MyRunnable, (a));
    EnterPrivateMemory(myRun);
  }
}
```
1 class MyRunnable {
2     fields {
3         A aField;
4     }
5     init { }
6     constr (arg) {
7         aField = arg;
8     }
9     method run() {
10        NewInstance(aField.field, Current, Object, ());
11    }
12 }
Translation strategy

- Functions define the translation from an \textit{SCJProgram} to an \textit{SCJmSafeProgram}

\[
\text{Translate} : \text{SCJProgram} \rightarrow \text{SCJmSafeProgram}
\]

\[
\forall \text{ program} : \text{dom Translate} \\
| \text{ program} \in \text{WellTypesProgs} \\
\exists \text{ scjmsafe} : \text{SCJmSafeProgram} \\
\exists \text{ scjSafelet} : \text{SCJClass} \\
| \text{ scjSafelet} \in \text{program.classes} \\
\wedge \text{ scjSafelet}.\text{extends} = \text{safelet} \\
\text{ scjmsafe.safelet} = \text{TranslateSafelet}(...)
\]

- Compositional translation

- \textit{TranslateMission}(...), \textit{TranslateHandler}(...), \textit{TranslateClass}(...), ...
Two functions to translate expressions:

\[\text{TranslateExpression} : \text{SCJExpression} \mapsto \text{Com}\]

\[\text{dom TranslateExpression} \subseteq \text{WellTypedExprs} \wedge \forall \text{scjExpr} : \text{dom TranslateExpression} \]

\[\exists (\exists e_1, e_2 : \text{SCJExpression} \mid \text{scjExpr} = \text{assignment}(e_1, e_2) \wedge (\text{let lexpr} == \text{ExtractExpression} e_1 \wedge (\text{let rexpr} == \text{ExtractExpression} e_2 \wedge (... \wedge (\text{TranslateExpression scjExpr} = \text{Seq}((\text{TranslateExpression e2}), (\text{Asgn}(\text{lexpr}, \text{rexpr)}))))))))\]

- \(a = (b = c)\); translated to \((b = c; a = b;)\)
Extracting expressions

\[ \text{ExtractExpression} : \text{SCJExpression} \rightarrow \text{Expr} \]

\[ \text{dom ExtractExpression} \subset \text{WellTypedExprs} \]
\[ \land \forall \text{scjExpr} : \text{dom ExtractExpression} \]
\[ \bullet \ldots \]
\[ \lor (\exists \text{e1, e2} : \text{SCJExpression}) \]
\[ | \text{scjExpr} = \text{assignment}(\text{e1, e2}) \]
\[ \bullet \text{ExtractExpression scjExpr} = \text{ExtractExpression e1} \]
\[ \ldots \]
\[ \lor (\exists \text{name : Name; id : Identifier}) \]
\[ | \text{scjExpr} = \text{identifier name} \land \text{id} = \text{VariableName name} \]
\[ \bullet \text{ExtractExpression scjExpr} = \text{ID id} \]

\[ (a = (b = c);) \rightarrow (b = c; a = b;) \]
\[ \text{ExtractExpression}(a) = a \]
\[ \text{ExtractExpression}(b = c) = b \]
Translating commands

\[ \text{TranslateCommand} : \text{SCJCommand} \rightarrow \text{Com} \]

\[ \text{dom TranslateCommand} \subseteq \text{WellTypedComs} \]
\[ \land (\forall \text{scjCom} : \text{dom TranslateCommand} \]
\[ \quad \bullet \ldots \]
\[ \quad \lor (\exists \text{e1 : SCJExpression; c1, c2 : SCJCommand} \]
\[ \quad \quad | \text{scjCom} = \text{if(e1, c1, c2)} \]
\[ \quad \quad \bullet \text{TranslateCommand scjCom} = \]
\[ \quad \quad \quad \text{Seq}((\text{TranslateExpression e1}), \]
\[ \quad \quad \quad \quad (\text{If}((\text{ExtractExpression e1}), \]
\[ \quad \quad \quad \quad \quad (\text{TranslateCommand c1}), \]
\[ \quad \quad \quad \quad (\text{TranslateCommand c2})))))) \]

\bullet \text{if (x) then y else z translated to} \]
\[ \text{Seq}((\text{SideEffects}(x), \text{if}(x) \text{ then Translation(y) else Translation(z)))) \]
Automatic translation

- Automatic translation from SCJ to *SCJ-mSafe* implemented
- Implementation is based on the formalised translation strategy.
- Uses the compiler-tree API to analyse SCJ programs
- Translation based on tree visitors for individual components
- Successfully applied to all examples we can find
1. Introduction
2. Memory safety of safety-critical Java programs
3. SCJ-mSafe: An abstract language for memory-safety checking
4. SCJ-mSafe program analysis
5. Checking SCJ-mSafe programs
6. Experiments and results
7. Conclusion and future work
Environment

- Allows us to maintain a model of information required to reason about memory safety
- Records information about left expressions that reference objects
- Share relations identify expressions that reference the same object
- Reference sets describe the set of reference contexts in which objects may reside.
- Updated after every command or expression
- Used by the inference rules to check for possible violations
- Worst-case analysis: false-negatives may be raised.
Environment - Aliasing example

```
1 ... 
2 o1 = new Object();
3 o2 = new Object();
4 a = o1;
5 o1.field = o2;
6 ...
```

Environment

```
{ a↦→ o1, o1.field↦→ o2 }
  ↦{ o1↦→ {MMem}, o2↦→ {MMem}, a↦→ {MMem},
      o1.field↦→ {MMem} }
```
Environment - Aliasing example (cont.)

1 ... 
2 \( o_1 = \text{new Object}(); \)
3 \( o_2 = \text{new Object}() \)
4 \( a = o_1; \)
5 \( o_1.\text{field} = o_2; \)
6 ... 
7 \( a.\text{field}.\text{var} = \ldots \)
8 ... 

A more precise Environment

\[
\{ a \mapsto o_1, o_1.\text{field} \mapsto o_2, a.\text{field} \mapsto o_2 \} \\
\mapsto \{ o_1 \mapsto \{ \text{MMem} \}, o_2 \mapsto \{ \text{MMem} \}, a \mapsto \{ \text{MMem} \}, \\
\quad o_1.\text{field} \mapsto \{ \text{MMem} \}, a.\text{field} \mapsto \{ \text{MMem} \} \}
\]
Let’s assume that we have a variable \( a \) that references an object that resides in \( IMem \):

```java
1  ...
2  Object o;
3  if (x > y) {
4      o = new Object();
5  } else {
6      MemoryArea memArea =
7          MemoryArea.getMemoryArea(a);
8      o = memArea.newInstance(Object.class);
9  }
10 ...
```

Environment:

\[
\{
\} \mapsto \{ a \mapsto \{ IMem \}, \text{memArea} \mapsto \{ IMem \}, o \mapsto \{ MMem, IMem \}\}
\]
Let's assume that we have two variables o1 and o2 that point to objects that reside in \textit{MMem} and \textit{IMem} respectively:

\begin{verbatim}
1  ...
2  if (x > y) {
3      ref = o1;
4  } else {
5      ref = o2;
6  }
7  ...
\end{verbatim}

### Environment

\begin{align*}
\{\{\text{ref} \mapsto o1\} & \mapsto \{o1 \mapsto \{\text{MMem}\}, o2 \mapsto \{\text{IMem}\}, \text{ref} \mapsto \{\text{MMem}\}\}, \\
\{\text{ref} \mapsto o2\} & \mapsto \{o1 \mapsto \{\text{MMem}\}, o2 \mapsto \{\text{IMem}\}, \text{ref} \mapsto \{\text{IMem}\}\}\end{align*}
Environment - formalisation

Share relation

\[ \text{ExprShareRelation} == \text{LExpr} \leftrightarrow \text{LExpr} \]

Reference set

\[ \text{ExprRefSet} == \text{LExpr} \rightarrow \mathcal{P} \text{ RefCon} \]

Environment

\[ \text{Env} == \{ \text{env} : \text{ExprShareRelation} \rightarrow \text{ExprRefSet} \mid \forall \text{rel} : \text{ExprShareRelation}; \text{ref} : \text{ExprRefSet} \mid (\text{rel}, \text{ref}) \in \text{env} \}
\]

\[ \bullet \text{dom}((\text{rel} \ast \cup (\text{rel} \ast) \sim)) = \text{dom ref} \]

\[ \wedge (\forall e_1, e_2 : \text{LExpr} \mid e_1 \mapsto e_2 \in (\text{rel} \ast \cup (\text{rel} \ast) \sim) \]

\[ \bullet \text{ref } e_1 = \text{ref } e_2 \} \]
Updating the environment - Assignments

Assignment 1
- RHS is a value.
- \( x = 10 \)

Assignment 2
- LHS is a variable and RHS is not a value.
- \( x = a \)

Assignment 3
- LHS is a field access, and RHS is not a value.
- \( x.y = a \)

Assignment 4
- LHS is an array element, and RHS is not a value.
- \( x[y] = a \)
RHS is a value.

No changes to environment

Before

\[
\begin{align*}
\{x \mapsto x\} &\mapsto \{x \mapsto \{\text{Prim}\}\} \\
\end{align*}
\]

\(x = 10\)

After

\[
\begin{align*}
\{x \mapsto x\} &\mapsto \{x \mapsto \{\text{Prim}\}\} \\
\end{align*}
\]
LHS is a variable and RHS is not a value.

Mapping from LHS to RHS added

Fields of RHS added as fields of LHS

RefCon of LHS is the RefCon of the RHS

Before

\[
\{x \mapsto y, \; x.field \mapsto z\} \mapsto \{x \mapsto \{PRMem \; h\}, \; x.field \mapsto \{MMem\}\}
\]

\[
x = a
\]

where \(a\) resides in \(MMem\) and has field \(b\) which references \(c\) in \(MMem\)

After

\[
\{x \mapsto a, \; x.b \mapsto c\} \mapsto \{x \mapsto \{MMem\}, \; x.b \mapsto \{MMem\}\}
\]
LHS is a field access, and RHS is not a value.

- Mapping from LHS to RHS added
- Fields of RHS added as fields of LHS
- RefCon of LHS is the RefCon of the RHS
- Expressions equal to containing object of LHS also updated

**Before**

\[
\{ x \mapsto p, \ x.y \mapsto q, \ p.y \mapsto q \} \\
\mapsto \{ x \mapsto \{ PRMem \ h \}, \ x.y \mapsto \{ PRMem \ h \}, \ p.y \mapsto \{ PRMem \ h \} \}
\]

- \( x.y = a \) where \( a \) resides in \( MMem \) and has field \( b \)

**After**

\[
\{ x \mapsto p, \ x.y \mapsto a, \ p.y \mapsto a, \ x.y.b \mapsto a.b, \ p.y.b \mapsto a.b, \ldots \} \\
\mapsto \{ x \mapsto \{ PRMem \ h \}, \ x.y \mapsto \{ MMem \}, \ p.y \mapsto \{ MMem \}, \ x.y.b \mapsto \{ MMem \}, \ p.y.b \mapsto \{ MMem \} \}
LHS is an array element, and RHS is not a value.
Mapping from LHS to RHS added
Fields of RHS added as fields of LHS
RefCon of LHS updated to include the RefCon of the RHS
Expressions equal to containing object of LHS also updated

Before

\[
\{x[Val] \mapsto p, \ x[Val] \mapsto q\} \\
\mapsto \{x \mapsto \{MMem\}, x[Val] \mapsto \{MMem\}\}
\]

\[
x[y] = a \text{ where } a \text{ resides in IMem}
\]

After

\[
\{x[Val] \mapsto p, \ x[Val] \mapsto q\} \\
\mapsto \{x \mapsto \{MMem\}, x[Val] \mapsto \{IMem, MMem\}\}
\]
Method Properties

- Methods may be safe in one allocation context, but not another.
- We do not restrict methods to a particular context.
- Methods are analysed to create a set of parametrised properties for each one.
- Properties are generated independently of execution context.
- Methods may be unsafe in any context.
- At analysis time, we resolve method calls by extracting information from the left expression and the types of arguments.
- If more than one method matches the criteria, due to dynamic binding, all are analysed.
Meta-reference contexts allow us to describe properties independently of parameters.

Example

```java
public void myMethod(A a, A b) {
    a.x = new customClass();
    b.x = a.x;
}
```

\{a.x \mapsto b.x\} \mapsto \{a.x \mapsto \{\text{Current}\}, \ b.x \mapsto \{\text{Erc a.x}\}\}

Methods

\[\text{Method} \equiv \text{Name} \times \text{Type} \times \text{seq Dec} \times \text{Com} \times ((\text{LExpr} \leftrightarrow \text{LExpr}) \rightarrow (\text{LExpr} \rightarrow \mathbb{P} \text{MetaRefCon}))\]
Automatic analysis

- Environment update functions for all *SCJ-mSafe* components implemented.
- Automatic generation of method properties implemented
- Program analysis to build and maintain the environment implemented
- Based on an stub SCJ infrastructure implementation
- Successfully applied to all examples we can find
Introduction

Memory safety of safety-critical Java programs

SCJ-\textit{mSafe}: An abstract language for memory-safety checking

SCJ-\textit{mSafe} program analysis

Checking \textit{SCJ-\textit{mSafe}} programs

Experiments and results

Conclusion and future work
∀(share, ref) : env •
∀(le₁, refSet₁) : ref •
  le₁ ∈ staticVars
  ∧ DominatesLeast(refSet₁) ⊸ IMem ∈ Dominates*
∧ ∀(le₁, refSet₁), (le₂, refSet₂) : ref •
  le₁ ≠ le₂ ∧ fieldOf(le₂, le₁)
  ∧ DominatesLeast(refSet₂) ⊸ DominatesTop(refSet₁) ∈ Dominates*
∧ ∀(le₁, refSet₁) : ref •
  le₁ ∈ localVariable
  ∧ DominatesLeast(refSet₁) ⊸ rc ∈ Dominates*

mSafeEnv(env, rc)
Example - *SCJ-mSafe*

```java
handler MyHandler {
    fields { }
    init { }
    handleEvent {
        A a;
        NewInstance(a, Current, A, ());
        MyRunnable myRun;
        NewInstance(myRun, Current, MyRunnable, (a));
        EnterPrivateMemory(myRun);
    }
}
```
```java
class MyRunnable {
    fields {
        A aField;
    }
    init {
    }
    constr (arg) {
        aField = arg;
    }
    method run() {
        NewInstance(aField.field, Current, Object, ());
    }
}
```
Run method properties

Analysing method 'run' in class MyRunnable
Method properties print-out....

Method Property:
{
  (aField.field -> aField.field)
}
->
{
  (aField.field -> {Current})
}
Example (cont.)

**Environment**

\[
\{(\text{sequencer.mission.handler.myRun.aField} \\
\quad \rightarrow \text{sequencer.mission.handler.a})
\}
\]

\[
\rightarrow \{
\quad (\text{sequencer} \rightarrow \text{IMem}) \\
\quad (\text{sequencer.mission} \rightarrow \text{MMem}) \\
\quad (\text{sequencer.mission.handler} \rightarrow \text{MMem}) \\
\quad (\text{sequencer.mission.handler.a} \rightarrow \text{PRMem(MyHandler)}) \\
\quad (\text{sequencer.mission.handler.a.field} \rightarrow \text{PRMem(MyHandler)}) \\
\quad (\text{sequencer.mission.handler.myRun} \rightarrow \text{PRMem(MyHandler)}) \\
\quad (\text{sequencer.mission.handler.myRun.aField} \\
\quad \rightarrow \text{PRMem(MyHandler)}) \\
\quad (\text{sequencer.mission.handler.myRun.aField.field} \\
\quad \rightarrow \text{TPMem(MyHandler, 0)})
\]

Chris Marriott - University of York, UK

Checking SCJ-mSafe programs

Thesis Seminar 41 / 44
Checking to see if field sequencer.mission.handler.myRun.aField.field of sequencer.mission.handler.myRun.aField is safe in current env entry

POSSIBLE MEMORY SAFETY VIOLATION - The field 'sequencer.mission.handler.myRun.aField.field' may reference an object stored in 'TPMem(MyHandler, 0)' when its containing object 'sequencer.mission.handler.myRun.aField' resides in 'PRMem(MyHandler)'
1 Introduction

2 Memory safety of safety-critical Java programs

3 SCJ-mSafe: An abstract language for memory-safety checking

4 SCJ-mSafe program analysis

5 Checking SCJ-mSafe programs

6 Experiments and results

7 Conclusion and future work
Tool and experiments

- Automatic translation and checking implemented
- Applied to CDx, PapaBench, ACCS, Pacemaker, and all case studies published in other approaches.
- No need for annotations
- No need for class duplication
- All memory-safety violations found
- No false-negatives raised in any programs analysed
- False-negatives raised in other techniques
- Memory-safety violation in CDx
1 Introduction

2 Memory safety of safety-critical Java programs

3 SCJ-mSafe: An abstract language for memory-safety checking

4 SCJ-mSafe program analysis

5 Checking SCJ-mSafe programs

6 Experiments and results

7 Conclusion and future work
Conclusions and future work

Contributions

- Abstraction technique from SCJ to SCJ-mSafe
- Identified rules to define memory safety
- Formalised the translation and checking techniques in Z
- Implemented a tool to automatically translate and check programs
- Successfully checked examples without annotations or class duplication

Future work

- Identify programming patterns that are always memory safe/unsafe
- Minimize false negatives
- Prove soundness
   P.h.: The key tool, integrating object oriented design and formal verification. software and systems modeling 4, 2005.

   Ownership types for safe region-based memory management in real-time Java.

   Safety-Critical Java in circus.


A type system to assure scope safety within Safety-Critical Java modules.  

Hardware support for safety-critical java scope checks.  
Proving the absence of rtsj related runtime errors through data flow analysis.

Static checking of Safety-Critical Java annotations.