A Path towards a Common Binary Analysis IR

Jeff Hollingsworth

Why A Binary Program IR

- Useful for many types of analyses
  - Identification of functions
  - Control flow graphs
  - Slicing
  - Information flow
- Sharing
  - Low level parts are tedious
  - Many uses of higher analyses (CFG, Slicing, etc.)
  - Use previous analyses to perform others

Approach

- Start from a machine impendent instruction abstraction
  - Provides basis for platform independent analyses
- Generic Annotation Framework
  - Way to store results of analyses
  - Allows use by other analyses
- Serialization Framework
  - Share results with other tools
  - Ruse expensive analyses in different runs

Annotation Framework

- Many analyses generate data while examining instructions/functions etc.
  - Generally costly operations
    - Store the result !
- Dyninst Tradition:
  - New analysis means add variable(s) classes
  - Error prone
  - API changes
  - Requires rebuild

Annotation Framework

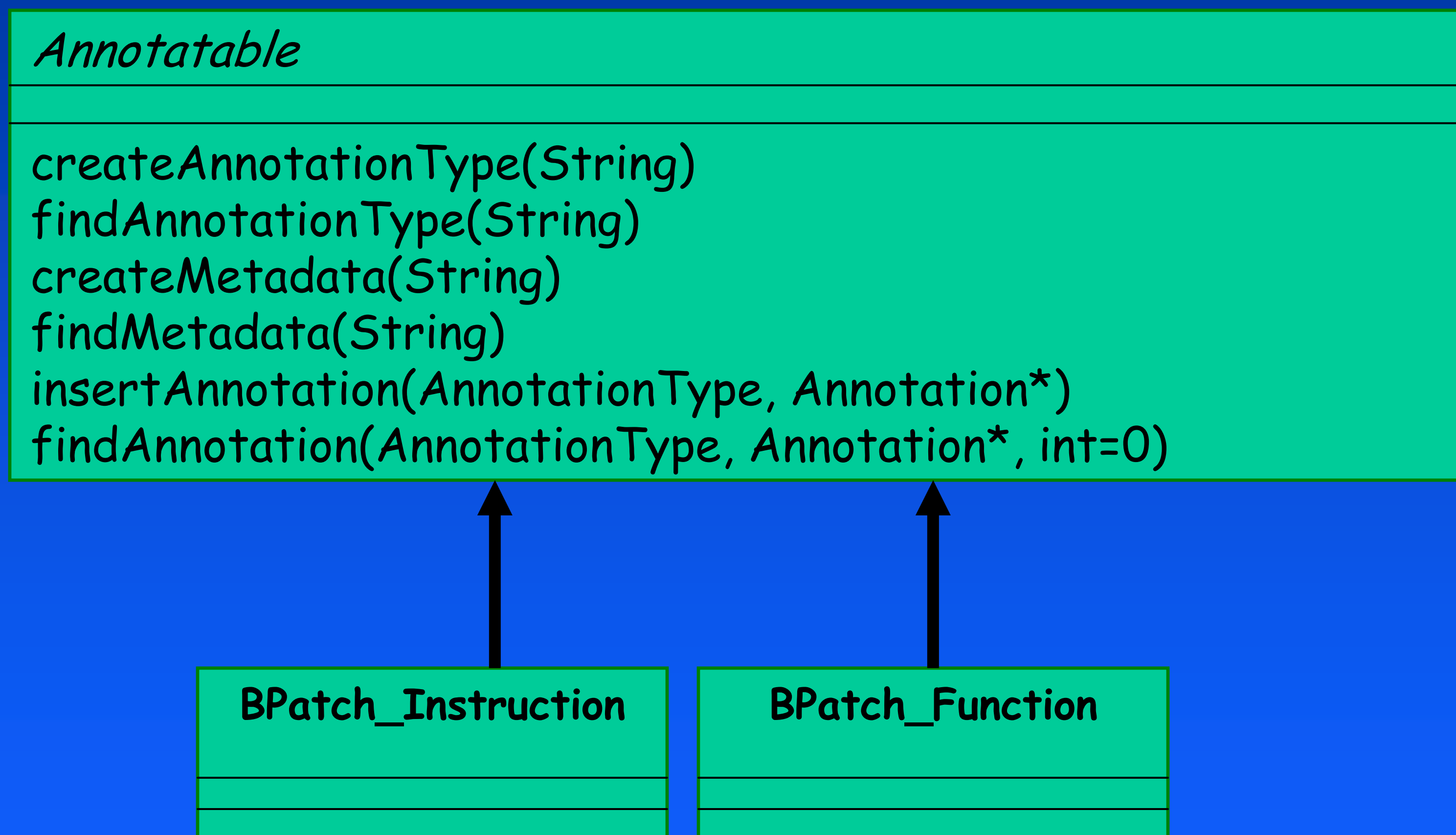
- Create a unified Annotation Framework instead
- Use a well-defined interface for each object that needs to be annotated
- Has to be extensible
  - Add new annotation types at runtime
- Support for storing metadata along with data
  - Confidence metrics
  - Pedigree data

Annotation Framework Example

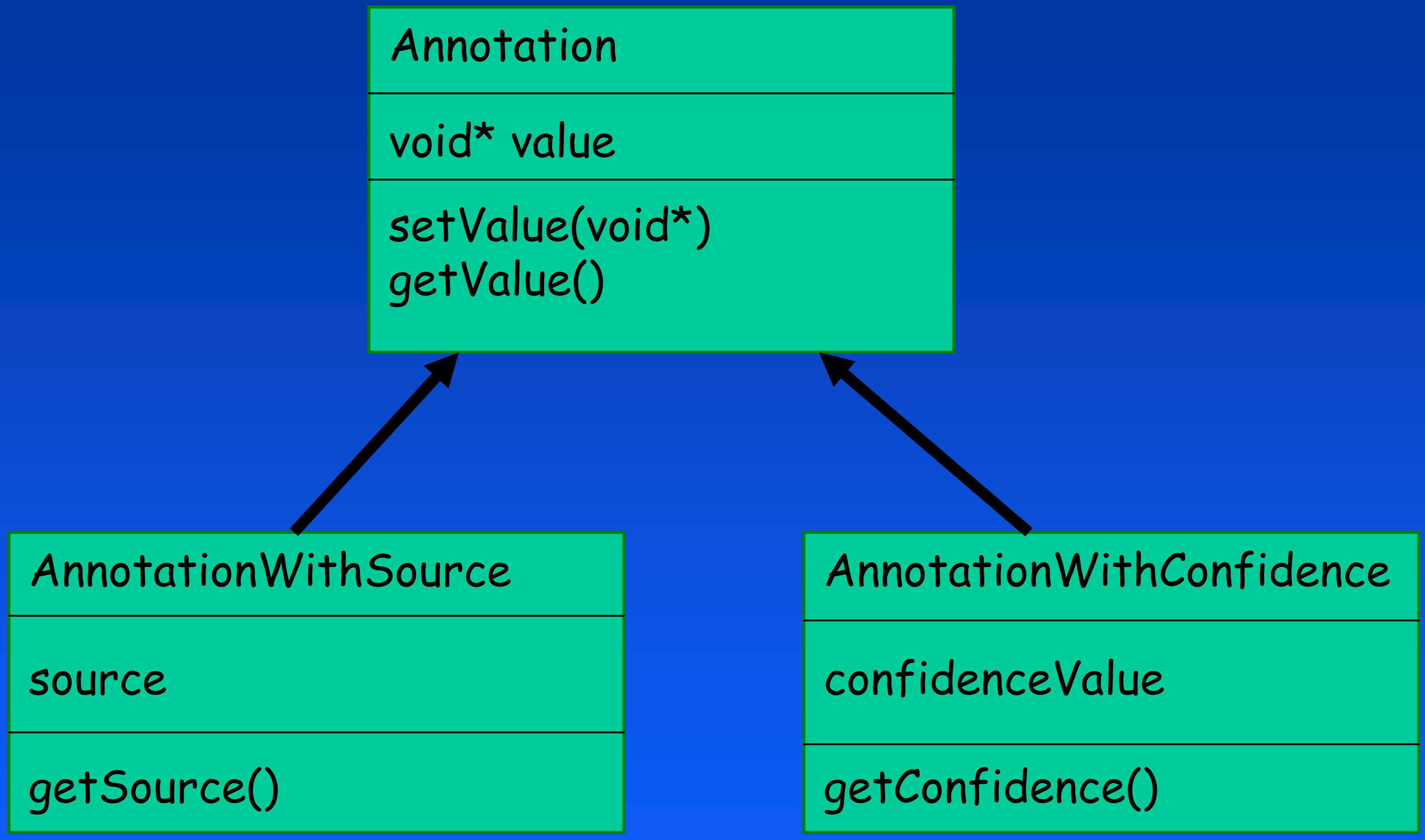
BPatch_instruction	BPatch_function
Register readSet[] Register writeSet[]	Graph* CFG Graph* dataDependenceGraph Graph* controlDependenceGraph Graph* programDependenceGraph Graph* slicingGraph

- Requires development effort
- Not desirable
  - Error-prone
  - Tedious

Annotation Framework



Annotation Framework



Serialization

- Two Formats:
  - Xml – “portable” for sharing information
  - Binary – faster for reloading
- Binary serialization should be transparent
  - User-controlled on/off switch: Env. Var.
  - Granularity:
    - One binary cache file per library / executable
    - Per logical sub-library of Dyninst
  - Checksum-based cache invalidation
    - Rebuild a binary’s cache when it changes
  - Example: libc is large and expensive to fully analyze, but it seldom changes

Serialization policy

- Two phase strategy
  - (1) Bulk serialization of always required internal state
    - Straightforward structured I/O
  - (2) Incremental serialization of incremental state
    - Somewhat trickier
    - No specific orderings allowed

Review: Why XML Serialization?

- Create standardized representations for
  - Basic symbol table information
  - Abstract program objects
    - Functions, loops, blocks….
  - More complex binary analyses
    - CFG, Data Slicing, etc…
- Exports Dyninst's expertise for easy use by
  - Other tools
  - Interfacing the textual world
    - Parse-able snapshots of programs
  - Cross-platform aggregation of results
- Allows Dyninst to use output from other tools in its own analyses

Why Binary Serialization?

- Large Binaries
  - Some existing Dyninst analyses taking a prohibitively long time for large binaries (100s of MB)
    - Eg. Full CFG analysis of large statically programs
- More complex analyses are in the works
  - Dyninst continues to add more complex features
    - Control Flow Graphs
    - Data Slicing
    - Stripped binary analysis
  - Complex tools that use these analyses may find them cost-prohibitive
    - If they have to be re-performed every time the tool is run
    - Why not just save them?

Speedup from Bulk Structured I/O

- Results for symtabAPI

# Symbols	Regular Parse Time	Serialize Time	Deserialize Time	Parse Speedup
2×10^3	68 ms	24 ms	26 ms	2.6x
2×10^4	730	148	210	3.4x
2×10^5	8900	1950	2300	3.9x

- Not exactly a "real world" problem
  - Verified scaling under a controlled situation
  - Computer-generated programs
    - with identical characteristics
    - except # symbols
  - Expect greater time savings with more complex analyses

On-Demand Analyses

- Dyninst generates much of its internal state on-demand of API user
  - Phase 1 serialization better suited to a known, fixed set of internal state
    - existing by-default
    - Still useful, but needs augmentation
- "Structural" solution to on-demand data
  - Ideally want an "automatic" solution
    - Do an analysis, then…
    - Serialization should happen transparently
- Uses Annotation framework
  - Reepresenting "optional" data
  - Perfect fit for the representation of on-demand analyses

Serializing Annotations

- Basic Parameters
  - Not all Annotations will be serialized
    - Does not make sense for all cases
  - parameters controls serialization policy
- Serialization is structural
  - Performed when annotation is added
  - Serialization parameters for annotation:
    - Just enough information to reconstruct
      - Annotatee ID
        - "this" Pointer suffices
      - Annotation Name
        - Annotation Type is determined by Name

Dyn inst

Example: Serialize Line Information

```
class Module : public
Annotatable<LineInformation,
         "line_info", true>
```

```
class LineInfo {
  vector<tuple>
};
```

Line Information:

- Part of SymtabAPI
    - Belongs to class Module
- Exists only on-demand

Example: Serialize Line Information

```
class Module : public
Annotatable<LineInformation,
         “line_info”, true>
```

```
class LineInfo {
  vector<tuple>
};
```

addAnnotation(LineInfo *)

- Marks entry in static annotation map

Example: Serialize Line Information

```
class Module : public
Annotatable<LineInformation,
          "line_info", true>
```

```
class LineInfo {
  vector<tuple>
};
```

Translator *toBin*

- append (f.bin)
- Start_annotation(f)
- Out_val(an_type)
- Out_val(par_id)

f.bin

<Annotation>
<AnnoType> an_type </AnnoType>
<Annotatee ID> par_id </Annotatee ID>

anno->serialize(LineInfo *)

- First output Annotation Information
- Just enough for full reconstruction
    - Annotation Type
    - ID of Parent

Example: Serialize Line Information

```
class Module : public
Annotatable<LineInformation,
        "line_info", true>
```

```
class LineInfo {
  vector<tuple>
};
```

Translator *toBin*

- append (f.bin)
- Start_annotation(f)
- Out_val(an_type)
- Out_val(par_id)
- Out (line_info)
  - Foreach (tuple)
  - out (tuple)

f.bin

<Annotation>
<AnnoType> an_type
</AnnoType>
<Annotatee ID> par_id
</Annotatee ID>
<LineInformation>
<num_entries> num
</num_entries>
<Tuple>
<file> f1 </file>
<line> ln </line>
<offset> off </offset>
</Tuple>
⋮ ⋮ ⋮
<Tuple>
</Tuple>
</LineInformation>
</Annotation>

anno->serialize(LineInfo *)

- Finally Translate LineInformation
- Using ordinary hierarchical I/O translation routine

Dyninst

Deserializing Annotations

- Basic Parameters
  - Need to construct new object given:
    - Annotatee ID
      - Build a working map between serialized Annotatee IDs and rebuilt Annotatable Objects
    - Annotation Type
      - Maintain static map between Annotation Type and deserialization function
- Deserialization sequence
  - Read Annotation Type
  - Read Annotatee ID
  - Lookup/call constructor for Annotation Type
  - Deserialize Annotation Object
  - Lookup Annotatee and re-annotate

Summary

- Annotation Framework
  - Status: Designed, at implementation stage
  - Unifies the way objects are annotated
  - Slicing will be the first user
- Annotations provide a natural way to serialize
  - External API provides users a way to attach arbitrary information to Dyninst class instances
  - Other uses still pending
    - Still flexible until other uses are resolved