Static analysis of modern software systems: Taming control-flow

Matt Might
University of Utah
matt.might.net
ucombinator.org
Problem

Software fails.
Problem

- Software fails because we can’t engineer it.
- We can’t engineer what we can’t predict.
- We can’t predict the behavior of software.
Message

• Static analysis of modern software is hard!
• Control-flow analysis is the gatekeeper.
• Yet, precise control-flow analysis is possible.
Goal

- Optimization
- Parallelism
- Security
- Correctness

CFA++

CFA

Static analysis
Goal

- Optimization
- Parallelism
- Security
- Correctness

CFA++

CFA

Static analysis
Software “engineering”
Why we need software engineering
Security vulnerabilities

$80\ billion\ in\ cyber-crime\ each\ year.\n
FBI
Cost and cause of insecurity

- Vulnerabilities
- Fraud
- Dumb Employees
- DoS

Cost of cybercrime ($80bn)
Cost and cause of insecurity

Vulnerabilities
Fraud
Dumb Employees
DoS

Cost of cybercrime ($80bn)

Source: CSI/FBI Survey 2007
Cost and cause of insecurity

- Vulnerabilities
- Fraud
- Dumb Employees
- DoS

Cost of cybercrime ($80bn)

- Vulnerabilities: 40%
- Fraud: 38%
- Dumb Employees: 17%
- DoS: 5%

Type of vulnerability

- Buffer Overflow: 53%
- Injection: 7%
- Int Overflow: 13%
- Format String: 23%
- Other: 3%

Source: CSI/FBI Survey 2007

Source: US CERT, Feb 2008
Software bugs

Bugs cost U.S. economy $60 billion annually.

NIST
Why bugs are bad

<exploding-rocket-video />
$10 billion
Parallelism is here
Parallelism is here

Source: Intel
Parallelism is here

Source: Intel
Parallelism is here

Source: Intel
Parallelism is here

We are here.

Source: Intel
Parallelism is here

1968

2010

Source: Intel
Tomorrow’s software

Low dependence  High locality
Tomorrow’s software

Low dependence

High locality

The future?
Bottom line

If we want software that is...

• ...more parallel,
• ...more correct,
• ...more secure,

then we need engineering.
Why can’t we predict what software will do?
Why can’t we predict what software will do?

Because Alan Turing said we can’t.
“Thou shalt not write a program which determines whether a program halts.”
Banned by corollary

• Will a program eventually do $X$?
• Will a program never do $Y$?
A “loop” hole

• Always answering “yes” or “no” is impossible.
• Answering “yes,” “no” or “maybe” is allowed.
The static analysis game
The static analysis game
The static analysis game

MAX++

*a++ =

*0
The static analysis game

MAX++

*a++ =

*0
The static analysis game

MAX++

*a++ =

*0
The static analysis game

"Full employment theorem." - Appel
Why analyzing modern software is hard
What happens here?

animal.eat(food);
What happens here?

What is animal?

animal.eat(food);

What is food?
What happens here?

```java
void process (Animal animal) {
    food = world.gather() ;
    animal.eat(food);
}
```
What happens here?

Who calls `process`?

```java
void process (Animal animal) {
    food = world.gather() ;
    animal.eat(food);
}
```

What is `world`?
The control-flow problem
The control-flow problem
Before we can do anything interesting, we must bound interprocedural control-flow.
Essence of the problem

Value = Object
Essence of the problem

Value = Object
  = Class + Record
Essence of the problem

Value = Object
= Class + Record
⊆ Code + Data
Which language is the paragon of value = code + data?
\( \lambda \)-calculus
Assertion

If we can analyze \( \lambda \)-calculus expressions, we can analyze object-oriented programs.
λ-calculus (Church, 1928)
\( \lambda \text{-calculus (Church, 1928)} \)

- Minimalist, universal language

Alonzo Church
$\lambda$-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  - $\nu$ [variable]
  - $e_1(e_2)$ [function application]
  - $\lambda \nu. e$ [anonymous function]
λ-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  - $\nu$ [variable]
  - $e_1(e_2)$ [function application]
λ-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  - $v$ [variable]
  - $e_1(e_2)$ [function application]
  - $\lambda v. e$ [anonymous function]

Alonzo Church
Lisp and Scheme

- \( v \equiv v \)
- \( f(e) \equiv (f \ e) \)
- \( \lambda v. e \equiv (\text{lambda } (v) \ e) \)
λ-fortified

- Lisp
- SML
- Haskell
- Scala
- Java
- C#
- C++ (Boost)
- Python
- Ruby
- Smalltalk
- JavaScript
- PHP(!)
One rule: $\beta$-reduction

$$(\lambda v. v^2)(3)$$
One rule: $\beta$-reduction
Another interpretation

- Functions ≡ Closures
- \textit{Closure} = \lambda \times Env
- \textit{Env} = \textit{Var} \rightarrow \text{Value}
- \textbf{Ex:} \((\lambda x. x + z, [z \mapsto 1])\)
Essence of the essence

- Value = Code + Data
- Closure = $\lambda + \text{Environment}$
Control-flow question

Given a call site $f(x)$, what could $f$ be?
Control-flow scenarios

\[ f(x) \]
Control-flow scenarios

\[
\text{let } f = \lambda z. z \\
\text{in } f(x)
\]
Control-flow scenarios

\[ \lambda f. f(x) \]
Control-flow analysis

A control-flow analysis conservatively approximates the procedures which may be invoked at a given call site.
Control-flow analysis

A control-flow analysis conservatively approximates the procedures which may be invoked at a given call site.

A value-flow analysis conservatively approximates the values to which an expression may evaluate.
Techniques for CFA

- Ad hoc techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation
Techniques for CFA

- Ad hoc techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation
Techniques for CFA

- Ad hoc techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation
Constraint-based 0CFA
What is 0CFA?

Lambda-flow analysis.
The 0CFA approximation

- Value = Code x Data
- Closure = Lambda x Env
- Object = Class x Record
The 0CFA approximation

- Value = Code
- Closure = Lambda
- Object = Class
0CFA

\[ e_1(e_2) \]
$\lambda v. e_b$

$e_1(e_2)$

$val$
$\lambda v. e_b$

$e_1(e_2)$

$val$
\( \lambda v. e_b \in \text{FlowsTo}[e_1] \) and \( val \in \text{FlowsTo}[e_2] \)

\( val \in \text{FlowsTo}[v] \)
$\lambda v. e_b \Rightarrow e_1(e_2) \Rightarrow val$
0CFA

\( \lambda v.e_b \)

\( e_1(e_2) \)

val
$\lambda v. e_b \in \text{FlowsTo}[e_1]$ and $val \in \text{FlowsTo}[e_b]

val \in \text{FlowsTo}[e_1(e_2)]$
0CFA

\[ \lambda v.e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad val \in \text{FlowsTo}[e_b] \]

\[ val \in \text{FlowsTo}[e_1(e_2)] \]
0CFA

\[ \lambda v. e_b \in \text{FlowsTo}[\lambda v. e_b] \]

\[ \lambda v. e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad \text{val} \in \text{FlowsTo}[e_b] \]
\[ \text{val} \in \text{FlowsTo}[e_1(e_2)] \]

\[ \lambda v. e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad \text{val} \in \text{FlowsTo}[e_2] \]
\[ \text{val} \in \text{FlowsTo}[v] \]
0CFA (Palsberg, 1995)

\[ \{\lambda v.e_b\} \subseteq \text{FlowsTo}[\lambda v.e_b] \]

\[ \lambda v.e_b \in \text{FlowsTo}[e_1] \]
\[ \text{FlowsTo}[e_b] \subseteq \text{FlowsTo}[e_1(e_2)] \]

\[ \lambda v.e_b \in \text{FlowsTo}[e_1] \]
\[ \text{FlowsTo}[e_2] \subseteq \text{FlowsTo}[v] \]
0CFA (Palsberg, 1995)

\[
\{\lambda v.e_b\} \subseteq \text{FlowsTo}[\lambda v.e_b]
\]

\[
\lambda v.e_b \in \text{FlowsTo}[e_1] \\
\text{FlowsTo}[e_b] \subseteq \text{FlowsTo}[e_1(e_2)]
\]

\[
\lambda v.e_b \in \text{FlowsTo}[e_1] \\
\text{FlowsTo}[e_2] \subseteq \text{FlowsTo}[v]
\]
0CFA (Palsberg, 1995)

\[
\{\lambda v.e_b\} \subseteq \text{FlowsTo}[\lambda v.e_b]
\]

\[
\lambda v.e_b \in \text{FlowsTo}[e_1] \quad \Rightarrow \quad \text{FlowsTo}[e_2] \subseteq \text{FlowsTo}[e_1(e_2)]
\]

\[
\lambda v.e_b \in \text{FlowsTo}[e_1] \quad \Rightarrow \quad \text{FlowsTo}[e_2] \subseteq \text{FlowsTo}[v]
\]

+ Constraint Solver

= Control-flow analysis
Applications

- Classic data-flow analysis
- Data-flow optimizations
- Global register allocation
- Defunctionalization
- Static method resolution
- Global constant propagation
- Global copy propagation
- Loop detection/optimization
- Escape analysis
- Constant folding
A problem with the “CFA-first” approach
Problem: Cross-flow

map f list
Problem: Cross-flow

\[
\text{fireMissile}(n) \quad []
\]

\[
\text{map f list}
\]

\[
\text{petBunny}(n) \quad [1,2,3]
\]
Problem: Cross-flow

\[
\text{fireMissile}(n) \quad [\quad ]
\]

\[
\text{map } f \text{ list}
\]

\[
\text{petBunny}(n) \quad [1,2,3]
\]
Problem: Cross-flow

- `fireMissile(n)`
  - `[]`
- `map f list`
  - `[1,2,3]`
- `petBunny(n)`
Problem: Cross-flow

fireMissile(n) → []
map f list

petBunny(n) → [1,2,3]
Problem: Cross-flow

fireMissile(n) \rightarrow \emptyset

map f list

petBunny(n) \rightarrow [1,2,3]
Problem: Cross-flow

fireMissile(n) ↣ []

map f list ↛ [1,2,3]

petBunny(n) ↣ []
Problem: Cross-flow

- fireMissile(n)
- petBunny(n)
- map f list
- []
- [1,2,3]
Problem: Cross-flow

\[ \text{fireMissile}(n) \rightarrow \text{map } f \text{ list} \rightarrow \text{petBunny}(n) \]

\[ [], [1,2,3] \]
Problem: Cross-flow

fireMissile(n) ➔ []

map f list ➔ [1,2,3]

petBunny(n) ➔ [1,2,3]
Solution platform: Small-step abstract interpretation
Small-step advantage
Small-step advantage
Small-step advantage
Small-step advantage
Small-step strategy

- Model program as infinite-state machine
- Approximate program with finite-state machine
Small-step machine
Small-step machine

- Convert program $e$ into machine state $s_0$
Small-step machine

- Convert program $e$ into machine state $s_0$
- Transition from state $s_n$ to state $s_{n+1}$

\[ e \quad \rightarrow \quad s_0 \quad \rightarrow \quad s_1 \quad \rightarrow \quad s_2 \quad \rightarrow \quad s_3 \quad \rightarrow \quad s_4 \quad \rightarrow \quad \ldots \]
Abstract machine

\[ e \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]
Abstract machine

\[ e \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]
Abstract machine

$e$

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots$

$\hat{S}_0$
Abstract machine
Abstract machine

$e$

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots$

$\hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4$

$\hat{s}_{3,1}$
Abstract machine

\[ e \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]

\[ \hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4 \]

\[ \hat{s}_{3,1} \]
Abstract machine

Theorem: The abstract simulates the concrete.
Abstraction
Abstraction
Example: Abstract graph

(letrec ((lp1 (\ (i x) (if (= 0 i) x (letrec ((lp2 (\ (j f y) (if (= 0 j) (lp1 (- i 1) y) (lp2 (- j 1) f (f y)))))) (lp2 10 (\ (n) (+ n i)) x))) (lp1 10 0)))

Figure 1 shows the flow-sensitive, context-sensitive abstract transition graphs generated by this loop first without, and then with, abstract garbage collection. Garbage-collecting environment structure during the exploration of the abstract state space yields an order of magnitude improvement in the size of the state space—enough so that the doubly-nested structure of the loop is visually apparent from the second graph. (Besides the improvement in analytic precision, we also get a secondary benefit in that the processor time and memory space needed to explore the abstract state space are also greatly reduced.)

Abstract garbage collection sets the stage for another technique known as abstract counting [9]. With abstract counting, we track the "cardinality" of an abstract object; that is, we track whether an abstract object currently represents zero, one or more than one concrete values. Suppose we were to use sets of concrete values for our abstract values. Ordinarily, if abstract value A were equal to abstract value B, we could not infer that any concrete value a ∈ A is equal to any concrete value b ∈ B, except for the case where A and B have size one. The ability to transfer abstract equality to concrete equality allows us to more precisely evaluate conditions, e.g. (= x y), in the abstract.

In previous work [9], we developed a higher-order flow-analysis framework, ΓCFA, which synergistically combines abstract counting and abstract garbage collection as we've just outlined above. The benefit of combining the two is that we can use abstract counts to reason more precisely about reachable values during abstract garbage collection. This, in turn, increases the chance that we can cut off even more branches from the abstract transition graph.

Our purpose in this paper is to show how ΓCFA technology can be applied to the problem of model-checking software written in higher-order languages. Our technical contributions are:

1. Enhancing abstract garbage collection by switching from reachability to usability as the criterion for liveness. That is, our garbage collector discards abstract values and environment structure which are "reachable," but whose use is dominated by conditions which have become unsatisfiable. We term this conditional garbage collection.
Example: Abstract graph
Small-step CFA for CPS
Continuation-passing style

\[ \begin{align*}
\nu \in \text{Var} \\
\text{f, e} \in \text{Exp} &= \text{Var} + \text{Lam} \\
\text{lam} \in \text{Lam} &\ ::= (\lambda (\nu_1 \ldots \nu_n) \text{call}) \\
\text{call} \in \text{Call} &\ ::= (\text{f e}_1 \ldots \text{e}_n)
\end{align*} \]
Concrete state-space

\[ \varsigma \in \Sigma = \text{Call} \times \text{BEnv} \times \text{Store} \times \text{Time} \]

\[ \beta \in \text{BEnv} = \text{Var} \rightarrow \text{Addr} \]

\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \text{Clo} \]

\[ \text{clo} \in \text{Clo} = \text{Lam} \times \text{BEnv} \]

\[ a \in \text{Addr} \text{ is a set of addresses} \]

\[ t \in \text{Time} \text{ is a set of time-stamps} \]
Simpler option

\( \varsigma \in \Sigma = \text{Call} \times Env \)

\( \rho \in Env = \text{Var} \rightarrow \text{Clo} \)

\( clo \in Clo = \text{Lam} \times Env \)
Concrete state-space

\[ \varsigma \in \Sigma = \text{Call} \times BEnv \times \text{Store} \times \text{Time} \]
\[ \beta \in \text{BEnv} = \text{Var} \rightarrow \text{Addr} \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \text{Clo} \]
\[ \text{clo} \in \text{Clo} = \text{Lam} \times \text{BEnv} \]
\[ a \in \text{Addr} \text{ is a set of addresses} \]
\[ t \in \text{Time} \text{ is a set of time-stamps} \]
State-spaces
State-spaces
Injector

\[ \mathcal{I} : \text{Call} \rightarrow \Sigma \]
\[ \mathcal{I}(call) = (call, [], [], t_0) \]
State-spaces
State-spaces

\[ \sum \]

\[ I \]

\( \zeta \)

\textit{call}
Factored evaluator

\[ E(v, \beta, \sigma) = \sigma(\beta(v)) \]

\[ E(lam, \beta, \sigma) = (lam, \beta) \]
Concrete semantics

When \( \text{call} = \llbracket (f \ e_1 \ldots e_n) \rrbracket \):

\[
\begin{align*}
\text{(call, } \beta, \sigma, t) & \Rightarrow (\text{call}', \beta'', \sigma', t'), \text{ where} \\
(lam, \beta') &= \mathcal{E}(f, \beta, \sigma) \\
clo_i &= \mathcal{E}(e_i, \beta, \sigma) \\
lam &= \llbracket (\lambda (v_1 \ldots v_n) \text{ call}') \rrbracket \\
t' &= \text{tick}(\text{call, } t) \\
a_i &= \text{alloc}(v_i, t') \\
\beta'' &= \beta'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto clo_i]
\end{align*}
\]
Concrete semantics

When \( \text{call} = \llbracket (f \ e_1 \ldots e_n) \rrbracket \):

\[
\text{call}, \beta, \sigma, t \Rightarrow (\text{call}', \beta'', \sigma', t'), \text{ where }
\]

\[
(\text{lam}, \beta') = \mathcal{E}(f, \beta, \sigma)
\]

\[
clo_i = \mathcal{E}(e_i, \beta, \sigma)
\]

\[
\text{lam} = \llbracket (\lambda (v_1 \ldots v_n) \text{ call}') \rrbracket
\]

\[
t' = \text{tick} (\text{call}, t)
\]

\[
a_i = \text{alloc} (v_i, t')
\]

\[
\beta'' = \beta'[v_i \mapsto a_i]
\]

\[
\sigma' = \sigma[a_i \mapsto \text{clo}_i]
\]
The easy solution

\[ Time = \mathbb{N} \]

\[ Addr = \text{Var} \times \text{Time} \]

\[ \text{tick}(\_, t) = t + 1 \]

\[ \text{alloc}(v, t) = (v, t) \]
State-spaces
State-spaces

\[ \Sigma \]

\[ \mathcal{I} \]

\[ \varsigma \rightarrow \varsigma \]

\[ \sum \]

\[ \varsigma \rightarrow \varsigma \]

\[ \text{call} \]
Abstracting into a control-flow analysis
Abstract state-space

\[\hat{\zeta} \in \hat{\Sigma} = \text{Call} \times \hat{\text{BEnv}} \times \hat{\text{Store}} \times \hat{\text{Time}}\]

\[\hat{\beta} \in \hat{\text{BEnv}} = \text{Var} \rightarrow \hat{\text{Addr}}\]

\[\hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \mathcal{P}(\hat{\text{Clo}})\]

\[\hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\text{BEnv}}\]

\[\hat{a} \in \hat{\text{Addr}} \text{ is a finite set of addresses}\]

\[\hat{t} \in \hat{\text{Time}} \text{ is a finite set of time-stamps}\]
Abstract state-space

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{BEnv} \times \hat{\text{Store}} \times \hat{\text{Time}} \]
\[ \hat{\beta} \in \hat{BEnv} = \text{Var} \rightarrow \hat{\text{Addr}} \]
\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \mathcal{P}(\hat{\text{Clo}}) \]
\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{BEnv} \]
\[ \hat{a} \in \hat{\text{Addr}} \text{ is a } \text{finite} \text{ set of addresses} \]
\[ \hat{t} \in \hat{\text{Time}} \text{ is a } \text{finite} \text{ set of time-stamps} \]
Abstract state-space

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{BEnv} \times \hat{\text{Store}} \times \hat{\text{Time}} \]
\[ \hat{\beta} \in \hat{BEnv} = \text{Var} \rightarrow \hat{\text{Addr}} \]
\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \mathcal{P} (\hat{\text{Clo}}) \]
\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \hat{\text{Lam}} \times \hat{BEnv} \]
\[ \hat{\text{a}} \in \hat{\text{Addr}} \text{ is a finite set of addresses} \]
\[ \hat{\text{t}} \in \hat{\text{Time}} \text{ is a finite set of time-stamps} \]

\[ \varsigma \in \Sigma = \text{Call} \times BEnv \times Store \times Time \]
\[ \beta \in BEnv = \text{Var} \rightarrow \text{Addr} \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \mathcal{P} (\text{Clo}) \]
\[ \text{clo} \in \text{Clo} = \text{Lam} \times BEnv \]
\[ \text{a} \in \text{Addr} \text{ is a set of addresses} \]
\[ \text{t} \in \text{Time} \text{ is a set of time-stamps} \]
Non-recursive

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{BEnv} \times \hat{\text{Store}} \times \hat{\text{Time}} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \to \mathcal{P}(\hat{\text{Clo}}) \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\text{BEnv}} \]

\[ \hat{\beta} \in \hat{\text{BEnv}} = \text{Var} \to \hat{\text{Addr}} \]

\[ \hat{a} \in \hat{\text{Addr}} \text{ is a finite set of addresses} \]

\[ \hat{t} \in \hat{\text{Time}} \text{ is a finite set of time-stamps} \]
Non-recursive

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{\text{BEnv}} \times \hat{\text{Store}} \times \hat{\text{Time}} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \text{Addr} \rightarrow \mathcal{P}(\hat{\text{Clo}}) \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\text{BEnv}} \]

\[ \hat{\beta} \in \hat{\text{BEnv}} = \text{Var} \rightarrow \hat{\text{Addr}} \]

\[ \hat{a} \in \hat{\text{Addr}} \text{ is a finite set of addresses} \]

\[ \hat{t} \in \hat{\text{Time}} \text{ is a finite set of time-stamps} \]
State-spaces
State-spaces

\[ \Sigma \]

\[ \varsigma \rightarrow \varsigma \]

\[ \mathcal{I} \]

\[ \text{call} \]

\[ \hat{\Sigma} \]

\[ \hat{\varsigma} \rightarrow \hat{\varsigma} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \hat{P}(\hat{\text{Clo}}) \]

\[ \hat{\beta} \in \hat{\text{BEnv}} = \hat{\text{Var}} \rightarrow \hat{\text{Addr}} \]

\[ \hat{a} \in \hat{\text{Addr}} \]

\[ \hat{t} \in \hat{\text{Time}} \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} \]

\[ \hat{\text{store}} \]

\[ \hat{\text{env}} \]

\[ \hat{\text{time}} \]

\[ \text{is a finite set of addresses} \]

\[ \text{is a finite set of time-stamps} \]
Injector

\[ \hat{I}(call) = (call, [], [], t_0) \]
State-spaces

\( \Sigma \)

\( \tau \) call

\( \zeta \rightarrow \zeta \)
State-spaces

Σ

call

\hat{\Sigma}

\hat{\sigma} \in \hat{\text{Store}} = \text{Addr} \rightarrow \text{P}((\hat{\text{Clo}}))

\hat{\beta} \in \hat{\text{BEnv}} = \text{Var} \rightarrow \hat{\text{Addr}}

\hat{a} \in \hat{\text{Addr}}

\hat{t} \in \hat{\text{Time}}

\hat{t} \in \hat{\text{Time}}

is a finite set of addresses

is a finite set of time-stamps
Abstract components

$(\sim) \subseteq \hat{\Sigma} \times \hat{\Sigma}$

$\hat{E} : \text{Exp} \times \hat{BEnv} \times \hat{Store} \rightarrow \mathcal{P} \left( \hat{Clo} \right)$
\[ \mathcal{E} : \text{Exp} \times \overbrace{BEnv} \times \overbrace{Store} \rightarrow \mathcal{P} (\overbrace{Clo}) \]

\[ \mathcal{E}(v, \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}(v)) \]

\[ \mathcal{E}(\text{lam}, \hat{\beta}, \hat{\sigma}) = \{(\text{lam}, \hat{\beta})\} \]
Abstract semantics

When $call = \llbracket (f \ e_1 \ldots e_n) \rrbracket$:

$$(call, \hat{\beta}, \hat{\sigma}, \hat{t}) \leadsto (call', \hat{\beta}'', \hat{\sigma}', \hat{t}')$$, where

$$(lam, \hat{\beta}') \in \mathcal{E}(f, \hat{\beta}, \hat{\sigma})$$

$$\hat{C}_i = \mathcal{E}(e_i, \hat{\beta}, \hat{\sigma})$$

$$lam = \llbracket (\lambda (v_1 \ldots v_n) \ call') \rrbracket$$

$$\hat{t}' = \overline{\text{tick}}(call, \hat{t})$$

$$\hat{a}_i = \overline{\text{alloc}}(v_i, \hat{t}')$$

$$\hat{\beta}'' = \hat{\beta}'[v_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{C}_i]$$
State-spaces

\[ \Sigma \]

\[ \iota \text{ call } \hat{\iota} \]

\[ \varsigma \]

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{\Delta_{\text{BEnv}}} \times \hat{\Delta_{\text{Store}}} \times \hat{\Delta_{\text{Time}}} \]

\[ \hat{\sigma} \in \hat{\Delta_{\text{Store}}} = \text{Addr} \rightarrow \text{P}((\hat{\Delta_{\text{Clo}}})) \]

\[ \hat{\varsigma} \in \hat{\Delta_{\text{BEnv}}} = \text{Var} \rightarrow \hat{\Delta_{\text{Addr}}} \]

\[ \hat{\beta} \in \hat{\Delta_{\text{BEnv}}} = \text{Var} \rightarrow \hat{\Delta_{\text{Addr}}} \]

\[ \hat{\alpha} \in \hat{\Delta_{\text{Addr}}} \text{ is a finite set of addresses} \]

\[ \hat{t} \in \hat{\Delta_{\text{Time}}} \text{ is a finite set of time-stamps} \]
State-spaces

\[ \Sigma \]

\[ \mathcal{I} \]

\[ \text{call} \]

\[ \hat{\mathcal{I}} \]

\[ \zeta \]

\[ \hat{\zeta} \]

\[ \hat{\Sigma} \]

\[ \hat{\sigma} \in \hat{\Sigma} \]

\[ \hat{\alpha} \in \hat{\Sigma} \]

\[ \hat{\beta} \in \hat{\Sigma} \]

\[ \hat{\tau} \in \hat{\Sigma} \]

\[ \hat{\mu} \in \hat{\Sigma} \]

\[ \text{I} \]

\[ \text{call} \]

\[ \hat{\text{call}} \]

\[ \hat{\Sigma} \]

\[ \text{I} \]

\[ \text{call} \]

\[ \hat{\text{call}} \]
Abstraction maps

\[ \alpha(\text{call}, \beta, \sigma, t) = (\text{call}, \alpha(\beta), \alpha(\sigma), \alpha(t)) \]
\[ \alpha(\beta) = \lambda v. \alpha(\beta(v)) \]
\[ \alpha(\sigma) = \lambda \hat{a}. \bigcup_{\alpha(a) = \hat{a}} \alpha(\sigma(a)) \]
\[ \alpha(\text{lam}, \beta) = \{(\text{lam}, \alpha(\beta))\} \]
\[ \alpha(a) \text{ is set by parameter} \]
\[ \alpha(t) \text{ is set by parameter} \]

\[ \hat{\sigma} \sqcup \hat{\sigma}' = \lambda \hat{a}. (\hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})) \]
State-spaces

\[ \sum \]

\[ \iota \] call \[ \hat{\iota} \]

\[ \varsigma \]

\[ \hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \hat{\mathcal{BEnv}} \times \hat{\mathcal{Store}} \times \hat{\mathcal{Time}} \]

\[ \hat{\sigma} \in \hat{\mathcal{Store}} = \hat{\text{Addr}} \rightarrow \hat{\text{P}}(\hat{\text{Clo}}) \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\mathcal{BEnv}} \]

\[ \hat{\beta} \in \hat{\mathcal{BEnv}} = \text{Var} \rightarrow \hat{\text{Addr}} \]

\[ \hat{\alpha} \in \hat{\text{Addr}} \]

\[ \hat{\text{t}} \in \hat{\text{Time}} \] is a finite set of time-stamps

\[ \hat{\varsigma} \]

\[ \hat{\text{Addr}} \] is a finite set of addresses
State-spaces

\[ \Sigma \]

\[ \text{call} \]

\[ \hat{\Sigma} \in \hat{\Sigma} = \text{Call} \times \hat{\beta} \times \hat{\text{Store}} \times \hat{\text{Time}} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \hat{\text{P}}(\hat{\text{Clo}}) \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \hat{\text{Lam}} \times \hat{\beta} \times \hat{\text{BEnv}} \]

\[ \hat{\beta} \in \hat{\text{BEnv}} = \text{Var} \rightarrow \hat{\text{Addr}} \]

\[ \hat{\text{a}} \in \hat{\text{Addr}} \]

\[ \hat{\text{t}} \in \hat{\text{Time}} \]

\[ \hat{\text{t}} \]

\[ \text{is a finite set of addresses} \]

\[ \text{is a finite set of time-stamps} \]
Soundness

\[ \varsigma \Rightarrow \varsigma' \]
\[ \alpha \downarrow \alpha \downarrow \]
\[ \sqsubseteq \sqsubseteq \]
\[ \hat{\varsigma} \sim \sim \hat{\varsigma}' \]

**Theorem:** If the concrete takes a step, then the abstract can take a matching step.
State-spaces

\[ \Sigma \]

\[ \varsigma \]

\[ \varsigma \hat{} \in \hat{\Sigma} = \text{Call} \times \hat{\mathcal{B}\text{Env}} \times \hat{\mathcal{S}\text{tore}} \times \hat{\mathcal{T}\text{ime}} \]

\[ \hat{\sigma} \in \hat{\mathcal{S}\text{tore}} = \hat{\mathcal{L}\text{am}} \times \hat{\mathcal{B}\text{Env}} \]

\[ \hat{a} \in \hat{\mathcal{A}\text{ddr}} \]

\[ \hat{t} \in \hat{\mathcal{T}\text{ime}} \]

\[ \text{is a finite set of addresses} \]

\[ \text{is a finite set of time-stamps} \]
State-spaces

$\Sigma$

$\iota$ call $\hat{\iota}$

$\varsigma \in \hat{\Sigma} = \text{Call} \times \hat{\mathcal{B}\text{-Env}} \times \hat{\mathcal{S}\text{-Store}} \times \hat{\mathcal{S}\text{-Time}}$

$\hat{\sigma} \in \hat{\mathcal{S}\text{-Store}} = \hat{\mathcal{L}\text{am}} \times \hat{\mathcal{B}\text{-Env}}$

$\hat{\beta} \in \hat{\mathcal{B}\text{-Env}} = \text{Var} \rightarrow \hat{\mathcal{S}\text{-Addr}}$

$\hat{\alpha} \in \hat{\mathcal{S}\text{-Addr}}$ is a finite set of addresses

$\hat{\tau} \in \hat{\mathcal{S}\text{-Time}}$ is a finite set of time-stamps
Application: Buffer-overflow checks
Logic-flow analysis
Logic-flow analysis
Logic-flow analysis

i < length(a)
Logic-flow analysis

\[ i < \text{length}(a) \]
Beyond CFA

- Abstract G.C.
- Nondeterministic A.I.
- Frame-string analysis
- Abstract counting
- Logic-flow analysis
- Dependence analysis
- See matt.might.net
Thanks!