Types and Language-Based Security

Christian Skalka

The University of Vermont

Compile Time Analyses

Compile time analyses– aka *static* analyses– are methodologies for enhancing the behavior of programming languages

- Transparent components of programming language implementations
- Analyze source code during some phase of compilation (before execution)
- Well known examples:
  - Control-flow analysis
  - Dataflow analysis

Compile Time Analyses

Compile time analyses enhance the behavior of programming languages:

- Improve run-time efficiency (time and space) of programs
- Ensure that programs are semantically well-formed
- Prevent possibly dangerous program effects (e.g. overwriting critical regions of memory)

Type Systems

Type systems are a particular kind of compile-time analysis

- Symbolic (algebraic), rather than graphical, representation
- Rigorous formal foundations in mathematical logic
  - Theoretical principles pre-dating Eniac
- Numerous, efficient algorithms
- Textual representation provides human-readable specification of program meaning

The Success of Types

While untyped languages exist (e.g. Scheme, Perl), types are a successful approach, and are components of many modern languages

- Types exist (in rudimentary form) even in C!

```
struct stack { ... };

int *f(stack x) { ... }

f(150);   /* type error */
```

Continuing Advances

Ongoing research explores new applications of type analyses:

- Memory management via types
- Enforcement of sophisticated program invariants
- Detection of deadlock, race conditions
- Programming language-based security...

Programming Language-Based Security

In distributed computing environments, intrinsic issues of *trust* require mechanisms to ensure *security*

- Systems-level solutions:
  - Authentication protocols (e.g. Kerberos)
  - Secure communication channels (e.g. SSL)
  - Secure Operating Systems (e.g. Unix, EROS)

Programming Language-Based Security

Programming language-based security approaches provide security abstractions as primitive language features

- Allows programmers direct control over security mechanisms
- Provides greater flexibility, hence robustness, in definition of security policies at application level
- Programming language runtime setting provides uniquely fine grained *context* for access-control decisions

Example: Java Stack Inspection

The Java JDK1.2 implements an access-control security model using the *stack inspection* algorithm:

- Call stack maintains context of *who-called-whom*, a functional audit trail
- Authorization for sensitive resources based on analysis of stack contexts
- Resource access denied if untrusted parties detected in dynamic calling context

Example: Java Stack Inspection

Type systems have been developed for compile-time enforcement of stack inspection policies:

- Well-typed programs guaranteed to contain no unauthorized resource access
  - Christian Skalka and Scott Smith. *Static Enforcement of Security with Types*. (ICFP00)
  - François Pottier, Christian Skalka, and Scott Smith. *A Systematic Approach to Static Access Control*. (TOPLAS03)

Example: History-Based Access Control

A new *history-based* security mechanism has been proposed for the Microsoft CLR:

- A global history tracks *program events*
- Unlike stack-based access control, historical events never “erased”
- Fine-grained access control decisions based on analysis of program history

Example: History-Based Access Control

Current research: developing compile-time analyses for statically enforcing history-based access control policies

- Analysis based on combination of type and *model checking* techniques
- Approach yields an extremely general analysis:
  - Enforcement of history-based policies, stack-based policies, communication protocols, (and more?)
  - Arbitrary access control checks (provided they're expressible in process logics, e.g. modal μ-calculus)

Future Work

On the foundation of research so far:

- Study mixed static and run time analyses
- Refine analyses for use in object-oriented settings (especially in presence of class hierarchies)
- Develop type analyses for *distributed* language models (e.g. RPC), incorporating cryptographic techniques
- Recast analyses to other computational models (e.g. web services)

`http://www.cs.uvm.edu/~skalka`