Vulnerable Semantics of Solidity

Sukyoung Ryu
with PLRG@KAIST and friends

October 20, 2018
Fortress, JavaScript, and Solidity

Sukyoung Ryu
with PLRG@KAIST and friends

October 20, 2018
KAIST: SML and OCaml

Links to other SML resources

Please feel free to send additional URLs that should be on this list to smlnj-dev-list@mailman.cs.uchicago.edu

SML Programming Resources

- [Concurrent ML](#)
- [sml_tk](#), a library for using the TK graphical interface, now updated to version 3.0
- Martin Erwig's [Functional Graph Library](#)
- Yi and Ryu's [SML/NJ exception analyzer](#)
Harvard: Assembly, C, and Modula-3

Debugging Everywhere

The goal of the Debugging Everywhere project is to make debugging a cheap, ubiquitous service. We intend to begin by getting compilers to emit Active Debugging Information, which we expect will support multi-language, multi-platform debugging much more readily than older approaches like Dwarf or dbx "stabs."

```
ldb Fib (stopped) > t
  0 <_print:2> (Mips/mjr.c:25,2) void _print(char *s = (0x1000008c) " ")
* 1 <fib:51+0x24> (Fib.java:23,32)
    void fib(Fib this = {int buffer = 10,
      int[] a = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    }, int n = 10)
  2 <main:3+0x18> (Fib.java:5,23) void main(String[] argv = {})
  3 <main:end+0x1c> (mininub.c:7,9)
    int main(int argc = 2, char **argv = 0x7fff7b24,
      char **envp = 0x7fff7b34)
```
Sun Microsystems: Java and Scala

z: Vec := 0
r: Vec := x
p: Vec := r
ρ: Elt := r^T r
for j ← seq(1 : cgit_max) do
q = A p
α = ρ / p^T q
z := z + α p
r := r − α q
ρ_0 := ρ
ρ := ρ^T r
β = ρ / ρ_0
p := r + β p
end
(z, ∥x − Az∥)

sources / ProjectFortress / src / com / sun / fortress / parser

<table>
<thead>
<tr>
<th>Filename</th>
<th>Author</th>
<th>Revision</th>
<th>Modified</th>
<th>Log Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compilation.rats</td>
<td>chf</td>
<td>4575</td>
<td>over 4 years ago</td>
<td>Updated Copyright notices</td>
</tr>
<tr>
<td>Declaration.rats</td>
<td>chf</td>
<td>4575</td>
<td>over 4 years ago</td>
<td>Updated Copyright notices</td>
</tr>
<tr>
<td>DelimitedExpr.rats</td>
<td>sukyoungryu</td>
<td>4921</td>
<td>about 4 years ago</td>
<td>Revising</td>
</tr>
<tr>
<td>Expression.rats</td>
<td>Guy Steele</td>
<td>5051</td>
<td>almost 4 years ago</td>
<td>Completely redid assignment and ...</td>
</tr>
<tr>
<td>Fortress.rats</td>
<td>chf</td>
<td>4575</td>
<td>over 4 years ago</td>
<td>Updated Copyright notices</td>
</tr>
</tbody>
</table>
Sun Microsystems: Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!
- “Growing a Language”  [https://www.youtube.com/watch?v=_ahvzDzKdB0&t=10s](https://www.youtube.com/watch?v=_ahvzDzKdB0&t=10s)
  - Guy L. Steele Jr., keynote talk, OOPSLA 1998
  - Higher-Order and Symbolic Computation 12, 221-236 (1999)
Sun Microsystems: Fortress

“It’s an absolute piece of beauty”

https://www.youtube.com/watch?v=_ahvzDzKdB0
Language *Manipulation*: Fortress

- Specification
- Parsing
- Static checking
- Compilation / Interpretation
- Testing
- Analysis
- Verification
Fortress as a Language Designer

- Development from scratch
- Language evolves to experiment with new features
- Constant changes in spec., parsing, checking, ...
- Language development by 3 teams in tandem:
  - Specification team
  - Implementation team
  - Library team
Fortress as a Language Designer

- Specification team
  - Formal concrete grammar in EBNF
  - Informal description in prose
  - Examples & formal calculi
Fortress as a Language Designer

- Specification team
  - Formal concrete grammar in EBNF
  - Informal description in prose
  - Examples & formal calculi
- Library team
  - Adventurous library in unimplemented Fortress features
Fortress as a Language Designer

- Specification team
  - Formal concrete grammar in EBNF
  - Informal description in prose
  - Examples & formal calculi

- Library team
  - Adventurous library in unimplemented Fortress features

- Implementation team
  - Implementation extension
  - Regression tests & new feature tests
Language Manipulation: Fortress

- Specification: automatic extraction/test of examples
- Parsing: automatic generation of parsers & ASTs
- Static checking: parallel development
- Compilation / Interpretation: cross validation
- Testing
- Analysis
- Verification
  - FFMM (Featherweight Fortress with Multiple Dispatch and Multiple Inheritance): 3,000 LOC (Coq)
Fortress: Symmetric Multiple Dispatch

- Method overloading
  - Multiple method declarations of the same name
- Symmetric multiple dispatch
  - Selection of a method declaration at run time using all the arguments equally
- Overloading rules
  - Static rejection of ambiguous method declarations
  - No ambiguous nor undefined method call at run time!
Fortress: Symmetric Multiple Dispatch

collide(c: Car, cc: CampingCar): Int = _
collide(cc: CampingCar, c: Car): Int = _

cc1: CampingCar = CampingCar()
cc2: CampingCar = CampingCar()
collide(cc1, cc2)
Fortress: Symmetric Multiple Dispatch

collide(c: Car, cc: CampingCar): Int = _
collide(cc: CampingCar, c: Car): Int = _
collide(cc: CampingCar, c: CampingCar): Int = _
cc1: CampingCar = CampingCar()
cc2: CampingCar = CampingCar()
collide(cc1, cc2)
Fortress: Symmetric Multiple Dispatch

sort[P <: Car](x: List[P]): SortedList[P] = _

sort[P <: CampingTrailer](x: List[P]): SortedList[P] = _

1: List[CampingCar] = List(CampingCar())

sort(cc)
Fortress: Symmetric Multiple Dispatch

```
sort[P <: Car](x: List[P]): SortedList[P] = _
sort[P <: CampingTrailer](x: List[P]): SortedList[P] = _
sort[P <: CampingCar](x: List[P]): SortedList[P] = _
l: List[CampingCar] = List(CampingCar())
sort(cc)
```
Fortress: Symmetric Multiple Dispatch with Parametric Polymorphism

Type Checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance

Eric Allen  
Oracle Labs  
eric.allen@oracle.com

Justin Hilburn  
Oracle Labs  
justin.hilburn@oracle.com

Scott Kilpatrick  
University of Texas at Austin  
scottk@cs.utexas.edu

Victor Luchangco  
Oracle Labs  
victor.luchangco@oracle.com

Sukyoung Ryu  
KAIST  
sryu.cs@kaist.ac.kr

David Chase  
Oracle Labs  
david.r.chase@oracle.com

Guy L. Steele Jr.  
Oracle Labs  
guy.steele@oracle.com

- OOPSLA’11
- No variance, No dynamic dispatch algorithm
- No type soundness proof
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Polymorphic Symmetric Multiple Dispatch with Variance

GYUNGHEE PARK, Oracle Labs and KAIST
JAEMIN HONG, KAIST
GUY L. STEELE JR., Oracle Labs
SUKYOUNG RYU, KAIST

- POPL’19
- Yes variance, Yes dynamic dispatch algorithm
- Yes type soundness proof
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Program

\[ \Pi ::= \bar{\psi}, e \]

Class declaration

\[ \psi ::= \text{trait } T[\overline{V} \beta] ::= \{ \bar{t} \} \quad \bar{\mu} \quad \text{end} \mid \text{object } O[\overline{\beta}](\bar{x}:\overline{\tau}) ::= \{ \bar{t} \} \quad \bar{\mu} \quad \text{end} \]

Method definition

\[ \mu ::= m[\overline{k}](\bar{x}:\overline{\tau}):\tau = e \]

Class type parameter binding

\[ \beta ::= P <: \{ \tau \} \]

Method type parameter binding

\[ \kappa ::= \{ \tau \} ::= P <: \{ \tau \} \]

Variance type parameter binding

\[ \nu ::= + \mid - \mid = \]

Expression

\[ e ::= z \mid ((\bar{x}:\overline{\tau}):\tau \Rightarrow e) \mid e@e(\bar{e}) \mid O[\overline{\tau}](\bar{e}) \mid e.m(\bar{e}) \]

Bindable variable

\[ z ::= x \mid \text{self} \]

Type

\[ \tau ::= P \mid c \mid (\overline{\tau}) \mid (\tau \rightarrow \tau) \mid \text{Any} \]

Constructed type

\[ c ::= t \mid O[\overline{\tau}] \]

Trait type

\[ t ::= T[\overline{\tau}] \]
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Well-formed programs:

\[
\begin{align*}
\Pi & = \psi, e \\
\Delta & = (\psi) \\
\Delta & \vdash \psi \text{ ok} \\
\Delta & \vdash e : (\neg, g) \\
\end{align*}
\]  
T-PROGRAM

Well-formed trait and object declarations:

\[
\begin{align*}
\Delta' & = \Delta \cup \{ i < P < (\xi) \} \\
\Delta' & \vdash \xi \text{ ok} \\
\Delta' & \vdash i \text{ ok} \\
\end{align*}
\]  
D-TRAIT

\[
\begin{align*}
\Delta & \vdash \text{trait } T \{ P : (\xi) \} \text{ ok} \\
\Delta & \vdash i \text{ ok} \\
\end{align*}
\]  
D-OBJECT

Well-formed method declarations:

\[
\begin{align*}
\Delta & \vdash \{ (\xi) < Q < (\xi) \} \text{ ok} \\
\Delta' & = \Delta \cup \{ (\xi) < Q < (\xi) \} \\
\Delta' & \vdash \xi \text{ ok} \\
\Delta' & \vdash \xi \text{ ok} \\
\end{align*}
\]  
D-METHOD

Fig. 3. Well-formed programs, class declarations, and method declarations
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Well-formed programs:
\[
\Pi = \bar{\psi}, e \quad \text{distinct}(\text{name}(\bar{\psi})) \quad \Lambda = \{\bar{\psi}\} \quad \Delta \vdash \psi \quad \Delta \vdash e : (\bar{\psi}, g) \quad \text{[T-PROGRAM]}
\]
\[
\implies \Pi : g
\]

Well-formed trait and object declarations:
\[
\Lambda \vdash \psi \text{ ok}
\]
\[
\Delta' = \Delta \cup \{\{\} < P < \{\bar{\xi}\}\} \quad \text{distinct}(P) \quad \Delta' \vdash \psi \text{ ok} \quad \Delta' \vdash t \text{ ok} \quad \text{[D-Trait]}
\]
\[
\{\bar{\bar{x}}\} = \text{properAncestors}(\Lambda', T[P]\bar{\bar{x}}) \quad \Delta' ; \bar{\psi} \vdash \bar{\psi} \text{ and } \bar{\psi} \text{ ancestors ok} \quad \text{[Anc-Diff-Trait]}
\]
\[
\bar{\bar{x}} \neq \text{name}(\bar{\psi}) \quad \bar{\bar{x}}' \vdash \psi \text{ ok} \quad (\bar{\bar{d}}) = \text{allVisible}(\Delta', T[P]_{\bar{\bar{d}}}) \quad \text{[Anc-Same-Trait]}
\]
\[
\Delta' \vdash d_i \text{ not duplicate of } d_j \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \Delta' \vdash d_i \text{ meet } d_j \text{ wrt } (\bar{\bar{d}}) \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \Delta' \vdash d_i \text{ return type wrt } d_j \text{ ok} \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \text{[D-Object]}
\]
\[
\Delta \vdash \text{object } \{P < \{\bar{\xi}\}\} \text{ ok} \quad \Lambda' \vdash t \text{ end ok}
\]

Well-formed method declarations:
\[
\Lambda \vdash \{\{\} < Q < \{\bar{\xi}\}\} \quad \Delta' = \Delta \cup \{\{\} < Q < \{\bar{\xi}\}\} \quad \Delta' \vdash t \text{ ok} \quad \Delta' \vdash t \text{ ok} \quad \text{[D-Method]}
\]
\[
\Delta \vdash \Delta' ; \bar{\psi} \vdash \bar{\psi} \text{ and } \bar{\psi} \text{ ancestors ok} \quad \text{[Anc-Same-Trait]}
\]
\[
\bar{\bar{x}} \neq \text{name}(\bar{\psi}) \quad \bar{\bar{x}}' \vdash \psi \text{ ok} \quad (\bar{\bar{d}}) = \text{allVisible}(\Delta', T[P]_{\bar{\bar{d}}}) \quad \text{[Anc-Diff-Trait]}
\]
\[
\Delta' \vdash d_i \text{ not duplicate of } d_j \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \Delta' \vdash d_i \text{ meet } d_j \text{ wrt } (\bar{\bar{d}}) \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \Delta' \vdash d_i \text{ return type wrt } d_j \text{ ok} \quad (1 \leq i < j \leq \#(\bar{\bar{d}})) \quad \text{[D-Object]}
\]
\[
\Delta \vdash \text{object } \{P < \{\bar{\xi}\}\} \text{ ok} \quad \Lambda' \vdash t \text{ end ok}
\]

Well-formed ancestors:
\[
\Lambda ; \bar{\psi} \vdash \bar{\psi} \text{ and } \bar{\psi} \text{ ancestors ok} \quad \text{[Anc-Same-Trait]}
\]
\[
\{\{\} < P < \{\bar{\xi}\}\} \quad \Delta \vdash \{\{\} < P < \{\bar{\xi}\}\} \quad \text{[Anc-Diff-Trait]}
\]
\[
\{\{\} < Q < \{\bar{\xi}\}\} \quad \Delta \vdash \{\{\} < Q < \{\bar{\xi}\}\} \quad \text{[Anc-Same-Trait]}
\]
\[
\text{FV}(\{\bar{\xi}\}) \subseteq \text{parameters}(\Delta) \cup \{\bar{\bar{x}}\} \quad \text{FV}(\{\bar{\xi}\}) \subseteq \text{parameters}(\Delta) \quad \text{[Binding-Empty]}
\]
\[
\text{FV}(\{\bar{\xi}\}) \subseteq \text{parameters}(\Delta) \quad \{\{\} < P \leq \{\bar{\xi}\}\} \quad \{\{\} < P \leq \{\bar{\xi}\}\} \quad \text{[Binding-Step]}
\]

Fig. 3. Well-formed programs, class declarations, and method declarations

Fig. 4. Well-formed ancestors and type parameter bindings
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Well-formed programs:
\[ \Pi \vdash \phi \text{ and } \Delta = \{ \gamma \} \]

Well-formed trait and object declarations:
\[ \Delta' = \Delta \cup \{ \{ \} \subseteq \mathcal{P} \} \]
\( \Delta' = \Delta \cup \{ \{ \} 

\text{No Duplicates Rule:} \]
\[ \Delta \vdash d \text{ not duplicate of } d' \]
\[ \text{name}(d) \neq \text{name}(d') \]
\[ \text{Meet Rule:} \]
\[ \Delta \vdash d \text{ meet } d' \text{ wrt } \{ \delta \} \text{ ok} \]
\[ \text{Return Type Rule:} \]
\[ \Delta \vdash d \text{ return type wrt } d'' \text{ ok} \]
\[ \text{Return-Test} \]

Well-formed ancestors:
\[ \Delta ; J \vdash J \text{ and } J \text{ ancestors ok} \]
\[ T \neq T' \]
\[ \text{Meet-Third} \]

Fig. 5. Overloading rules for FCGV

Fig. 3. Well-formed programs, class declarations, and method declarations
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Well-formed programs:

\[ \Pi \vdash g \]

\[ \Pi = \bar{y}, e \quad \text{distinct(name}(\bar{y})) \quad \Delta = \{\bar{y}\} \quad \Delta \vdash \bar{y} \quad \Delta \vdash \bar{e} : \langle \cdots, g \rangle \quad \text{[T-PROGRAM]} \]

\[ \Pi \vdash g \]

Well-formed trait and object declarations:

\[ \Delta \vdash \text{tr} \quad \Pi \vdash P < \check{T} \]

\[ \Delta' = \Delta \cup \{\{i\} \subset P < \check{T}\} \quad \text{disn} \]

\[ \{J\} = \text{properAncestors}(\Delta', \Pi, P) \]

\[ \bar{J} \neq \text{name}(\bar{J}) \quad \Delta' ; \text{self}: T \quad \Pi \vdash \bar{J} \quad V \]

\[ \Delta' \vdash d_j \text{ not duplicate of } d_i \quad 1 \leq i, j \leq \#(d) \]

\[ \Delta' \vdash d_j \text{ return type } \text{wrt } d_j \text{ ok} \]

\[ \Delta' \vdash \text{trait } T \quad V \]

\[ \Delta' = \Delta \cup \{\{i\} \subset P < \check{T}\} \quad \text{disn} \]

\[ \{J\} = \text{properAncestors}(\Delta', \Pi, P) \]

\[ \bar{J} \neq \text{name}(\bar{J}) \quad \Delta' ; \text{self}: O \quad \Pi \vdash \bar{J} \quad V \quad \tau \]

\[ \Delta' \vdash d_j \text{ not duplicate of } d_i \quad 1 \leq i, j \leq \#(d) \]

\[ \Delta' \vdash d_j \text{ return type } \text{wrt } d_j \text{ ok} \]

Well-formed method declarations:

\[ \Delta ; (\Gamma, P) \quad \Pi \vdash \bar{P} \]

\[ \Delta \vdash \{\{i\} \subset Q < \check{T}\} \quad \text{ok} \quad \Delta' = \Delta \cup \{\{i\} \subset Q < \check{T}\} \quad \text{ok} \]

\[ \text{distinct}(\bar{Q}) \quad \Delta' ; \tau \quad \Delta' \quad \omega \quad \text{ok} \]

\[ \text{distinct}(\bar{P}, \bar{Q}) \quad (V = \tau) = \forall (\forall T . ((\bar{F} \rightarrow \tau) \neq T) \lor (\Delta + T \text{ variance } V)) \]

\[ \Delta ; \Gamma ; \bar{P} \quad m[\{i\} \subset Q < \check{T}\} \quad \text{ok} \]

Well-formed ancestors:

\[ \Delta ; J \vdash J \quad \text{and } J \text{ ancestors } \quad \text{ok} \]

\[ \text{[D-METHOD]} \]

Existential inner subtyping:

\[ \Delta \vdash \Xi \subseteq \Xi \quad \text{using } \sigma \]

\[ \Delta' = \Delta \cup \{\{\bar{x}\} \subset P < \check{\eta}\} \quad \{\bar{P}\} \quad \forall (P \chi)^{\bar{x}} \quad \bar{\eta}, a' = \emptyset \quad \sigma = \gamma[\bar{P}] \quad \gamma \neq \text{Bottom} \]

\[ \Delta' + a = a' \quad \text{[E-Sub]} \]

Universal inner subtyping:

\[ \Delta' = \Delta \cup \{\{\bar{x}\} \subset Q < \check{\eta}\} \quad \{\bar{P}\} \quad \forall (P \chi)^{\bar{x}} \quad \bar{\eta}, a = \emptyset \quad \sigma = \gamma[\bar{P}] \quad \gamma \neq \text{Bottom} \]

\[ \Delta' + a = a' \quad \text{[U-Sub]} \]

Existential reduction:

\[ \Delta + \Xi \rightarrow \Xi \quad \text{using } \sigma \quad \text{otherwise} \]

\[ \Delta + a = a' \quad \text{Bottom} \quad \Delta + \text{unify}(C \land C') = (\sigma, C') \quad \text{toBound(C')} = (\bar{K}) \]

\[ \Delta + \exists [\bar{K}] \alpha \rightarrow \exists [\bar{K}] \sigma \alpha \quad \text{using } \sigma \]

\[ \Delta + \exists [\bar{K}] \alpha \rightarrow \exists [\bar{K}] \alpha \quad \text{using } \sigma \]

\[ \Delta + \forall (\forall \bar{K} \alpha \rightarrow \sigma \omega) \rightarrow \forall (\forall \bar{K} \alpha \rightarrow \sigma \omega) \quad \text{using } \sigma \]

\[ \Delta + \Xi \subseteq \Xi' \quad \text{using } \sigma \]

\[ \Delta + \Xi \subseteq \Xi' \quad \text{using } \sigma \]

\[ \Delta + \Xi \subseteq \Xi' \quad \text{using } \sigma \]

\[ \Delta + \Xi \subseteq \Xi' \quad \text{using } \sigma \]

Fig. 5. Overloading rules for FC

Fig. 6. Subtype relations of quantified types
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

Well-formed programs: \[
\Pi = \bar{\psi}, e \quad \text{distinct}(\text{name}(\bar{\psi})) \quad \Delta = \{\bar{\psi}\} \quad \Delta + \psi \text{ ok} \quad \Delta \cdot \epsilon : (\neg \psi) \quad [\text{T-PROGRAM}]
\]
\[
\Pi = g \quad \Delta \cdot \psi \text{ ok} \quad \Delta \cdot \epsilon : (\neg g) \quad [\text{T-PROGRAM}]
\]

Well-formed trait and object declarations: \[
\Delta \cdot \text{trait } T \{ Y \leftarrow P < \}
\]
\[
(\bar{J}) = \text{properAncestors}(\Delta \cdot T[P], \bar{P}) \quad \bar{J} \neq \text{name}(\bar{J}) \quad \Delta^\prime \cdot \text{self}\cdot T[P], \bar{P}; \bar{V}; \bar{T} = P \quad \Delta \cdot \bar{J} \text{ return type wrt } d_j \text{ ok}
\]
\[
\Delta^\prime \cdot d_j \text{ not duplicate of } d_j \quad [\text{NO-DUP-LESS}]
\]
\[
\Delta \cdot \text{trait } T \{ Y \leftarrow P < \}
\]
\[
\Delta^\prime = \Delta \cup \{J \lessdot P < \}
\]
\[
\Delta \cdot \text{trait } T \{ Y \leftarrow P < \}
\]
\[
\Delta^\prime \cdot \text{self}\cdot T[P], \bar{P}; \bar{V}; \bar{T} = P \quad \Delta \cdot \bar{J} \text{ return type wrt } d_j \text{ ok}
\]
\[
\Delta^\prime \cdot d_j \text{ not duplicate of } d_j \quad [\text{NO-DUP-LESS}]
\]

Well-formed ancestors: \[
\Delta \cdot J \triangleright J \text{ and } J \text{ ancestors ok}
\]
\[
\Delta \cdot \psi \text{ ok} \quad \Delta \cdot \psi \text{ ok} \quad \Delta \cdot \epsilon : (\neg \psi) \quad [\text{T-PROGRAM}]
\]

Existential inner subtyping: \[
\Delta \cdot \Xi \lessdot \Xi \text{ using } \sigma
\]
\[
\Delta^\prime = \Delta \cup \{[\bar{\chi}] \lessdot P < (\bar{\eta})\} \quad (\bar{P}) \cap F(V(\bar{\chi}, \bar{\eta}, \bar{\alpha}_\eta) = 0 \quad \sigma = \lfloor \bar{\eta} \rfloor \quad \bar{y} \neq \text{Bottom}
\]
\[
\Delta^\prime \cdot \sigma \text{ on } (\bar{Q}) \text{ obeys } (\lfloor \bar{\chi} \rfloor) \text{ and } (\lfloor \bar{\eta} \rfloor) \quad \Delta^\prime \cdot \alpha < \alpha' \quad [\text{E-Sub}]
\]

Universal inner subtyping: \[
\Delta \cdot \Xi \lessdot \Xi \text{ using } \sigma
\]
\[
\Delta^\prime = \Delta \cup \{[\bar{\chi}] \lessdot Q < (\bar{\eta})\} \quad (\bar{Q}) \cap F(V(\bar{\chi}, \bar{\eta}, \bar{\alpha}_\eta) = 0 \quad \sigma = \lfloor \bar{\eta} \rfloor \quad \bar{y} \neq \text{Bottom}
\]
\[
\Delta^\prime \cdot \sigma \text{ on } (\bar{P}) \text{ obeys } (\lfloor \bar{\chi} \rfloor) \text{ and } (\lfloor \bar{\eta} \rfloor) \quad \Delta^\prime \cdot \alpha < \alpha' \quad [\text{U-Sub}]
\]

Many more rules …
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

PROVED

Next: Coq Mechanization
Fortress: Symmetric Multiple Dispatch with Polymorphism and Variance

### Return Type Rule

\[ \Delta \vdash d \text{ return type wrt } d' \text{ ok} \]

<table>
<thead>
<tr>
<th>No Duplicates Rule:</th>
<th>Δ ⊢ d not duplicate of d''</th>
</tr>
</thead>
<tbody>
<tr>
<td>name(d) ≠ name(d'')</td>
<td>[No-DUP-TRIV]</td>
</tr>
<tr>
<td>Δ ⊢ d not duplicate of d'</td>
<td></td>
</tr>
<tr>
<td>¬(Δ ⊢ dom(d) ⊆ dom(d'))</td>
<td>[No-DUP-NOT-LESS]</td>
</tr>
<tr>
<td>Δ ⊢ d not duplicate of d'</td>
<td></td>
</tr>
<tr>
<td>¬(Δ ⊢ dom(d') ⊆ dom(d))</td>
<td>[No-DUP-NOT-GTR]</td>
</tr>
<tr>
<td>Δ ⊢ d not duplicate of d''</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meet Rule:</th>
<th>Δ ⊢ d meet d' wrt {d} ok</th>
</tr>
</thead>
<tbody>
<tr>
<td>name(d) ≠ name(d')</td>
<td>[Meet-Triv]</td>
</tr>
<tr>
<td>Δ ⊢ d meet d' wrt ok</td>
<td></td>
</tr>
<tr>
<td>Δ ⊢ (dom(d) ∩ dom(d')) ⊆ \exists\text{Bottom}</td>
<td>[Meet-EXCL]</td>
</tr>
<tr>
<td>Δ ⊢ d meet d' wrt ok</td>
<td></td>
</tr>
<tr>
<td>Δ ⊢ dom(d) ⊆ dom(d')</td>
<td>[Meet-LESS]</td>
</tr>
<tr>
<td>Δ ⊢ d meet d' wrt ok</td>
<td></td>
</tr>
<tr>
<td>Δ ⊢ dom(d') ⊆ dom(d)</td>
<td>[Meet-GTR]</td>
</tr>
<tr>
<td>Δ ⊢ d meet d' wrt ok</td>
<td></td>
</tr>
</tbody>
</table>

\[ d'''' \in \{d''\} \quad \text{name}(d) = \text{name}(d') = \text{name}(d''') \quad \Delta \vdash \text{dom}(d''') \equiv (\text{dom}(d) \cap \text{dom}(d')) \]

<table>
<thead>
<tr>
<th>Return Type Rule:</th>
<th>Δ ⊢ d return type wrt d ok</th>
</tr>
</thead>
<tbody>
<tr>
<td>name(d) ≠ name(d')</td>
<td>[RETURN-TRIV]</td>
</tr>
<tr>
<td>Δ ⊢ d return type wrt d' ok</td>
<td></td>
</tr>
<tr>
<td>arrow(d) = ∀[\overline{K}](\alpha \to \rho) \quad \kappa = {\overline{\alpha}} &lt; P &lt; {\overline{\eta}}</td>
<td>[RETURN-NOT-LESS]</td>
</tr>
<tr>
<td>\kappa' = {\overline{\alpha'}} &lt; Q &lt; {\overline{\eta'}}</td>
<td>[RETURN-TEST]</td>
</tr>
<tr>
<td>δ\text{distinct}(\overline{P}, \overline{Q})</td>
<td>[RETURN-TEST]</td>
</tr>
<tr>
<td>Δ ⊢ dom(d) ⊆ dom(d')</td>
<td>[RETURN-Test]</td>
</tr>
</tbody>
</table>

Δ ⊢ d return type wrt \(d'\) ok

Fig. 5. Overloading rules for FGV
Language Manipulation: JavaScript

- Specification: ECMAScript
- Parsing: automatic generation of parsers & ASTs
- Static checking: parallel development
- Compilation / Interpretation: cross validation
- Testing
- Analysis: SAFE, TAJS, WALA
- Verification
Analyzing JS


Seonghoon Kang and Sukyoung Ryu. **Formal Specification of a JavaScript Module System** (OOPSLA’12)

Changhee Park, Hongki Lee, and Sukyoung Ryu. **All about the “with” Statement in JavaScript: Removing “with” Statements in JavaScript Applications** (DLS’13)

Analyzing JS

Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the “with” Statement in JavaScript: Removing “with” Statements in JavaScript Applications (DLS’13)

Analyzing JS Web Apps

Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. **Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling** (ASE’15)

Changhee Park and Sukyoung Ryu. **Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity** (ECOOP’15)
Analyzing JS Web Apps in the Wild

SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. **SAFEwAPI: Web API Misuse Detector for Web Applications** (FSE’14)

Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. **Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications** (ASE’15)
Analyzing JS Web Apps in the Wild Partially

Joonyoung Park, Inho Lim, and Sukyoung Ryu. **Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild** (ICSE-SEIP’16)

Joonyoung Park, Kwangwon Sun, and Sukyoung Ryu. **EventHandler-based Analysis Framework for Web Apps using Dynamically Collected States** (FASE’18)
Analyzing JS Web Apps in the Wild Partially

JavaScript Bug Detection

```javascript
function invertMatrix(self) {
    var tx = self[3];
    var ty = self[7];
    var tz = self[11];
    for (h = 0; h < 3; h++)
        for (v = 0; v < 3; v++)
            for (i = 0; i < 16; i++)
                self[i] = temp[i];
    self[3] = tx * self[0];
    self[7] = ty * self[4];
    return self;
}
```

Web Application Bug Detection (I)

```javascript
loading: function(i) {
    // alert("img src=image/loading_"+i+".png");
    var tmdiv8 = document.getElementById("waitIcon");
    i = i == null ? 1 : i;
    // set timeout is 30s
    this.totalCnt++;
    if (this.totalCnt > 200) {
        this.endLoading();
        waited.className = "";
        this.showNoResultFound(tmdiv8);
        return;
    }
}
```

WiKIPEDIA

Wikipedia

English

The Free Encyclopedia

4,034,000+ articles

Español

La Enciclopedia Libre

1,047,000+ articles

 другие

Wikipedia de la Enciclopedia libre

1,033,000+ articles

Italiano

L'Enciclopedia Libera

1,450,000+ articles

Accept

Rabbit.js 753
Bug Detection in JS Web Apps with SAFE

SAFE 2.0

Pluggability  Extensibility  Debuggability

Prototype Implementation in SAFE

- Annotate untrusted (user) inputs
- Propagate tainted values
- Run SAFE with extended abstract domain
- Add taint checking as post-processing phase
- Report taint at sinks

https://github.com/sukyoung/safe

Procedure

1. Enqueue Projects
2. Take a job
3. Already processed?
4. Run SAFE
5. Set “processed”

Revisiting Recency Abstraction for JavaScript
Towards an Intuitive, Compositional, and Efficient Heap Abstraction
Singleton Abstraction

Jihyeok Park  Xavier Rival  Sukyoung Ryu
KAIST  DIENS, ÉNS, CNRS, PSL Research University and INRIA  KAIST
Now, Solidity!

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

The DAO, the distributed autonomous organization that had collected over $150m worth of the cryptocurrency ether, has reportedly been hacked, sparking a broad market sell-off.

A leaderless organization comprised of a series of smart contracts written on the ethereum codebase, The DAO has lost 3.6m ether, which is currently sitting in a separate wallet after being split off into a separate grouping dubbed a "child DAO."

http://www.dailymail.co.uk/sciencetech/article-5062543/200-MILLION-virtual-currency-Ether-lost.html

Code bug freezes $150m of Ethereum crypto-cash

Hackers Have Walked Off With About 14% of Big Digital Currencies

By Olga Kharif
January 18, 2018, 7:19 PM GMT+5:30

Cybercriminals compromise Bitcoin, Ether supply, blockchains
Crypto-crazed users adopt technology without weighing risks

Ethereum Hacks

The press is reporting a $32M theft of the cryptocurrency Ethereum. Like all such thefts, they're not a result of a cryptographic failure in the currencies, but instead a software vulnerability in the software surrounding the currency -- in this case, digital wallets.

This is the second Ethereum hack this week. The first tricked people in sending their Ethereum to another address.

This is my concern about digital cash. The cryptography can be bulletproof, but the computer security will always be an issue.

Tags: cryptocurrency, cryptography, hacking, theft, vulnerabilities

Posted on July 20, 2017 at 9:12 AM • 46 Comments
**Smart Contract Vulnerabilities: Research**

---

**ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER  BYZANTIUM VERSION d1ca9d0 - 2018-03-0827**

---

**0s: Stop and Arithmetic Operations**

All arithmetic is modulo $2^{256}$ unless otherwise noted. The zero-th power of zero $0^0$ is defined to be one.

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>$\delta$</th>
<th>$\alpha$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>STOP</td>
<td>0</td>
<td>0</td>
<td>Halts execution.</td>
</tr>
<tr>
<td>0x01</td>
<td>ADD</td>
<td>2</td>
<td>1</td>
<td>Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$</td>
</tr>
<tr>
<td>0x02</td>
<td>MUL</td>
<td>2</td>
<td>1</td>
<td>Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$</td>
</tr>
<tr>
<td>0x03</td>
<td>SUB</td>
<td>2</td>
<td>1</td>
<td>Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$</td>
</tr>
</tbody>
</table>
| 0x04  | DIV      | 2        | 1        | Integer division operation.  

\[
\mu'_s[0] = \begin{cases} 
0 & \text{if } \mu_s[1] = 0 \\
\lfloor \mu_s[0] / \mu_s[1] \rfloor & \text{otherwise}
\end{cases}
\]

| 0x05  | SDIV     | 2        | 1        | Signed integer division operation (truncated).  

\[
\mu'_s[0] = \begin{cases} 
0 & \text{if } \mu_s[1] = 0 \\
-2^{255} & \text{if } \mu_s[0] = -2^{255} \land \mu_s[1] = -1 \\
\text{sgn}(\mu_s[0] / \mu_s[1]) \lfloor |\mu_s[0] / \mu_s[1]| \rfloor & \text{otherwise}
\end{cases}
\]

Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when $-2^{255}$ is negated.

---

Smart Contract Vulnerabilities: Research

- Formal Verification of Smart Contracts, PLAS 2016
  - A small subset of the Solidity programming language
  - A tiny language and no automatic verification
Smart Contract Vulnerabilities: Research

- Making Smart Contracts Smarter, CCS 2016
- Oyente: Symbolic execution of EVM bytecode
- Not sound nor complete

```
6060604052123
123123528....
```

# Diagram

- **ByteCode**
- **Ethereum State**
- **CFG BUILDER**
- **EXPLORER**
- **CORE ANALYSIS**
- **VALIDATOR**
- **Z3 Bit-Vector Solver**
- **Visualizer**
# Smart Contract Vulnerabilities: Research

- A Survey of Attacks on Ethereum Smart Contracts, POST 2017

<table>
<thead>
<tr>
<th>Level</th>
<th>Cause of vulnerability</th>
<th>Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solidity</td>
<td>Call to the unknown</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>Gasless send</td>
<td>4.2</td>
</tr>
<tr>
<td></td>
<td>Exception disorders</td>
<td>4.2, 4.5</td>
</tr>
<tr>
<td></td>
<td>Type casts</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Reentrancy</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>Keeping secrets</td>
<td>4.3</td>
</tr>
<tr>
<td>EVM</td>
<td>Immutable bugs</td>
<td>4.4, 4.5</td>
</tr>
<tr>
<td></td>
<td>Ether lost in transfer</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Stack size limit</td>
<td>4.5</td>
</tr>
<tr>
<td>Blockchain</td>
<td>Unpredictable state</td>
<td>4.5, 4.6</td>
</tr>
<tr>
<td></td>
<td>Generating randomness</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Time constraints</td>
<td>4.5</td>
</tr>
</tbody>
</table>

**Table 1.** Taxonomy of vulnerabilities in Ethereum smart contracts.
Smart Contract Vulnerabilities: Research

- Zeus: Analyzing Safety of Smart Contracts, NDSS 2018
- Verification using an LLVM model checker after compiling Solidity code to LLVM bitcode
Smart Contract Vulnerabilities: Research

- A Semantic Framework for the Security Analysis of Ethereum Smart Contracts, POST 2018

**Formal guarantees**

- We prove our abstraction to be sound:
  - “Every concrete execution can be mimicked by derivations in the abstract semantics”

\[ \Gamma \vdash s_c :: S \rightarrow^* S' + +S \]

- \( \beta_s(s, id, 0) \cup \beta_c(C_{call}, id) \vdash \beta_s(S', C_{call}, id, 0) \)

- artificial id of contract \( c \) (in accordance with \( C_{call} \))
- set of known contracts

- encoding of execution state as predicate instances
- encoding of program logics of known contracts as horn clauses
- encoding of call stack as predicate instances
Smart Contract Vulnerabilities: Research

USENIX Security 2018

Track 1

Smart Contracts

Session Chair: Suman Jana, Columbia University

teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts
Johannes Krupp and Christian Rossow, CISPA, Saarland University, Saarland Informatics Campus

Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts
Lorenz Breidenbach, Cornell Tech, IC3, ETH Zurich; Philip Daian, Cornell Tech, IC3; Florian Tramer, Stanford; Ari Juels, Cornell Tech, IC3, Jacobs Institute

Arbitrum: Scalable, private smart contracts
Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten, Princeton University

Erays: Reverse Engineering Ethereum's Opaque Smart Contracts
Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey, University of Illinois, Urbana-Champaign

Fortress, JavaScript, and Solidity
Smart Contract Vulnerabilities: Research

ACM CCS 2018

**Smart Contracts** (203 AB)

*Session Chair: Yan Chen*

**SECURIFY: Practical Security Analysis of Smart Contracts**  
Petar Tsankov (ETH Zurich), Andrei Marian Dan (ETH Zurich), Dana Drachsler Cohen (ETH Zurich), Arthur Gervais (Imperial College London), Florian Buzenzi (ETH Zurich), Martin Vechev (ETH Zurich)

**BitML: a calculus for Bitcoin smart contracts**  
Massimo Bartoletti (University of Cagliari), Roberto Zunino (University of Trento)
Vulnerable Semantics of Solidity: Scope

Scoping and Declarations

A variable declared anywhere within a function will be in scope for the *entire function*, regardless of where it is declared (this will change soon, see below). This happens because Solidity inherits its scoping rules from JavaScript. This is in contrast to many languages where variables are only scoped where they are declared until the end of the semantic block. As a result, the following code is illegal and cause the compiler to throw an error, 

```
Identifier already declared:
```
Vulnerable Semantics of Solidity: Scope

Scoping and Declarations

A variable declared anywhere within a function will be in scope for the entire function, regardless of where it is declared (this will change soon, see below). This happens because Solidity inherits its scoping rules from JavaScript. This is in contrast to many languages where variables are only scoped where they are declared until the end of the semantic block. As a result, the following code is illegal and cause the compiler to throw an error, **Identifier already declared**: 

```javascript
function myFunction()
{
    let x = 10;
    let y = 20;
    let x; // Error: Identifier already declared
}
```
Vulnerable Semantics of Solidity: Scope

Scoping and Declarations

A variable declared anywhere within a function will be in scope for the entire function, regardless of where it is declared (this will change soon, see below). This happens because Solidity inherits its scoping rules from JavaScript. This is in contrast to many languages where variables are only scoped where they are declared until the end of the semantic block. As a result, the following code is illegal and cause the compiler to throw an error, **Identifier already declared**: 

Scoping starting from Version 0.5.0

Starting from version 0.5.0, Solidity will change to the more widespread scoping rules of C99 (and many other languages): Variables are visible from the point right after their declaration until the end of a `{}`-block. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.
Vulnerable Semantics of Solidity: MM

Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

⚠️ Note

Return parameters are not taken into account for overload resolution.

```solidity
pragma solidity ^0.4.16;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```
Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

Note

Return parameters are not taken into account for overload resolution.

Return Type Rule:

\[
\Delta \vdash d \text{ return type wrt } d' \text{ ok} \\
\Delta \vdash d \text{ return type wrt } d' \text{ ok} \\
\Delta \vdash d \text{ return type wrt } d' \text{ ok}
\]
Vulnerable Semantics of Solidity: MM

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses “C3 Linearization” to force a specific order in the DAG of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important: You have to list the direct base contracts in the order from “most base-like” to “most derived”. Note that this order is different from the one used in Python. In the following code, Solidity will give the error “Linearization of inheritance graph impossible”.

```solidity
pragma solidity ^0.4.0;

contract X {}
contract A is X {}
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.
Vulnerable Semantics of Solidity: MM

pragma solidity ^0.4.22;

contract owned { ... }

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* cleanup 1 */ mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public { /* cleanup 2 */ mortal.kill(); }
}

contract Final is Base1, Base2 { ... }
Vulnerable Semantics of Solidity: MM

pragma solidity ^0.4.22;

contract owned { ... }

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* cleanup 1 */ super.kill(); }  
}

contract Base2 is mortal {
    function kill() public { /* cleanup 2 */ super.kill(); }  
}

contract Final is Base1, Base2 { ... }
Vulnerable Semantics of Solidity: MM

Crowdsale

withinPeriod && nonZeroPurchase

CappedCrowdsale

super.validPurchase() && withinCap

WhitelistedCrowdsale

super.validPurchase() || (whitelist[msg.sender] && !hasEnded())

MDTCrowdsale

((withinPeriod && nonZeroPurchase) && withinCap) || (whitelist[msg.sender] && !hasEnded())

Vulnerable Semantics of Solidity: MM

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses "C3 Linearization" to force a specific order in the DAG of base classes. This results in the desirable property of monotonicity but disallows some

(The name "C3" is not an initialism.) It was first published at the 1996 OOPSLA conference, in a paper entitled "A Monotonic Superclass Linearization for Dylan".[1] It was adapted to the Open Dylan implementation in January 2012[2] following an enhancement proposal.[3] It has been chosen as the default algorithm for method resolution in Python 2.3 (and newer),[4][5] Perl 6,[6] Parrot,[7], Solidity, and PGF/TikZ's Object-Oriented Programming module.[8] It is also available as an alternative, non-default MRO in the core of Perl 5 starting with version 5.10.0.[9] An extension implementation for earlier versions of Perl 5 named Class::C3 exists on CPAN.[10]
Smart Contract Vulnerabilities: Research

- Which platform?
  - Ethereum, Michelson/Liquidity, Zilliqa/Scilla, …

- Which language?
  - Source-level: Solidity, LLL, Vyper, …
  - Bytecode

- What problems?
  - Bugs: type-related, resource-related, …
  - Vulnerabilities: security, privacy, …
PLRG@KAIST for the Wild with Theory

- PL to save the world from bugs in real-world applications
- PL with proofs
- PL for Software Engineering
- PL for Security

열심히, 졸겁게, 자발적으로
PLRG@KAIST for the Wild with Theory

Come visit PLRG@KAIST!