

# 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 Program Properties

*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<sub>sec</sub>*:

$ev[i]$       $i$  is an identifier

Events annotate program points in FJ<sub>sec</sub> code:

```
class Example extends Object {  
  void genInt(){  
    ev[genIntInvoked];  
    ev[createInt];  
    return new Int(0);  
  }  
}
```

## Object Oriented Language Model: FJ<sub>sec</sub>

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

The run-time semantics of FJ<sub>sec</sub> are defined via an evaluation relation on configurations:

$$\begin{array}{c} \varepsilon, (\text{new Example}()).\text{genInt}() \\ \longrightarrow^* \\ ev[\text{genIntInvoked}]; ev[\text{createInt}], \text{new Int}(0) \end{array}$$

Events are accrued in program traces in the order in which they're encountered.

## Static Trace Approximations for FJ<sub>sec</sub>

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

$$\Gamma \vdash_W e : T/C, H \quad \textit{type judgement}$$

$T$  *type*       $\Gamma$  *type environment*       $C$  *type constraints*       $H$  *trace effects*

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

## Trace Effects

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

$ev[i]$  *single event*       $H;H$  *ordered sequence*       $H|H$  *nondeterministic choice*  
 $h$  *abstract effect*       $\mu h.H$  *recursive trace effect*

- 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:

$$a ::= \text{ev}[c] \mid \varepsilon \quad \textit{labels}$$

$$\text{ev}[c] \xrightarrow{\text{ev}[c]} \varepsilon \quad H_1 | H_2 \xrightarrow{\varepsilon} H_1 \quad H_1 | H_2 \xrightarrow{\varepsilon} H_2 \quad \mu h.H \xrightarrow{\varepsilon} H[\mu h.H/h]$$

$$\varepsilon; H \xrightarrow{\varepsilon} H \quad H_1; H_2 \xrightarrow{a} H'_1; H_2 \text{ if } H_1 \xrightarrow{a} H'_1$$

- 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  $\llbracket H \rrbracket$ , is the set of traces that can be generated by reduction of  $H$ .

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

$$\llbracket H \rrbracket = \{ \text{ev}[1], \text{ev}[2] \text{ev}[1], \text{ev}[2] \text{ev}[2] \text{ev}[1], \dots \}$$

*Correctness of analysis:* If  $\Gamma \vdash_W e : T/C, H$  and  $\varepsilon, e \rightarrow^* \eta, e'$ , then  $\eta \in \llbracket H \rrbracket$ .

- 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, *method subtyping* must take latent effects into account:

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

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:

$$\frac{\bar{T} <: \bar{S} \quad S <: T \quad \llbracket H_1 \rrbracket \subseteq \llbracket H_2 \rrbracket}{\bar{S} \xrightarrow{H_1} S' <: \bar{T} \xrightarrow{H_2} T'}$$

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  $FJ_{\text{SEC}}$ ; 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* \*.

Code is signed by its *owner*, and access control lists associate owners with their authorized *privileges*.

P    code owners/signers  
R    privileges  
 $\mathcal{A}$     access control lists

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

`demand`    access control check

\* *Martín Abadi and Cédric Fournet. Access Control Based on Execution History, NDSS03.*

## Trace Effects and Dynamic Dispatch

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

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

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

For example, imagine:

$$\begin{aligned} \mathcal{A}(\text{System}) &= \{\text{FileWrite}, \text{FileRead}, \dots\} \\ \mathcal{A}(\text{Applet}) &= \emptyset \end{aligned}$$

## Trace Effects and Dynamic Dispatch

Continuing the example, imagine:

```
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`*:

```
System;H;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  $FJ_{\text{sec}}$ , `H` would be the nondeterministic choice of the effects of all versions of `format`.

## Trace Effects and Dynamic Dispatch

*But*, imagining:

```
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 `Writer.safeWrite` is:

```
System; (System|Applet); demand(FileWrite)
```

Thus, *any invocation* of `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:

$$x.format : () \xrightarrow{h} \text{StringT}$$
$$\text{Writer.safewrite} : \forall h. (\dots) \xrightarrow{\text{System}; h; \text{demand}(\text{FileWrite})} \text{void}$$

If `Writer.safewrite` is invoked on a `Formatter` object:

`System; System; demand(FileWrite)`

If `Writer.safewrite` is invoked on a `AppFormatter` object:

`System; Applet; demand(FileWrite)`

## Subtyping and Downcasts

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

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

Cast checking ensures that a *downcast* of the form  $(\text{Triangle})e$  will be stuck if  $e$  does not evaluate to a  $\text{Triangle}$ .

Supposing  $(\text{Triangle})e : [T' \text{Triangle}]$ :

- There exists a relation between  $T$  and  $[T' \text{Triangle}]$ , but...
- Type soundness does not allow  $T <: [T' \text{Triangle}]$ ; allows only *selective* flow from  $T$  to  $[T' \text{Triangle}]$ .

## Soft Subtyping for Downcasts

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

$$T \triangleleft [T' \text{ Triangle}]$$

Recalling:

$$[S \text{ Polygon}] \triangleleft T \qquad [R \text{ Triangle}] \triangleleft T$$

The relation  $T \triangleleft [T' \text{ Triangle}]$  entails  $R \triangleleft T'$ , *not*  $S \triangleleft T'$ .

- 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:

$$\begin{aligned} \mathit{stackify}(\mathit{ev}[i]; H) &= \mathit{ev}[i]; \mathit{stackify}(H) \\ \mathit{stackify}(H_1 | H_2; H) &= \mathit{stackify}(H_1; H) | \mathit{stackify}(H_2; H) \\ \mathit{stackify}(h; H) &= h | \mathit{stackify}(H) \\ \mathit{stackify}(\mu h.H_1; H_2) &= (\mu h. \mathit{stackify}(H_1)) | \mathit{stackify}(H_2) \end{aligned}$$

Example:

$$\mathit{stackify}(\mu h. \mathit{ev}[1] | (\mathit{ev}[2]; h)); \mathit{ev}[3]) = (\mu h. \mathit{ev}[1] | (\mathit{ev}[2]; h)) | \mathit{ev}[3]$$

## 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{e1}catch{e2}`    *exception handlers*             $H_1 \vec{\rightarrow} H_2$     *pre-effect of handlers*

$$\begin{array}{c} \dots, \text{throw} \vdash_W \text{throw} : \dots \\ \hline \dots, H_1 \vdash_W e_1 : \dots \quad \dots, H_2 \vdash_W e_2 : \dots \\ \hline \dots, H_1 \vec{\rightarrow} H_2 \vdash_W \text{try}\{e_1\}\text{catch}\{e_2\} : \dots \end{array}$$

## 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.

$$\begin{aligned} \text{exnize}(\text{ev}[\mathbf{i}]) &= \{\text{ev}[\mathbf{i}]\}, \emptyset \\ \text{exnize}(\text{throw}) &= \emptyset, \{\epsilon\} \\ \text{exnize}(H_1; H_2) &= \text{let } s_1, t_1 = \text{exnize}(H_1) \text{ in} \\ &\quad \text{let } s_2, t_2 = \text{exnize}(H_2) \text{ in} \\ &\quad \{H_1; H_2 \mid H_1 \in s_1 \text{ and } H_2 \in s_2\}, \\ &\quad \{H_1; H_2 \mid H_1 \in s_1 \text{ and } H_2 \in t_2\} \cup t_1 \\ \text{exnize}(H_1 \dot{\mapsto} H_2) &= \text{let } s_1, t_1 = \text{exnize}(H_1) \text{ in} \\ &\quad \text{let } s_2, t_2 = \text{exnize}(H_2) \text{ in} \\ &\quad \{H_1; H_2 \mid H_1 \in t_1 \text{ and } H_2 \in s_2\} \cup s_1, \\ &\quad \{H_1; H_2 \mid H_1 \in t_1 \text{ and } H_2 \in t_2\} \end{aligned}$$

Note: The  $\mu$  case complicates matters.

## Exception Transformation Examples

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

*safe:*  $\text{ev}[1]; \text{ev}[3]$       *throw:*  $\text{ev}[1]; \text{ev}[2]$

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

*safe:*  $\mu\text{h}.\text{ev}[2] \mid (\text{ev}[1]; \text{h})$       *throw:*  $\mu\text{h}.\text{throw} \mid (\text{ev}[1]; \text{h})$

## 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>

## Type Reconstruction (Full Detail)

$$\text{T-Var} \\ \Gamma \vdash_W x : \Gamma(x) / \mathbf{true}, \varepsilon$$

$$\text{T-Field} \\ \frac{\Gamma \vdash_W e : [\mathbf{TC}] / C, H \quad D \ f \in \mathit{fields}(C)}{\Gamma \vdash_W e.f : [\mathbf{XD}] / C \wedge T \triangleleft (f : [\mathbf{XD}]), H}$$

$$\text{T-Invk} \\ \frac{\Gamma \vdash_W e : [\mathbf{TC}] / C, H \quad \Gamma \vdash_W \bar{e} : [\bar{\mathbf{S}}\bar{\mathbf{B}}] / D, H' \quad \mathit{mtype}(m, C) = \bar{D} \rightarrow D \quad \bar{B} \triangleleft \bar{D}}{\Gamma \vdash_W e.m(\bar{e}) : [\mathbf{XD}] / C \wedge D \wedge T \triangleleft (m : [\bar{\mathbf{S}}\bar{\mathbf{B}}] \xrightarrow{h} [\mathbf{XD}]), H; H'; h}$$

$$\text{T-New} \\ \frac{\Gamma(C) = \forall \bar{X}[D]. T \quad \Gamma \vdash_W \bar{e} : \bar{S} / C, H \quad T.\bar{f} = \bar{T} \quad \mathit{fields}(C) = \bar{C} \ \bar{f}}{\Gamma \vdash_W \mathbf{new} \ C(\bar{e}) : T[\bar{X}' / \bar{X}] / C \wedge D[\bar{X}' / \bar{X}] \wedge \bar{S} \triangleleft \bar{T}[\bar{X}' / \bar{X}], H}$$

$$\text{T-Event} \\ \Gamma \vdash_W \mathbf{ev}[i] : \mathbf{Unit} / \mathbf{true}, \mathbf{ev}[i]$$

$$\text{T-Check} \\ \Gamma \vdash_W \mathbf{chk}[i] : \mathbf{Unit} / \mathbf{true}, \mathbf{chk}[i]$$

$$\text{T-Cast} \\ \frac{\Gamma \vdash_W e : T / C, H}{\Gamma \vdash_W (D)e : [\mathbf{XD}] / C \wedge T \triangleleft [\mathbf{XD}], H}$$

$$\text{T-Meth} \\ \frac{\Gamma; \bar{x} : \bar{T} \vdash_W e : S / C, H \quad \Gamma(\mathbf{this}).m = \bar{T} \xrightarrow{h} T \quad \mathit{mbody}(m, C) = \bar{x}.e}{\Gamma, C \wedge S \triangleleft T \wedge H \triangleleft h \vdash_W m, C : \bar{T} \xrightarrow{h} S}$$

$$\text{T-Class} \\ \frac{\Gamma; C : T; \mathbf{this} : T, \bar{C} \vdash_W \bar{m} : \bar{T} \quad T = [\bar{f} : \bar{R} \ \bar{m} : \bar{S} \ C] \text{ is abstract}}{\Gamma \vdash_W C : \forall \bar{X}[\bar{C}]. [\bar{f} : \bar{R} \ \bar{m} : \bar{T} \ C]}$$

## Exnization (Full Detail)

$$\begin{aligned}
 \text{exnize}(\varepsilon) &= \{\varepsilon\}, \emptyset, \emptyset \\
 \text{exnize}(\text{ev}[i]) &= \{\text{ev}[i]\}, \emptyset, \emptyset \\
 \text{exnize}(\text{throw}) &= \emptyset, \{\varepsilon\}, \emptyset \\
 \text{exnize}(\mathbf{h}) &= \{\mathbf{h}\}, \emptyset, \{\mathbf{h}\} \\
 \text{exnize}(H_1 | H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
 &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
 &\quad s_1 \cup s_2, t_1 \cup t_2, r_1 \cup r_2 \\
 \text{exnize}(H_1 ; H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
 &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
 &\quad s_1 ; s_2, t_1 \cup (s_1 ; t_2), r_1 \cup (s_1 ; r_2) \\
 \text{exnize}(H_1 \uparrow H_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(H_1) \text{ in} \\
 &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(H_2) \text{ in} \\
 &\quad s_1 \cup (t_1 ; s_2), t_1 ; t_2, r_1 \cup (t_1 ; r_2) \\
 \text{exnize}(\mu \mathbf{h}. H) &= \text{let } s, t, r = \text{exnize}(H) \text{ in} \\
 &\quad \text{let } H_s = \mu \mathbf{h}. \text{join}(s) \text{ in} \\
 &\quad \text{let } r' = \text{map } (\lambda(H, \mathbf{h}'). H[H_s/\mathbf{h}]; \mathbf{h}') r \text{ in} \\
 &\quad \text{let } r_{\mathbf{h}} = \text{filter } (\lambda(H; \mathbf{h}'). \mathbf{h}' = \mathbf{h}) r' \text{ in} \\
 &\quad \text{let } t' = \text{map } (\lambda H. \mu \mathbf{h}. \text{join}(\{H[H_s/\mathbf{h}]\} \cup r_{\mathbf{h}})) t \text{ in} \\
 &\quad \{H_s\}, t', r' - r_{\mathbf{h}}
 \end{aligned}$$

## Stackification (Full Detail)

$$\begin{aligned} \mathit{stackify}(\varepsilon) &= \varepsilon \\ \mathit{stackify}(\varepsilon; H) &= \mathit{stackify}(H) \\ \mathit{stackify}(\mathit{ev}[i]; H) &= \mathit{ev}[i]; \mathit{stackify}(H) \\ \mathit{stackify}(h; H) &= h | \mathit{stackify}(H) \\ \mathit{stackify}((\mu h. H_1); H_2) &= (\mu h. \mathit{stackify}(H_1)) | \mathit{stackify}(H_2) \\ \mathit{stackify}((H_1 | H_2); H) &= \mathit{stackify}(H_1; H) | \mathit{stackify}(H_2; H) \\ \mathit{stackify}((H_1; H_2); H_3) &= \mathit{stackify}(H_1; (H_2; H_3)) \\ \mathit{stackify}(H) &= \mathit{stackify}(H; \varepsilon) \end{aligned}$$