Trace Effects and Object Orientation

Christian Skalka

Department of Computer Science
The University of Vermont
Foundations for Security

This talk is about mathematically well-founded techniques for enforcing security in software:

- Theory contributes to practice
  - Strong mathematical foundations provide clear guarantees about program behavior.

Specifically, we consider automatic type analysis for enforcing *programming language-based security*.

- Security mechanisms integrated into language design and implementation.
- Application programmers can specify and react to security policies.
Language Based Security

Our focus: large class of program safety and security properties expressible as *regular sequences of events*:

- File usage protocols (open before read or write)
- Memory allocation before memory usage
- Access control: privilege activation before privileged action

A variety of proposed enforcement mechanisms exists: program monitors, security automata, checks in program logics.
Trace based security models provide a general linguistic framework for enforcing program properties.

Language model is based on two fundamental abstractions:

- **Traces of events**
  - Events are explicit records of program actions, inserted by programmer or compiler
  - Traces are a component of the run-time system maintaining ordered sequence of events as they’re encountered

- **Trace assertions**
  - Local or global assertions expressible in temporal logic
Static Enforcement of Trace Properties

Trace based properties can be enforced statically, by a two-phase process:

1. Static program abstractions conservatively approximate dynamic program traces
2. Automatic analysis of program abstractions allows verification of temporal logic-specified trace properties

Conservative approximation = verification guarantees that trace properties will hold (not necessarily the other way around).

We propose a type and effect analysis to construct program abstractions, coupled with model checking for property verification.
Related Work

Previous related proposals for static enforcement of trace-based properties expressed in temporal logic:

Atsushi Igarashi and Naoki Kobayashi, Resource Usage Analysis, POPL02

K. Marriott, P. J. Stuckey and M. Sulzmann, Resource Usage Verification, APLAS03

Christian Skalka and Scott Smith, History Effects and Verification, APLAS04

- Type-theoretic program abstractions for core functional language

F. Besson and T. Jensen and D. Le Métayer and T. Thorn, Model checking security properties of control flow graphs, J. Computer Security

- Control flow graph abstraction (not higher-order analysis)
Trace Properties and Object Orientation

Goal of current research: to lift static analysis of trace based properties to an Object Oriented language model.

To address fundamental OO features, we consider Featherweight Java (FJ), extended with events, yielding $FJ_{sec}$:

\[ ev[i] \quad i \text{ is an identifier} \]

Events annotate program points in $FJ_{sec}$ code:

```java
class Example extends Object {
    void genInt()
    {
        ev[genIntInvoked];
        ev[createInt];
        return new Int(0);
    }
}
```
Object Oriented Language Model: FJ$_{sec}$

The machine model incorporates event traces $ev[1]; \ldots; ev[n]$, denoted $\eta$, in configurations $(\eta, e)$.

The run-time semantics of FJ$_{sec}$ are defined via an evaluation relation on configurations:

$$\varepsilon, (\text{new Example()}).\text{genInt()} \rightarrow^* \text{ev[genIntInvoked]; ev[createInt], new Int(0)}$$

Events are accrued in program traces in the order in which they’re encountered.
Static Trace Approximations for FJ_{sec}

A type and effect *reconstruction* algorithm statically infers approximations of program event traces.

\[ \Gamma \vdash_W e : T/C, H \quad \text{type judgement} \]

- Trace effects $H$ approximate program traces
- Object type forms $[T D]$ combine nominal typing and higher-order effects:
  - $D$ is object class name
  - $T$ lists object method types $\overline{T} \xrightarrow{H} S$ with their *latent effects*
Trace Effects

Trace effect program abstractions are *label transition systems (LTSs)*, similar to basic process algebras:

\[
\begin{align*}
ev[i] & \quad \text{single event} \\
H;H & \quad \text{ordered sequence} \\
H|H & \quad \text{nondeterministic choice} \\
h & \quad \text{abstract effect} \\
\mu h.H & \quad \text{recursive trace effect}
\end{align*}
\]

- Sequencing reflects order of evaluation
- Nondeterministic choice for “may analysis” of e.g. conditionals
- Recursive effects for possibly recursive methods
Trace Effect Semantics

Trace effects are endowed with an operational semantics, where transitions are labelled:

\[ a ::= \text{ev}[c] \mid \varepsilon \]

\[
\begin{align*}
\text{ev}[c] & \xrightarrow{\text{ev}[c]} \varepsilon \\
H_1;H_2 & \xrightarrow{\varepsilon} H_1 \\
H_1;H_2 & \xrightarrow{\varepsilon} H_2 \\
\mu h.H & \xrightarrow{\varepsilon} H[\mu h.H/h] \\
\varepsilon;H & \xrightarrow{\varepsilon} H \\
H_1;H_2 & \xrightarrow{a} H_1'; H_2 \text{ if } H_1 \xrightarrow{a} H_1' \\
\end{align*}
\]

- The string of labels generated by a sequence of transitions is a trace.
- Nondeterminism means that multiple traces can be generated by the same effect.
Trace Effect Approximations

The *interpretation* of an effect $H$, denoted $\lfloor H \rfloor$, is the set of traces that can be generated by reduction of $H$.

$$H \triangleq \mu h.\text{ev}[1] \mid (\text{ev}[2]; h)$$

$$\lfloor H \rfloor = \{ \text{ev}[1], \text{ev}[2]\text{ev}[1], \text{ev}[2]\text{ev}[2]\text{ev}[1], \ldots \}$$

**Correctness of analysis:** If $\Gamma \vdash W e : T/C$, $H$ and $\varepsilon, e \rightarrow^* \eta, e'$, then $\eta \in \lfloor H \rfloor$.

- Trace effects of expressions approximate their run-time traces, *including checks in context*. 
Subtyping and Trace Effects

Subtyping is a useful technique for OO programming, due to the presence of inheritance.

By type subsumption, if $e : S$ and $S$ is a subtype of $T$, written $S <: T$, then $e : T$.

To extend subtyping to a trace effect setting, \textit{method subtyping} must take latent effects into account:

$$\overline{S} \xrightarrow{H_1} S <: \overline{T} \xrightarrow{H_2} T$$

If $H_1$ approximates the effect of a method, then $H_2$ must be at least as approximate, or else relation is unsound.
Subtyping and Trace Effects

Method subtyping must ensure that method effects are not lost in subsumption, conserving approximations. Hence:

\[
\bar{T} <: \bar{S} \quad S <: T \quad [H_1] \subseteq [H_2]
\]

\[
\bar{S} \xrightarrow{H_1} S' <: \bar{T} \xrightarrow{H_2} T'
\]

Trace effects integrate naturally with subtyping via LTS interpretation.

Consider: as a natural extension to Java type system, the effect of a method \( m \) in any class \( C \) is the nondeterministic join of all versions of \( m \) in subclasses of \( C \).

The right definition of subtyping for \( FJ_{sec} \); objects can assume the type of their superclasses. But...
The Problem with Object Orientation

An extension of subtyping is not enough.

Trace effect analysis is significantly complicated by interaction with Object Oriented features, addressed by trace effect analysis:

- Inheritance, override, and dynamic dispatch
  - Parametric *effect polymorphism* allows flexibility for analysis of dynamically dispatched methods.

- Object *self* reference, subtyping, object downcasts
  - *Recursive constraint type representation* allows selective “pruning” of constraint graph for sound analysis.
Trace Effects and Dynamic Dispatch

To illustrate, consider the example of *history-based access control*. A code is signed by its owner, and access control lists associate owners with their authorized privileges.

\[
P \quad \text{code owners/signers}
\]
\[
R \quad \text{privileges}
\]
\[
\mathcal{A} \quad \text{access control lists}
\]

An access control check \( \text{demand}(R) \) ensures that all code affecting control flow so far is signed by owners authorized for \( R \).

\[ \text{demand} \quad \text{access control check} \]

*Martín Abadi and Cédric Fournet. Access Control Based on Execution History, NDSS03.*
Trace Effects and Dynamic Dispatch

If $P_1, \ldots, P_n$ are the sequence of owners signing code affecting the flow of control:

$$ \text{demand}(R) \iff R \in A(P_1) \cap \cdots \cap A(P_n) $$

In FJ$_{sec}$, this model can be implemented by annotating methods with an initial code signing event, and demand defined in temporal logic.

For example, imagine:

$$ A(\text{System}) = \{\text{FileWrite, FileRead, \ldots}\} $$
$$ A(\text{Applet}) = \emptyset $$
Trace Effects and Dynamic Dispatch

Continuing the example, imagine:

```java
class Writer extends Object {
    void safewrite(Formatter x, File f) {
        System;
        String s = x.format();
        demand(FileWrite);
        fwrite(s, f);
    }
}
```

At run-time:

- The check will succeed only if the dispatched version of `Formatter.format` is owned by a principal authorized for `FileWrite`.
- If `x` is a `System`-owned object, the check succeeds. If `x` is an `Applet`-owned object, the check fails.
Trace Effects and Dynamic Dispatch

The method `safewrite` can be statically assigned the following effect, where \( H \) *is the effect of* `x.format`

\[
\text{System;}H;\text{demand(FileWrite)}
\]

The problem is, what is \( H \)?

- In the presence of dynamic dispatch, `x.format` could be any version of `format`.
- In a naive extension of subtyping to FJ\(_{sec}\), \( H \) would be the nondeterministic choice of the effects of all versions of `format`. 
Trace Effects and Dynamic Dispatch

*But*, imagining:

```java
class Formatter extends Object {
    String format() { System; ... }
}

class AppFormatter extends Formatter {
    String format() { Applet; ... }
}
```

This implies \( H = (\text{System}|\text{Applet}) \), so the effect of \texttt{Writer.safewrite} is:

```java
System;(\text{System}|\text{Applet});demand(\text{FileWrite})
```

Thus, *any invocation* of \texttt{Writer.safewrite} will be *statically rejected*. 
Trace Effects and Dynamic Dispatch

A naive subtyping (“join of all version effects”) approach has the following fatal flaws:

- Authorization levels of dynamically dispatched methods will be determined by the least authorized version in the inheritance hierarchy.
- All code must be known in advance, precluding modularity.

*Our solution:* exploit parametric polymorphism. Assign abstract effects $h$ to dynamically dispatched methods.
Polymorphic Effects For Dynamic Dispatch

class Writer extends Object {
  void safewrite(Formatter x, File f) {
    System;
    String s = x.format();
    demand(FileWrite);
    write(s, f);
  }
}

The type assigned to x.format is abstract in its effect:

\[ x.\text{format} : () \rightarrow ^h \text{StringT} \]

\[ \text{Writer.safewrite} : \forall h. (...) \rightarrow ^{\text{System};h;\text{demand(FileWrite)}} \rightarrow \text{void} \]

If Writer.safewrite is invoked on a Formatter object:

\[ \text{System};\text{System};\text{demand(FileWrite)} \]

If Writer.safewrite is invoked on a AppFormatter object:

\[ \text{System};\text{Applet};\text{demand(FileWrite)} \]
Subtyping and Downcasts

Suppose Triangle <: Polygon and e : T, where e may evaluate to either a Triangle or Polygon, as reflected in constraints on T (or, constraint representation of T):

\[[S \text{ Polygon}] <: T \quad \text{[R Triangle]} <: T\]

Cast checking ensures that a downcast of the form (Triangle)e will be stuck if e does not evaluate to a Triangle.

Supposing (Triangle)e : [T’Triangle]:

- There exists a relation between T and [T’Triangle], but...

- Type soundness does not allow T <: [T’Triangle]; allows only selective flow from T to [T’Triangle].
Soft Subtyping for Downcasts

Our solution is to impose a *soft subtyping* relation between $T$ and $[T’\text{Triangle}]$:

$$T \preccurlyeq [T’\text{Triangle}]$$

Recalling:

$$[S \text{Polygon}] \ll T \quad [R \text{Triangle}] \ll T$$

The relation $T \preccurlyeq [T’\text{Triangle}]$ entails $R \ll T’$, *not* $S \ll T’$.

- Constraint representation of types allows “selective pruning” of constraint graph for soft subtyping.

- Soft subtyping also allows for greater precision in analysis, ignoring program flow that would cause cast checking failure at run-time.
Effect Transformations for Scalability*

Trace effects generated by type reconstruction can be post-processed to analyze variations on the core language.

- Theoretical interest: core analysis adaptable to non-trivial variations
- Uniform analysis for treating variations
- Possibly greater efficiency than composition with “direct” analysis of variations

*Joint work with Scott Smith (JHU), and David Van Horn (Brandeis)
Analysis of Stack Traces

In stack trace model, events occurring during function execution are “forgotten” when the function returns:

- Activations annotated with events; call-stack pop erases events
- Ubiquitous example: *Java stack inspection*

Post-processing of trace effects allows approximation of *stack traces*:

- Pushes and pops coincide with function scope
- Regularity of push and pop events allows stack contexts to be retrieved from trace effects
Stackification (Basic Idea)

Note: all methods are assigned a distinct $\mu$-scoped effect in effect reconstruction.

Stackification exploits this characteristic—$\mu$-scope delineates corresponding pushes and pops:

\[
\begin{align*}
\text{stackify}(ev[i];H) &= ev[i];\text{stackify}(H) \\
\text{stackify}((H_1 | H_2);H) &= \text{stackify}(H_1;H) \mid \text{stackify}(H_2;H) \\
\text{stackify}(h;H) &= h \mid \text{stackify}(H) \\
\text{stackify}((\mu h. H_1);H_2) &= (\mu h. \text{stackify}(H_1)) \mid \text{stackify}(H_2)
\end{align*}
\]

Example:

\[
\text{stackify}((\mu h. ev[1]|(ev[2];h));ev[3]) = (\mu h. ev[1]|(ev[2];h))|ev[3]
\]
Analysis of Exceptions

The effects of exceptions can be analyzed with the addition of two new *pre-effect* constructs and subsequent transformation.

```
throw  anonymous exception (and throw pre-effect)

try{e_1}catch{e_2}  exception handlers  H_1 \uparrow H_2  pre-effect of handlers
```

```
\ldots, throw \vdash_W throw : \ldots \quad \ldots, H_1 \vdash_W e_1 : \ldots \quad \ldots, H_2 \vdash_W e_2 : \ldots
\overline{\ldots, H_1 \uparrow H_2 \vdash_W try\{e_1\}catch\{e_2\} : \ldots}
```
Exception Transformation (Basic Idea)

The transformation separates a given trace effect into two sets of paths: paths that can end “safely”, and those that can end in a throw.

\[
\begin{align*}
\text{exnize}(\text{ev}[i]) & = \{\text{ev}[i]\}, \emptyset \\
\text{exnize}(\text{throw}) & = \emptyset, \{\varepsilon\} \\
\text{exnize}(\text{H}_1;\text{H}_2) & = \text{let } s_1, t_1 = \text{exnize}(\text{H}_1) \text{ in} \\
& \quad \text{let } s_2, t_2 = \text{exnize}(\text{H}_2) \text{ in} \\
& \quad \{\text{H}_1;\text{H}_2 \mid \text{H}_1 \in s_1 \text{ and } \text{H}_2 \in s_2\}, \\
& \quad \{\text{H}_1;\text{H}_2 \mid \text{H}_1 \in s_1 \text{ and } \text{H}_2 \in t_2\} \cup t_1 \\
\text{exnize}(\text{H}_1 \uparrow \text{H}_2) & = \text{let } s_1, t_1 = \text{exnize}(\text{H}_1) \text{ in} \\
& \quad \text{let } s_2, t_2 = \text{exnize}(\text{H}_2) \text{ in} \\
& \quad \{\text{H}_1;\text{H}_2 \mid \text{H}_1 \in t_1 \text{ and } \text{H}_2 \in s_2\} \cup s_1, \\
& \quad \{\text{H}_1;\text{H}_2 \mid \text{H}_1 \in t_1 \text{ and } \text{H}_2 \in t_2\}
\end{align*}
\]

Note: The \(\mu\) case complicates matters.
Exception Transformation Examples

\[(ev[1]; ev[2]; \text{throw}; ev[3]) | (ev[1]; ev[3])\]

\textit{safe:} \(ev[1]; ev[3]\) \hspace{1cm} \textit{throw:} \(ev[1]; ev[2]\)

\[
\mu_h.\text{throw}|ev[2]|(ev[1]; h)
\]

\textit{safe:} \(\mu_h.ev[2]|(ev[1]; h)\) \hspace{1cm} \textit{throw:} \(\mu_h.\text{throw}|(ev[1]; h)\)
Conclusion

Research focus: static analysis of trace based properties of Object Oriented programs.

- Type based trace effect analysis allows automatic approximation of program trace behavior, via type and effect inference.

- Application of particular type features allow precision and flexibility in static analysis of OO programs:
  - Polymorphism allows effect abstraction of dynamically dispatched methods.
  - Constraint type representation implements object subtyping, allows soft subtyping for sound analysis of downcasts.
Conclusion, Future Work

Research focus summary (contd.):

- **Type inference, conservation** means that analysis is backwards-compatible with existing Java codebase.

- Trace effects are subject to post-processing techniques that allow scalability to language extensions.
  - Stack-based security
  - Control flow operations (exceptions)

Topics for future work (short term):

- Extension to state, concurrency (threads)—**full Java**
- Testing, specialization of model checking techniques

[http://www.cs.uvm.edu/~skalka](http://www.cs.uvm.edu/~skalka)
Type Reconstruction (Full Detail)

T-Var
\[ \Gamma ⊢ W(x : \Gamma(x)/true, \varepsilon) \]

T-Field
\[ \Gamma ⊢ W(e : [TC]/C,H \quad D \cdot f \in fields(C)) \]
\[ \Gamma ⊢ W.e.f : [XD]/C \land T <:(f : [XD]), H) \]

T-Invk
\[ \Gamma ⊢ W(e : [TC]/C,H \quad \Gamma, \varnothing \Gamma ⊢ W(\bar{e} : \varnothing/\bar{C}, H) \]
\[ \Gamma, \varnothing \Gamma ⊢ W(\bar{e} \cdot m(\bar{e}) : [XD]/C \land D \land T <:(m : [SB] \rightarrow [XD]), H,H';h) \]

T-New
\[ \Gamma(C) = \forall \bar{X}.T \]
\[ \Gamma ⊢ W(e : S/C,H \quad T.f = \bar{T} \quad fields(C) = \bar{C} \bar{f}) \]
\[ \Gamma ⊢ W(new \ C(\bar{e}) : T[\bar{X}'/\bar{X}] / C \land D[\bar{X}'/\bar{X}] \land S <:(\bar{T}[\bar{X}'/\bar{X}], H) \]

T-Event
\[ \Gamma ⊢ W(\text{ev}[i] : Unit/false, ev[i]) \]

T-Check
\[ \Gamma ⊢ W(chk[i] : Unit/false, chk[i]) \]

T-Cast
\[ \Gamma ⊢ W(e : T/C,H) \]
\[ \Gamma ⊢ W(D)e : [XD]/C \land T \preceq [XD], H) \]

T-Meth
\[ \Gamma; \bar{x} : \bar{T} ⊢ W(e : S/C,H \quad \Gamma(\text{this}).m = \bar{T} h T \quad mb\text{ody}(m, C) = \bar{x}.e \]
\[ \Gamma, C \land S <:(\text{T} \land H <:(\text{h} \downarrow W \text{m}, C) : \bar{T} h S) \]

T-Class
\[ \Gamma; C : T; \text{this} : T, \bar{C} ⊢ W(m) : \bar{T} \quad T = [\bar{f} : \bar{R} \bar{m} : \bar{S} C] \text{ is abstract} \]
\[ \Gamma ⊢ W(C : \forall \bar{X}[\bar{C}], [\bar{f} : \bar{R} \bar{m} : \bar{T} C] \]
Exnization (Full Detail)

\[
\begin{align*}
\text{exnize}(\varepsilon) &= \{\varepsilon\}, \emptyset, \emptyset \\
\text{exnize}(\text{ev}[i]) &= \{\text{ev}[i]\}, \emptyset, \emptyset \\
\text{exnize}(\text{throw}) &= \emptyset, \{\varepsilon\}, \emptyset \\
\text{exnize}(h) &= \{h\}, \emptyset, \{h\} \\
\text{exnize}(H_1H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
&\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
&\quad s_1 \cup s_2, t_1 \cup t_2, r_1 \cup r_2 \\
\text{exnize}(H_1; H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
&\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
&\quad s_1; s_2, t_1; t_2, r_1 \cup (s_1; r_2) \\
\text{exnize}(H_1 \downarrow H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
&\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
&\quad s_1 \cup (t_1; s_2), t_1; t_2, r_1 \cup (t_1; r_2) \\
\text{exnize}(\mu h. H) &= \text{let } s, t, r = \text{exnize}(H) \text{ in} \\
&\quad \text{let } H_s = \mu h. \text{join}(s) \text{ in} \\
&\quad \text{let } r' = \text{map } (\lambda(H, h'). H[H_s/h]; h') \text{ in} \\
&\quad \text{let } r_h = \text{filter } (\lambda(H; h'). h' = h) \text{ in} \\
&\quad \text{let } t' = \text{map } (\lambda H. \mu h. \text{join}(H[H_s/h]) \cup r_h) \text{ in} \\
&\quad \{H_s\}, t', r' - r_h
\end{align*}
\]
Stackification (Full Detail)

\[ stackify(\varepsilon) = \varepsilon \]
\[ stackify(\varepsilon; H) = stackify(H) \]
\[ stackify(ev[i]; H) = ev[i]; stackify(H) \]
\[ stackify(h; H) = h|stackify(H) \]
\[ stackify((\mu_h.H_1); H_2) = (\mu_h.stackify(H_1)) \mid stackify(H_2) \]
\[ stackify((H_1|H_2); H) = stackify(H_1; H) \mid stackify(H_2; H) \]
\[ stackify((H_1; H_2); H_3) = stackify(H_1; (H_2; H_3)) \]
\[ stackify(H) = stackify(H; \varepsilon) \]