INCREASING PROGRAMMER PRODUCTIVITY IN BUILDING RELIABLE PROGRAMS BY SYNTHESIZING ANNOTATIONS

MADHUSUDAN PARTHASARATHY
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

EXCAPE WEBINAR
JOINT WORK WITH...

Xiaokang Qiu
Pranav Garg
Edgar Pek

Andrei Stefanescu
Christof Loeding (RWTH)
Daniel Neider (RWTH)

+ Sam King’s systems group at Illinois
BUILD RELIABLE SOFTWARE

Building systems with **proven** reliability and security guarantees.

- Not the same as testing or finding bugs

**General** stable applications with lots of users

**Specific** apps with lots of users but written by a small group

**Systems Software/OS/Compilers/Platforms/Services**

**Apps written within a company**

**CMS**
**WORDPRESS**

**Singularity**
**SELinux**
**ExpressOS**

**Embedded Software**

**Linux**
**Android**

**MS Hypervisor**

**Amazon Web Services**
**AUTOMATIC METHODS**

- Bug-finding tools are very effective, but verification tools work only for shallow properties or particular domains.

<table>
<thead>
<tr>
<th>Property</th>
<th>Deep</th>
<th>Shallow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very constrained</td>
<td>SLAM</td>
<td>ASTREE</td>
</tr>
<tr>
<td>software domain</td>
<td>ASTREE</td>
<td>SLAM</td>
</tr>
<tr>
<td>Very general</td>
<td>Static analysis / AI</td>
<td>Automatic type-checking</td>
</tr>
<tr>
<td>domains</td>
<td>(buff overflow, generic safety, etc.)</td>
<td>(buff overflow, generic safety, etc.)</td>
</tr>
</tbody>
</table>
METHODOLOGY: FLOYD-HOARE STYLE REASONING

• User provides specification using modular annotations
  (pre/post conditions, class invariants, loop invariants)

• **Automatic** generation of verification conditions (e.g. Boogie)
  (pure logic formulas whose validity needs to be verified)

• **Example:** 

\[
\begin{align*}
|x > y| & \quad \text{x:=x+2; y:=y+1; } \\
& \text{gives verification condition} \\
& x_{old}, x_{new}, y_{old}, y_{new} : \\
& ((x_{old} > y_{old} \land x_{new} = x_{old} + 2 \land y_{new} = y_{old} + 1) \rightarrow x_{new} > y_{new})
\end{align*}
\]

• **Validity of verification conditions checked mostly automatically**
  (e.g., SMT solvers)
CHALLENGES OF FH REASONING: BURDEN OF ANNOTATIONS

• Pre/post conditions for logical macro-modules
  - Natural to specify; absolutely essential for modular verification!
  - This is truly the specification at the code-level

• Pre/post conditions for smaller modules
  • Harder to write/specify
  • Seems inferrable

• Loop Invariants
  • Much harder to write… awkward as well, with border conditions
  • Programmer shouldn’t have to write most of them

• Proof tactics
  • Extremely hard to write, even by experts
  • Intimate knowledge of underlying
With *synthesis of annotations and proof tactics* we can provide programmers with tools adequate to build reliable/secure systems with reasonable overhead.

This will enable a new class of reliable software to be built that hasn’t been possible so far.

You don’t have to have a PhD in formal methods to use these tools. We can, say over a semester course, be able to enable programmers to write reliable code with proven properties.

Caveat:
- Reasonably deep properties, but not very deep/functional ones
Build reliable apps

Design spec logics

Automate proofs

Automate loop inv annotations

Build tools

ExpressOS: android platform
[ASPLOS’13]

Dryad sep logic

Natural proofs
[POPL12, PLDI13]

ICE Learning
(ML/ALT)
[CAV13, SAS13]

VCC-Dryad
InvLearn
EXPRESSOS: A SECURE FOUNDATION FOR MOBILE APPS

Drivers

Window mgnt.

Storage

Network

Runtime

L4

ExpressOS (15K LOC)

Hardware

Android apps

Android sysecalls

Android apps
EXPRESSOS: A SECURE FOUNDATION FOR MOBILE APPS

In collaboration with King’s systems group at Illinois

First OS architecture that provides verifiable, high-level abstractions for building mobile applications.

- Secure storage
- Memory isolation
- UI isolation
- Secure IPC

Application state is private by default.

- Meta-data on files controls access by apps; only app with appropriate permissions can access files
- Every page is encrypted before sending to the storage service.
- Memory isolation between apps
- A page fault handler serves a file-backed page for a process, the file has to be opened by the same process.
- Only the current app can write to the screen buffer.
- IPC channel correctness

And works!
Properties expressed as modular contracts and object invariants. Proved modularly using a combination of static analysis and Hoare-style verification.
## EVALUATION

### Vulnerability study

<table>
<thead>
<tr>
<th>Location</th>
<th>Number</th>
<th>Prevented</th>
</tr>
</thead>
<tbody>
<tr>
<td>The core of the kernel</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Libraries of applications</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>Systems services</td>
<td>240</td>
<td>226</td>
</tr>
<tr>
<td>Sensitive applications</td>
<td>32</td>
<td>27</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>383</strong></td>
<td><strong>364</strong></td>
</tr>
</tbody>
</table>

![Page load latency on web browsing](image)

- Android-x86
- L4Android
- ExpressOS
Verify invariants in ExpressOS [@UIUC, ASPLOS'13]
- “Every two assigned memory chunks are disjoint“
- “The start addresses are sorted along the doubly-linked list”
- “The memory allocator modifies only this doubly-linked list”

Key Challenge:
How do we state these properties?
These properties involve quantification; how do we prove them?
→ Dryad and Natural Proofs
Current logics/techniques don’t really work when manipulating dynamic heaps (pointer structures: lists/trees/… objects, etc )
FUNCTIONAL VERIFICATION OF HEAP-MANIPULATING PROGRAMS

Expressive Logics:
Separation logics, HOL, Matching logic, etc.

Decidable Logics:
LISBQ, CSL, STRAND etc.

Expressive

keep expressiveness

Dryad and Natural proofs
[POPL’12, PLDI’13]

give up decidability
sound but incomplete
preserve automaticity

Automatic

Bug-finding
NATURAL PROOFS: IN A NUTSHELL

- Handle a logic that is very expressive (inevitably undecidable)

- Retain automaticity at the same level as decidable logics

- Identify a class of simple and natural proofs $N$ such that
  - Many correct programs can be proved using a proof in class $N$
  - The class $N$ is effectively searchable (searching thoroughly for a proof in $N$ is efficiently decidable using an SMT solver)
  - “Unfold recursive defs + formula abstraction”
OVERVIEW

Aim:

To provide a single logical framework that supports natural proofs for general properties of structure, data, and separation

Logic design:

Many ways to say the same thing; some ways are better!
Reursion is better to exploit in proofs than arbitrary quantification.

Dryad: A recursive dialect of separation logic

- no explicit quantification, but supports recursive definitions
- admits a deterministic translation to classic logic
- Develop natural proofs for this logic using decision procedures (powered by SMT solvers)

Separation Logic 101 [CSL’01, LICS’02]

- Key insight: formulas should be defined on local (small) heaplets by default, not the global heap
BST*: \( (x) = (x = \text{nil} \land \text{emp}) \lor (x \mapsto x_l, x_r, x_k) \)

Keys*: \( (x) = (x = \text{nil}: \emptyset; \langle x_l, x_r, x_k \rangle \ast \text{true}; \{ x_k \} \cup \text{keys}^* (x_l) \cup \text{keys}^* (x_r) ; \) \)

Recursion base case:
- empty heaplet: \( \text{root} \)
- Empty set: \( \text{the left branch: a BST} \)
- The set of keys is the union of the keys stored in the root and the left/right sub-trees

Base case: empty set

Recursive function

Separating conjunction

\((\text{loop invariant for "bst-search"}):\)

- \( \text{root}\) points to a BST and \( \text{curr}\) points into the BST;
- \( k\) is stored in the BST \iff \( k\) is stored under \( \text{curr} \)
Verification in 4 steps

1. **Translate Dryad to classic logic**
   Classical logic with recursion, sets, integers, etc. (undecidable)

2. **VC-Generation** (compute strongest-post)

3. **Unfolding recursive definitions across the footprint touched by the program**
   (no loss of completeness)

4. **Formula Abstraction**
   (recursive definitions become uninterpreted, solvable using SMT, sound but incomplete)
EXPERIMENTAL EVALUATION

A prototype verifier with Z3 as the backend solver

(more details at http://web.engr.illinois.edu/~qiu2/dryad/)

~ 100 programs manipulating data-structures automatically verified

10+ complex data structures
  singly-linked list, sorted list, doubly-linked list, cyclic list, max-heap, BST, Treap, AVL tree, red-black tree, binomial heap, ...

59 Dryad-annotated standard library programs
  • insert, delete, search, reverse, rotate, ...
  • recursive & iterative implementation
  • Full-functional, partial correctness (e.g., data structure invariants, expected set-of-keys, …)
47 Dryad-annotated programs from open source libraries
Glib library, OpenBSD library, Linux kernel, …

A secure foundation for ExpressOS [@Illinois, ASPLOS’13]
• Verified memory isolation, UI isolation, etc.
• supports Android apps

• All the VCs were automatically proved by Z3
• Few routines exceed 1 sec, even fewer exceed 100 secs

“RBT-delete_iter” spent 225 secs, “binomial-heap-merge_rec” spent 153 secs
VCC+DRYAD: NATURAL PROOFS FOR C

Natural proofs work for a toy programming language. But what about a real language? Challenges: memory model, pointers, types, quantification associated with these....

VCC+Dryad:
- Add natural proofs to VCC
- Replace VCC’s original translation to annotations that force natural proofs
- VCC → Boogie → Z3 ensure verification goes through
- Preliminary results: works very well
  Proves C code manipulating data-structures correct.
Build reliable apps

VCC-Dryad

Build tools

Design spec logics

Automate proofs

Automate loop inv annotations

ExpressOS: android platform [ASPLOS’13]

Dryad sep logic

Natural proofs [POPL12, PLDI13]
INVARIANT GENERATION

Most automated methods generate invariants.

• Abstract Interpretation (lub, widening)
• Predicate abstraction (model-checking)

Challenges:

• Tuning the invariant to the level required by the specification
• Handling quantified invariants over unbounded data: arrays, lists, heaps, etc.
• Handling real code
  (sound precise abstractions are hard to write)
Renewed interest in application of learning to synthesizing invariants [Sharma et al. CAV’12, SAS’13], [Garg et al. CAV’13], [Kong et al. APLAS-10]

Advantages with respect to white-box techniques:
- verification of complex program with simple invariants
- generalization
Black-box learning of invariants

First approach:

• Passive learning.
• Generalize from concrete examples gathered on dynamic runs (a la Daikon)

Problems:

• Invariant independent of spec (similar to abstract int)
• In fact, very similar to recent AI techniques proposed recently (see abstract Houdini; Thakur et al)
ICE learning: An active learning model

Active learning:

• Learner proposes hypothesis $H$
• Teacher verifies using the invariant and checks if it is adequate $(\text{SMT; natural proofs})$
• Finds concrete configurations that tell the learner what is wrong
Traditional learning algorithms are not robust

Requirements on a loop invariant: \( pre;S; \ while(b) \ do \ L; \ od \ S'; \ post \)

\[
\text{strpost}(pre,S) \Rightarrow H
\]
\[
p \in pre, p' \notin H \quad \Rightarrow \quad \text{Add } p' \text{ as a positive example}
\]

\[
\text{strpost}(H \land \neg b, S') \Rightarrow post
\]
\[
p \in H, p' \notin post \quad \Rightarrow \quad \text{Add } p \text{ as a negative example}
\]

\[
\text{strpost}(H \land b, L) \Rightarrow H'
\]
\[
p \in H, \ p' \notin H \quad \Rightarrow \quad \text{Either add } p \text{ as a negative example OR add } p' \text{ as a positive example}
\]

- Teacher makes an arbitrary choice \( \Rightarrow \) divergence and bias.
ICE: Learning using Examples, Counter-examples and Implications

- To refute non-inductiveness of H, the teacher communicates \((p, p')\)
  - if \(p \in H\), then \(p' \notin H\) -- learner’s choice depends on simplicity, etc.

- Robust framework
  - progress
  - honest teacher
ICE learning frameworks: Data domains

- Data domains for invariants over scalar variables:
  Template-based synthesis \( \{x_1, \ldots, x_n\} \)

\[
H = \bigvee_i \bigwedge_j s_{ij} v^i_j + s^i_j v^j_2 \leq c_{ij} \in \mathbb{Z}
\]

- Handling examples, counter-examples and implications

Simply as new constraints
ICE learning frameworks: Quantified Invariants

- Quantified invariants over arrays, lists, etc.

- Quantified formulas \( \psi = \forall y_1 \ldots y_k \cdot \varphi(y_1 \ldots y_k) \) form a concept class.

- Quantified Data Automata learning to learn such invariants
  - QDAs accept quantified properties of data-words
  - Adapted RPNI passive learning of automata to QDAs

\[ \forall y_1 y_2 . \left( \text{head} \xrightarrow{\text{next}}^* y_1 \xrightarrow{\text{next}}^* y_2 \Rightarrow \text{data}(y_1) \leq \text{data}(y_2) \right) \]
<table>
<thead>
<tr>
<th>Program</th>
<th>#Disj., #Conj., #Vars.</th>
<th>#Rounds</th>
<th>#Ex.</th>
<th>#Cex.</th>
<th>#Imp.</th>
<th>Time(s) learner</th>
<th>Time(s) teacher</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cegar1</td>
<td>(1,2,2)</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>0.1</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>cegar2</td>
<td>(1,3,3)</td>
<td>6</td>
<td>17</td>
<td>12</td>
<td>24</td>
<td>2.2</td>
<td>1.2</td>
<td>3.4</td>
</tr>
<tr>
<td>fig1</td>
<td>(2,1,2)</td>
<td>7</td>
<td>26</td>
<td>30</td>
<td>18</td>
<td>0.8</td>
<td>1.6</td>
<td>2.4</td>
</tr>
<tr>
<td>tacas06</td>
<td>(3,2,4)</td>
<td>5</td>
<td>26</td>
<td>18</td>
<td>18</td>
<td>6.7</td>
<td>1.4</td>
<td>8.1</td>
</tr>
<tr>
<td>fig3</td>
<td>(2,2,3)</td>
<td>2</td>
<td>17</td>
<td>12</td>
<td>6</td>
<td>0.3</td>
<td>0.4</td>
<td>0.7</td>
</tr>
<tr>
<td>fig8</td>
<td>(1,2,6)</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>fig9</td>
<td>(1,2,2)</td>
<td>4</td>
<td>4</td>
<td>12</td>
<td>12</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>ex23</td>
<td>(1,4,3)</td>
<td>8</td>
<td>61</td>
<td>24</td>
<td>36</td>
<td>9.0</td>
<td>2.0</td>
<td>11.0</td>
</tr>
<tr>
<td>ex7</td>
<td>(1,2,4)</td>
<td>7</td>
<td>29</td>
<td>36</td>
<td>0</td>
<td>1.1</td>
<td>1.4</td>
<td>2.5</td>
</tr>
<tr>
<td>ex4</td>
<td>(1,1,2)</td>
<td>2</td>
<td>17</td>
<td>6</td>
<td>0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>ex11</td>
<td>(1,2,2)</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>4</td>
<td>0.1</td>
<td>0.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>
ICE learning: quantified properties of arrays/lists

<table>
<thead>
<tr>
<th>Program</th>
<th>#Rounds</th>
<th>#Ex.</th>
<th>#Cex.</th>
<th>#Imp.</th>
<th>#States</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>1.3</td>
</tr>
<tr>
<td>copy</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>copy-lt-key</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>13</td>
<td>1.2</td>
</tr>
<tr>
<td>find</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>1.2</td>
</tr>
<tr>
<td>init</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0.6</td>
</tr>
<tr>
<td>max</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0.4</td>
</tr>
<tr>
<td>sorted-find</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>17</td>
<td>5.1</td>
</tr>
<tr>
<td>sorted-insert</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>21</td>
<td>2.0</td>
</tr>
<tr>
<td>devres</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>rm_pkey</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0.7</td>
</tr>
</tbody>
</table>
CONCLUSIONS

Build reliable apps

ExpressOS: android platform
[ASPLOS’13]

Build tools

Design spec logics

Logic
Proof tactics
Heuristics
Machine Learning
Automata theory

Automate loop inv annotations

Automate proofs

VCC-Dryad
InvLearn

ICE Learning
(ML/ALT)
[CAV13,SAS13]

Dryad
sep logic

Natural proofs
[POPL12,
PLDI13]
FUTURE WORK

• **Build/prove more systems secure/correct.**
  - Can we prove large non-systems software correct?
  - Like say a large content-management system like Drupal?

• **Killer application**
  - Armed with all this new tech, can we verify a new class of applications?

• **End-to-end safety security**
  - So far we proved only specs written at the level of code
  - How can we prove a property like “The memory of two apps can never overlap”
  - B / Z: refinement of spec to code specs needed
  - Can we lift all the frameworks to higher-level specs?