Symbolic Execution Game Semantics

Yu-Yang Lin
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Nikos Tzevelekos
Queen Mary
University of London

Objective: Checking Higher-Order Libraries

Finding bugs in stateful higher-order libraries

Higher-Order: open with free variables of arbitrary order

Libraries: collection of methods

Stateful: globally accessible higher-order references

Bugs: specified by reachability of assertion violations
(i.e. we are interested in safety properties)

First-Order vs Higher-Order Bugs

```
#first-order
def f(x):
   if x >= 0:
      assert(false)
```

```
#higher-order
def f(g,x):
   if g(x) >= 0:
      assert(false)
```

First-Order Errors:

- Counter-example: value
- All code is available
- All contexts are known

Higher-Order Errors:

- Counter-example: context
- Not all code is available
- Need to guess context

Why open code matters

```
#in The DAO
def withdraw(user,m):
   if funds[user] >= m:
      user.send(m)
      funds[user] -= m
      assert(funds[user]>=0)
```

The DAO: *Decentralized Autonomous Organization* (DAO) in the Ethereum platform; somewhat like a **bank**

DAOs are a set of *smart contracts* (scripts) in the blockchain

The DAO bug analogous to the Python code above

Why open code matters

Library:

```
#in The DAO
def withdraw(user,m):
   if funds[user] >= m:
      user.send(m)
      funds[user] -= m
      assert(funds[user]>=0)
```

Environment:

```
#in the attacker
def send(m):
   wallet.add(m)
   withdraw(self,1)
```

Recursive call drains The DAO of **over 3.6 million ether**

Price of ether drops from **$20 to $13**

Ethereum network **hard-forked** to undo the "attack"

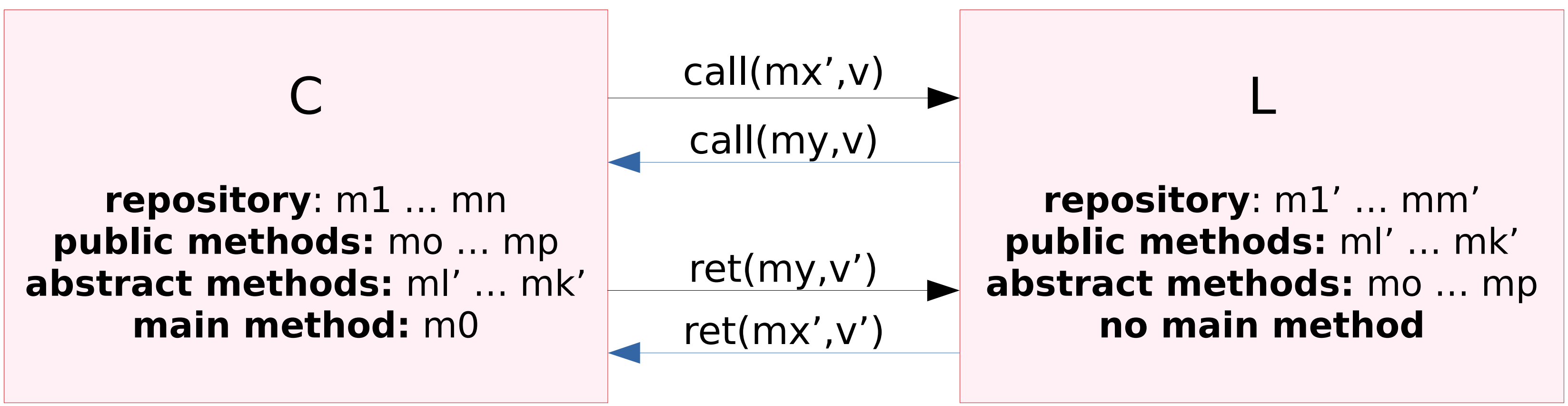
Some members reject the hard-fork and continue on the original blockchain, now called **Ethereum Classic**

In literature, this is called a **reentrancy attack**

Our Approach

Combine *Symbolic Execution* with *Game Semantics* to model check open code with free variables of arbitrary order

Use the *Library*(*L*)-*Client*(*C*) paradigm



Goal 1: check libraries independent of a client

Goal 2: compose the semantics of a library and a client to obtain the semantics of the whole program

Higher-Order Libraries

Libraries: sequence of *method declarations*

- may depend on *abstract* methods provided by the environment

$$
\begin{aligned}
\mathit{Libraries} \quad L &::= \; B \mid \texttt{abstract}\; m; L \\
\mathit{Blocks} \quad B &::= \; \varepsilon \mid \texttt{public}\; m = \lambda x.M; B \mid m = \lambda x.M; B \\
&\quad\;\; \mid m = \lambda x.M; B \mid \texttt{global}\; r := i; B
\end{aligned}
$$

Higher-Order Terms

We examine a higher-order language with references

- higher-order methods
- higher-order lambda abstractions
- higher-order (global) store

$$
\begin{aligned}
M ::= & \texttt{assert}(M) \mid x \mid m \mid i \mid () \mid r := M \mid !r \\
& \mid \lambda x.M \mid M M \mid M \oplus M \mid \langle M, M \rangle \mid \pi_1 M \mid \pi_2 M \\
& \mid \texttt{if } M \texttt{ then } M \texttt{ else } M \mid \texttt{let } x = M \texttt{ in } M \\
& \mid \texttt{letrec } x = \lambda x.M \texttt{ in } M \mid (\!| M |\!)
\end{aligned}
$$

$$
\frac{M : \texttt{int}}{\texttt{assert}(M) : \texttt{unit}} \qquad \frac{}{() : \texttt{unit}} \qquad \frac{}{i : \texttt{int}} \qquad \frac{x \in \texttt{Vars}_\theta}{x : \theta} \qquad \frac{m \in \texttt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'}
$$

$$
\frac{M : \texttt{int} \qquad M_0, M_1 : \theta}{\texttt{if } M \texttt{ then } M_1 \texttt{ else } M_0 : \theta} \qquad \frac{r \in \texttt{Refs}_\theta}{!r : \theta} \qquad \frac{r \in \texttt{Refs}_\theta \qquad M : \theta}{r := M : \texttt{unit}} \qquad \frac{M' : \theta \to \theta' \qquad M : \theta}{M' M : \theta'}
$$

Operational Semantics

Configurations of the form (M,R,S,k)

- **Counter** k for nested method application

M: term to evaluate
R: method repository
S: store
k: call counter

Example transition rules:

$$(E[\texttt{assert}\ (i)], R, S, k) \to (E[()], R, S, k)$$
$$(E[!r], R, S, k) \to (E[S(r)], R, S, k)$$

$$(E[\texttt{if}\ 0\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_0], R, S, k)$$
$$(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_1], R, S, k) \quad (i \neq 0)$$

$$(E[mv], R, S, k) \to (E[(\!|M\{v/x\}|\!)], R, S, k+1) \text{where } R(m) = \lambda x.M$$
$$(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k-1)$$

$$E ::= \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid \pi_j E \mid mE$$
$$\mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid (\!|E|\!) \mid vE$$

Bounded Games

We present a *trace semantics* for *open terms*

Traces: sequences of moves of the form $m(v)?$ (question) or $m(v)!$ (answer)

Call counters for both players:

- Proponent (*P*): Library to check; call depth with *k* as before
- Opponent (*O*): Environment for library; l counts *chattering*, i.e. number of calls O plays at the same *level*

$$(M, R, S, \mathcal{E}, \mathcal{P}, \mathcal{A}, k)_p$$

Proponent Configuration

$$(l, R, S, \mathcal{E}, \mathcal{P}, \mathcal{A}, k)_o$$

Opponent Configuration

M,R,S,k as before, $\mathcal{E}$ is a call stack,
$\mathcal{P}$ and $\mathcal{A}$ are the method names of P and O

Back to The DAO Attack

Consider the following library:

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

We start from an opponent configuration and bound to k,l=2:

$$C_0 = (0, R, \{funds \mapsto 50\}, \varepsilon, \{wdraw\}, \{send\}, 0)_o$$

where R(wdraw) = λm. … and dom(R) = {wdraw}

Back to The DAO Attack

C_0

$$\xrightarrow{wdraw(42)?} (wdraw(42), R, S, (wdraw, 1) :: \varepsilon, -, 0)_p$$

$$\rightarrow^* (E[send(42)], R, S, (wdraw, 1) :: \varepsilon, -, 1)_p$$

$$\xrightarrow{send(42)?} (1, R, S, (send, E) :: \ldots, -, 1)_o$$

$$\xrightarrow{wdraw(42)?} (wdraw(42), R, S, (wdraw, 0) :: \ldots, -, 1)_p$$

$$\rightarrow^* (E[send(42)], R, S, (wdraw, 2) :: \ldots, -, 2)_p$$

$$\xrightarrow{send(42)?} (2, R, S, (send, E) :: \ldots, -, 2)_o$$

$$\xrightarrow{send()!} (E'[()], R, S, (wdraw, 2) :: \ldots, -, 2)_p$$

$$\rightarrow^* ((), R, S[funds \mapsto 8], (wdraw, 2) :: \ldots, -, 2)_p$$

$$\xrightarrow{wdraw()!} (1, R, S[funds \mapsto 8], (send, E) :: \ldots, -, 1)_o$$

$$\xrightarrow{send()!} (E[()], R, S[funds \mapsto 8], (wdraw, 1) :: \varepsilon, -, 1)_p$$

$$\rightarrow^* (E[assert(-34 \geq 0)], R, S[funds \mapsto -34], (wdraw, 1) :: \varepsilon, -, 1)_p$$

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

$E = \bullet; funds := !funds - 42;$
$\quad$ assert$(!funds \geq 0)$

```
public send;
abstract withdraw;

call_counter := 0;
send = λm.
 if   !call_counter==0
 then withdraw(42); skip;
 else skip

main = λ().withdraw(42)
```

Soundness and Completeness of Games

- Linking a library L to a client is written *L*;*C*
- We call a client *good* if it contains no assertions

Theorem: For any library L, the following are equivalent:

1) There exists a good client C such that *L*;*C* fails.
2) There exists a trace in $[\![L]\!]$ reaching an assertion violation.

Proof:

- **Compositionality:** $[\![L;C]\!]$ can be decomposed into $[\![L]\!]$ and $[\![C]\!]$
- **Definability:** there exists a matching client for every trace in $[\![L]\!]$

$(1) \implies (2)$: if $[\![L;C]\!]$ fails, then by decomposing we have a trace in $[\![L]\!]$ that fails

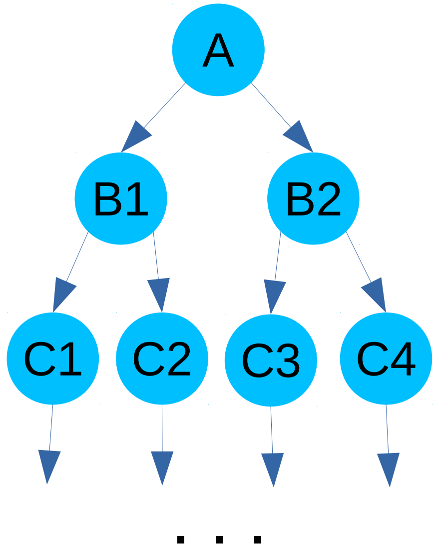
$(2) \implies (1)$: if a trace in $[\![L]\!]$ fails, a good client is definable such that $[\![L;C]\!]$ fails

Symbolic Execution

Given a program *M* with free variables $x_1, x_2, \ldots, x_n$

Execute *M* using:

- symbolic values v_i in place of x_i
- Symbolic environment σ
- Path condition pc

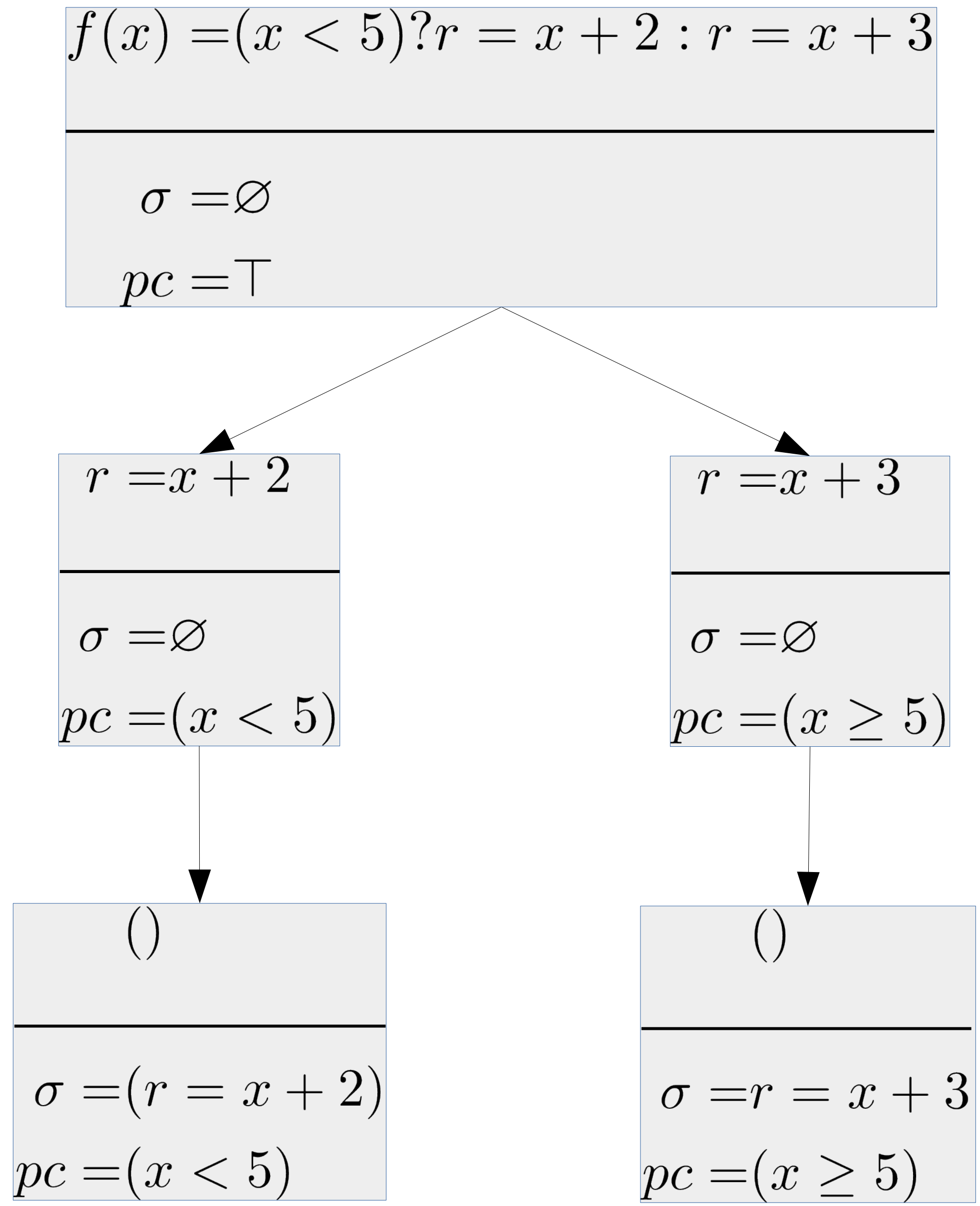


Goal:

explore the computation tree of *M* by independently executing each path in it

Example

```
def f(x):
  if x < 5:
    r = x + 2
  else:
    r = x + 3
```



$$\mathcal{M} \models \sigma \wedge pc$$

Symbolic Execution

Add symbolic environment and path condition; check for assertion violations

Symbolic branching on assertions:

$$(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \rightarrow_s (E[\texttt{assert}(0)], \sigma, pc \wedge (\kappa = 0), k)$$
$$(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \rightarrow_s (E[()], R, \sigma, pc \wedge (\kappa \neq 0), k)$$

Updating the symbolic environment:

$$(E[!r], R, \sigma, pc, k) \rightarrow_s (E[\sigma(r)], R, \sigma, pc, k)$$
$$(E[r := \tilde{v}], R, \sigma, pc, k) \rightarrow_s (E[()], R, \sigma[r \mapsto \tilde{v}], pc, k)$$

Symbolic branching on conditionals:

$$(E[\texttt{if}\ \kappa\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, \sigma, pc, k) \rightarrow_s (E[M_0], R, \sigma, pc \wedge (\kappa = 0), k)$$
$$(E[\texttt{if}\ \kappa\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, \sigma, pc, k) \rightarrow_s (E[M_1], R, \sigma, pc \wedge (\kappa \neq 0), k)$$

Symbolic Games

Symbolic games: moves involve symbolic values, and a symbolic environment and path condition are used to model each path

Obtain symbolic games by:

- Extending game configurations with a symbolic environment (σ) and a path condition (pc)
- Transforming concrete moves into *symbolic moves* by allowing players to play symbolic values (free variables)
- Using symbolic execution as internal moves

Results in configurations:

$$(M, R, \sigma, \mathcal{E}, \mathcal{P}, \mathcal{A}, pc, k)_p$$

Proponent Configuration

$$(l, R, \sigma, \mathcal{E}, \mathcal{P}, \mathcal{A}, pc, k)_o$$

Opponent Configuration

Symbolic DAO Attack

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

$$C_0 \xrightarrow{wdraw(\kappa_1)?} (wdraw(\kappa_1), -, (wdraw, 1) :: \varepsilon, -, 0)_p$$
$$\to^* (E[send(\kappa_1)], -, (wdraw, 1) :: \varepsilon, -, 1)_p$$
$$\xrightarrow{send(\kappa_1)?} (1, -, (send, E) :: \ldots, -, 1)_o \qquad E = \bullet; funds :=!funds - \kappa_1; \mathtt{assert}(!funds \geq 0)$$
$$\xrightarrow{wdraw(\kappa_2)?} (wdraw(\kappa_2), -, (wdraw, 2) :: \ldots, -, 1)_p$$
$$\to^* (E'[send(\kappa_2)], -, (wdraw, 2) :: \ldots, -, 2)_p \qquad E' = \bullet; funds :=!funds - \kappa_2; \mathtt{assert}(!funds \geq 0)$$
$$\xrightarrow{send(\kappa_2)?} (2, -, (send, E') :: \ldots, -, 2)_o$$
$$\xrightarrow{send()!} (E'[()], -, (wdraw, 2) :: \ldots, -, 2)_p$$
$$\to^* ((), -, S[funds \mapsto 50 - \kappa_2], (wdraw, 2) :: \ldots, -, 2)_p$$
$$\xrightarrow{wdraw()!} (1, -, S[funds \mapsto 50 - \kappa_2], (send, E) :: \ldots, -, 1)_o$$
$$\xrightarrow{send()!} (E[()], -, S[funds \mapsto 50 - \kappa_2], (wdraw, 1) :: \varepsilon, -, 1)_p$$
$$\to^* (E[assert(!funds \geq 0)], -, S[50 - \kappa_2 - \kappa_1], (wdraw, 1) :: \varepsilon, -, 1)_p$$

```
public send;
abstract withdraw;

call_counter := 0;
send = λm.
 if   !call_counter==0
 then withdraw(κ₂); skip;
 else skip

main = λ().withdraw(κ₁)
```

$$\boxed{\begin{aligned} pc = &\ (\kappa_1 \leq 50) \land (\kappa_2 \leq 50) \land \neg(50 - \kappa_2 - \kappa_1 \geq 0) \\ \{(\kappa_1 \mapsto 1), (\kappa_2 \mapsto 50)\} \vDash &\ (1 \leq 50) \land (50 \leq 50) \land \neg(-1 \geq 0) \end{aligned}}$$

Soundness and Correctness of SE

Sound Errors: a library assertion violation is found iff the error is reached by executing the counterexample on the linked library-client system

- i.e. produces no false positives

Formally:

(I) **Soundness:** For any library L:

L concretely reaches final value *χ via trace τ and bounds k,l, iff there exists a client C such that L;C reaches χ with some bound k'*

(II) **Correctness:** For any library L:

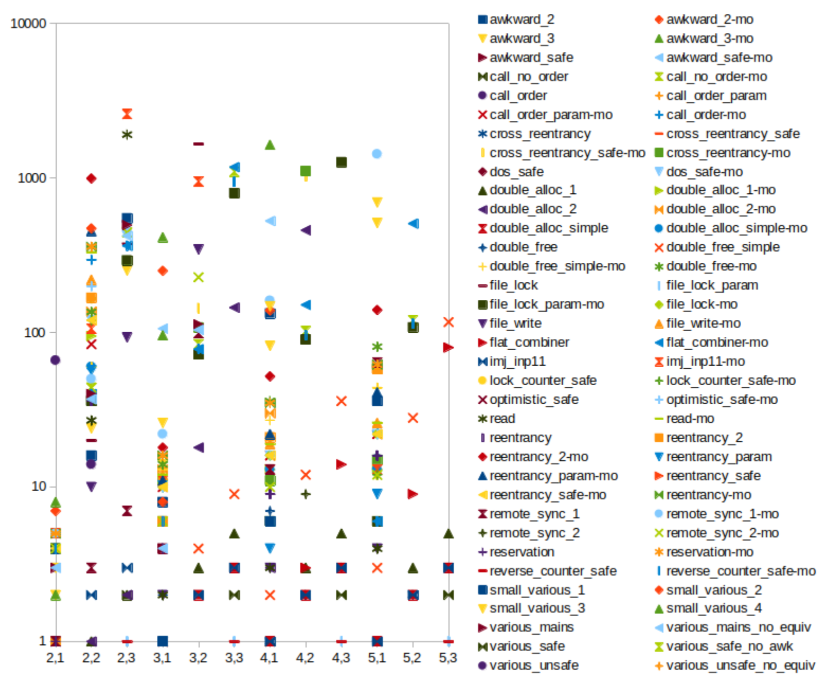
L symbolically reaches final value *χ with a satisfiable path condition, iff*
L can concretely reach the concrete equivalent of χ via the same trace

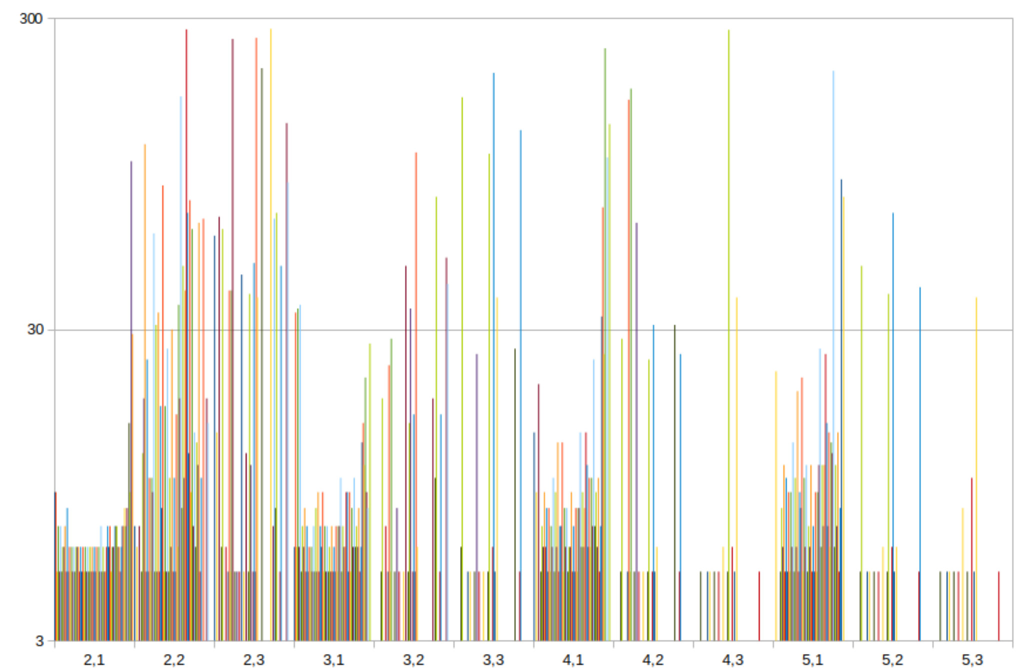
(III) **Sound Errors (I.1) ↔ (II.2):** corollary from (I) and (II)

Implementation: HOLiK

https://github.com/LaifsV1/HOLiK

- Implemented on the **K Semantic Framework** [Rosua and Serbanuta. JLAP 2010]
  - Semantic framework based on rewrite systems
- Benchmark (70 files) exceeds capability of standard techniques
  - Some tools partially cover open programs (e.g. KLEE, CBMC, EtherTrust)





Demo: HOLiK

Consider the DAO library seen before

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

What does HOLiK say about it?

Conclusions and Future Work

- We feasibly found difficult higher-order errors
- In practice, most errors seem to be shallow
- Techniques that find higher-order errors even on small programs seem useful in practice
  - e.g. Real DAO function was <100 LoC yet very costly
- Compositionality could be used for **modular verification**
  - Decomposing programs into small components that fit in memory
  - Guiding analysis of components using known traces
- Possible unbounded verification through **Abstract Interpretation**, or **Push-Down Systems**

Comparison with SCV

Software Contract Verifier [Nguyen et al. 2018]

- Total verification tool for Racket contracts (refinement types)
- Abstract interpretation of the so-called “Demonic Context”
- Demonic context equivalent to Games (both are complete semantics)

Comparison:

- SCV executes faster due to over-approximation
  - Up to an order of magnitude faster
- SCV over-approximation looses accuracy
  - Safe and unsafe DAO are indistinguishable to SCV
  - 33% of errors are not sound
- Games work as a foundational theory and could be a viable alternative
  - HOLiK checks at least medium-sized programs (<1000 LoC)
  - Real-world HO bugs are difficult to find, even on small programs
  - Checking if an error reported by SCV is real is not trivial

Program	LoC	Traces	Time (s)	LoC	Errors	Time (s)	False Errors
ack	17	0	6.0	9	N/A	2.4	N/A
ack-simple	13	0	6.5	9	0	2.4	0
ack-simple-e	13	1	6.5	9	2	2.5	0
dao	10	0	5.0	15	1	2.6	1
dao-e	16	1	5.5	15	1	2.7	0
dao-various	85	5	22.5	122	10	3.0	5
dao2-e	85	10	23.5	122	10	2.9	0
escape	9	0	5.0	9	0	2.6	0
escape-e	9	2	5.0	10	1	2.7	0
escape2-e	10	14	6.0	10	1	2.7	0
factorial	10	0	5.0	9	0	2.2	0
mc91	12	0	5.0	9	1	2.2	1
mc91-e	12	1	5.0	8	1	2.4	0
mult	14	0	5.0	11	2	2.7	2
mult-e	14	1	5.0	11	2	2.4	0
succ	7	0	5.0	7	1	2.5	1
succ-e	7	1	5.0	7	1	2.8	0
various	116	19	14.0	108	11	6.2	5
total	459	55	140.5	500	45	49.8	15

Comparison of HOLiK (left) and SCV (right).