Algorithmic Trace Effect Analysis
Context Based Security in Programming Languages

David Van Horn
dvanhorn@cs.uvm.edu
`http://www.cs.uvm.edu/~dvanhorn/`

Christian Skalka
skalka@cs.uvm.edu
`http://www.cs.uvm.edu/~skalka/`

Abstract

Trace effect analysis is a programming language analysis for ensuring correct and secure behavior of software with respect to so-called temporal properties of programs. Temporal properties express the well-ordering of program events, such as when files are opened or when resources are obtained. By enforcing program event order specifications, we can ensure that files are opened before being read, or that privilege activation occurs before privileged resources are acquired. In general, the ability to enforce temporal properties in programs allows a wide variety of safety and security guarantees to be made about software.

Further, this analysis is performed automatically at compile time, i.e. it takes place before program execution occurs, leading to the earlier detection of errors. A consequence of such compile time, aka static reasoning, is that if a program is deemed correct by the analysis, it will never violate its specification at run time, ensuring safe program behavior in all possible circumstances of program execution.

Common security mechanisms such as stack inspection, found in the Java programming language and .NET framework, are enforced via run time inspection of the sequence of events occurring up to the point of the inspection. These mechanisms are formalizable within trace effect analysis and thus can be enforced at compile time, implying all run time inspections are superfluous and can be eliminated. Any run time overhead associated with enforcing the security mechanism can therefore also safely be eliminated.

Our analysis consists of a program logic (a system of axioms and deduction rules) such that if a program has a derivable logical judgment, the program meets its temporal specification.

Additionally we have defined an algorithm for discovering such a judgment for a program. This algorithm has been proved sound, which is to say any judgment found by the algorithm is a valid logical judgment and represents a proof that the program meets its specification. This proof discovery procedure has been implemented for a foundational programming language with notions of atomic events, temporal assertions, and computational traces (sequences of events).

Logical typing rules

Event
$$\frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; ev(s) \vdash ev(e) : \mathrm{unit}}$$

Weaken
$$\frac{\Gamma, H \vdash e : \tau \qquad H \preccurlyeq H'}{\Gamma, H' \vdash e : \tau}$$

If
$$\frac{\Gamma, H_1 \vdash e_1 : \mathrm{bool} \qquad \Gamma, H_2 \vdash e_2 : \tau \qquad \Gamma, H_2 \vdash e_3 : \tau}{\Gamma, H_1; H_2 \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \tau}$$

Abs
$$\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H} \tau_2, H \vdash e : \tau_2}{\Gamma, \epsilon \vdash \lambda_z x.e : \tau_1 \xrightarrow{H} \tau_2}$$

App
$$\frac{\Gamma, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \qquad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3 \vdash e_1 e_2 : \tau}$$

Let
$$\frac{\Gamma, \epsilon \vdash v : \tau' \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing \qquad \Gamma; x : \forall\bar{\alpha}.\tau', H \vdash e : \tau}{\Gamma, H \vdash \mathsf{let}\ x = v \,\mathsf{in}\, e : \tau}$$

Fundamental abstraction: Event traces

Many program correctness properties are expressible as properties of *program event traces.*

- Security handshake protocols, eg. SSL
- File open before read
- Allocate before use
- Access control: privilege activation before privileged action

Well-formedness of traces expressible and enforceable as program monitors or checks in program logics.

We take a *language-based approach*, integrated the necessary abstractions into a programming language so that a programmer can articulate temporal properties.

Our language is endowed with notions of *events*, and *checks*.

- An *event* is an abstract program action, parameterized by a static constant. They are inserted by the programmer, or compiler.
- A *check* is a predicate, expressed in a temporal logic, over possibly inifinite sequences of events called a *trace*.

Type analysis and model checking

Our approach is a synthesis of software verification methods. We use a type analysis with a rich notion of program safety to represent program abstractions. The abstractions are then model checked for verfication.

A polymorphic type and effect inference system automatically extracts a program abstraction conservatively approximating the events and assertions that will arise at run-time. Such an abstraction can then be model-checked to obtain a static verification of these temporal program logics for higher-order programs.

Effect transformations for Flexibility

Trace effects can be post-processed to analyze variations to the core language.

- *Simplification*: Traces may be simplified in a semantic-preserving way in order to improve model checking efficiency.
- *Stack-based analysis*: In a stack trace model, event occuing during function execution are forgotten when the function returns. Function activations annotated with events; function return erases event.
- *Exceptions*: "Pre-effect" constructs allow us to add exceptions to the language with a trivial extension to the algorithm.

Example: Detecting security violations in Secure Socket Layer (SSL)

For a program sending and receiving data over an SSL socket, e.g. a web browser that supports `https`, the relevant events are opening and closing of sockets, and reading and writing of data packets.

An example event trace produced by a program run could be:

```
ssl_open("snork.cs.jhu.edu",socket_1); ssl_hs_begin(socket_1);
ssl_hs_success(socket_1); ssl_put(socket_1); ssl_get(socket_1);
ssl_open("moo.cs.uvm.edu",socket_2); ssl_hs_begin(socket_1);
ssl_put(socket_2); ssl_close(socket_1); ssl_close(socket_2)
```

Event traces can then be used to detect logical flaws or security violations. For SSL, sockets must first be opened, handshake begun, handshake success, and only then can data be get/put over the socket.

The above trace is illegal because data is put on `socket_2` before notification has been received that handshake was successful on that socket.

Results

- Proof establishing correctness of logical system
- Proof establishing correctness of transformation algorithms
- Proof establishing soundness of algorithm
- Prototype implementation of analysis
- Prototype implementation of transformation algorithms
- Extension of analysis to Java

Publications

- David Van Horn. *Algorithmic Trace Effect Analysis*. Masters Thesis, University of Vermont, 2005.
- Christian Skalka. Trace Effects and Object Orientation. In *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*, 2005.
- Christian Skalka, Scott Smith, and David Van Horn. A Type and Effect System for Flexible Abstract Interpretation of Java. In *Proceedings of the ACM Workshop on Abstract Interpretation of Object Oriented Languages*, Electronic Notes in Theoretical Computer Science, January 2005.
- Christian Skalka and Scott Smith. History Effects and Verification. In *Asian Programming Languages Symposium*, November 2004.