Recovery of Class Hierarchies and Composition Relationships from Machine Code[⋆]

Venkatesh Srinivasan[1] and Thomas Reps[1,2]

[1] University of Wisconsin, Madison, WI, USA
[2] GrammaTech, Inc., Ithaca, NY, USA

Abstract. We present a reverse-engineering tool, called Lego, which recovers class hierarchies and composition relationships from stripped binaries. Lego takes a stripped binary as input, and uses information obtained from dynamic analysis to (i) group the functions in the binary into classes, and (ii) identify inheritance and composition relationships between the inferred classes. The software artifacts recovered by Lego can be subsequently used to understand the object-oriented design of software systems that lack documentation and source code, e.g., to enable interoperability. Our experiments show that the class hierarchies recovered by Lego have a high degree of agreement—measured in terms of precision and recall—with the hierarchy defined in the source code.

1 Introduction

Reverse engineering of software binaries is an activity that has gotten an increasing amount of attention from the academic community in the last decade (e.g., see the references in [2, §1]). However, most of this work has had the goal of recovering information to make up for missing symbol-table/debugging information [1,18,24,16,6,10], to create other basic intermediate representations (IRs) similar to the standard IRs that a compiler would produce [2,3,22], or to recover higher-level protocol abstractions or file formats [5,17,9].

In this paper, we address a problem that is complementary to prior work on reverse engineering of machine code,[1] namely, the problem of *recovery of class structure* at the machine-code level. In particular, we present a technique

[⋆] Supported, in part, by NSF under grants CCF- {0810053, 0904371}; by ONR under grants N00014- {09-1-0510, 11-C-0447}; by ARL under grant W911NF-09-1-0413; by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

[1] We use the term "machine code" to refer generically to low-level code, and do not distinguish between actual machine-code bits/bytes and assembly code to which it is disassembled.

to group a program's procedures into classes, and to identify inheritance and composition relationships between classes.

Class hierarchies and composition relationships recovered from machine code can be used to understand the object-oriented design of legacy software binaries while porting them to newer platforms. They can also be used while designing new software that is aimed to be interoperable with existing software binaries. For instance, in the United States, the Digital Millennium Copyright Act (DMCA) prohibits users from circumventing access-control technologies [8]. However, the DMCA specifically grants a small number of exceptions, including one for reverse engineering for the purpose of interoperability (§1201(f)). Others [6] have used similar artifacts as fingerprints of code polymorphic viruses for malware detection.

We present a tool, called Lego, which takes a stripped executable as input and uses dynamic analysis to recover the class structure of the program, including inheritance and composition relationships. Lego is based on two common features of object-oriented languages. The first is the *this-pointer idiom*: at the machine-code level, the object pointer is passed as an explicit first argument to a class's methods. Lego exploits this idiom to group calls to instance methods (methods that have the *this*-pointer as an explicit first argument), including dynamically dispatched ones, that have a common receiver object. The second idiom is the presence of a *unique finalizer method* in most class declarations, which is called at the end of an object's lifetime to do cleanup. Lego exploits this idiom, along with the aforementioned method-call groupings, to group methods into classes, and to recover inheritance and composition relationships between recovered classes.

We tested Lego on ten open-source applications. Using the class structure declared in the source code as ground truth, the classes recovered by Lego had an average precision of 88% and an average recall of 86.7%.

The contributions of our work include the following:

– We show that even if an executable is stripped of symbol-table and debugging information, and, even if run-time-type information (RTTI) is not present in the executable, it is still possible to reconstruct a class hierarchy, including inheritance and composition relationships, with fairly high accuracy. Our technique is based on common semantic features of object-oriented languages, and is not tied to a specific language, compiler, or executable format. It can be used on any binary generated from a language that uses the *this*-pointer and the unique-finalizer features, and a compiler that faithfully implements those features.
– Our methods have been implemented in a tool, called Lego, that uses dynamic analysis to recover a class hierarchy. (Because Lego uses dynamic analysis, it can recover classes only for the parts of the program that are exercised during execution.)
– We present a scoring scheme that takes the structure of class hierarchies into account while scoring a recovered hierarchy with respect to a ground-truth hierarchy.

```
class Vehicle {
 public:
    Vehicle();
    ~Vehicle();
    void print_vehicle();
};

class GPS {
 public:
    GPS();
    ~GPS();
};

class Car : public Vehicle {
    public:
        Car();
        Car(int n);
        ~Car();
        void print_car();
    private:
        GPS g;
};

class Bus : public Vehicle {
    public:
        Bus();
        ~Bus();
        void print_bus();
    private:
        void helper();
};

void foo(bool flag) {
  if (flag) {
      Car c;
      c.print_car();
  } else {
      Car c(10);
      c.print_car();
  }
}

int main() {
  Vehicle v;
  Bus b;
  v.print_vehicle();
  foo(true);
  foo(false);
  b.print_bus();
  return 0;
}
```

Fig. 1. C++ program fragment, with inheritance and composition

– Lego is immune to certain compiler idiosyncrasies and optimization side-effects, such as reusing stack space for different objects in a given procedure activation-record.

2 Overview

Lego recovers class structure from binaries in two steps:

1. Lego executes the program binary, monitoring the execution to gather data about the various objects allocated during execution, the lifetime of those objects, and the methods invoked on those objects. Once the program terminates, Lego emits a set of object-traces (defined below) that summarizes the gathered data.
2. Lego uses the object-traces as evidence, and infers a class hierarchy and composition relationships that agree with the evidence.

This section presents an example to illustrate the approach.

In our study, all of the binaries analyzed by Lego come from source-code programs written in C++. Fig. 1 shows a C++ program fragment, consisting of four class definitions along with definitions of the methods `main` and `foo`. Classes `Vehicle`, `Car`, and `Bus` constitute an inheritance hierarchy with `Vehicle` being the base class, and `Car` and `Bus` being derived classes. There is a composition relationship between `Car` and `GPS`. (`Car` has a member of class `GPS`.) Assume that, in the class definition, `helper()` is called by `~Bus()`. Also assume that the complete version of the program shown in Fig. 1 is compiled and stripped to create a stripped binary.

Lego takes a stripped binary and a test input or inputs, and does dynamic binary instrumentation. When the execution of the binary under the test input terminates, Lego emits a set of object-traces, one object-trace for every unique object identified by Lego during the program execution. An *object-trace* of an object O is a sequence of method calls and returns that have O as the receiver object. Additionally, the set of methods directly called by each method in the

```
v_1:
 Vehicle() C
 Vehicle() R            c_1:                    c_2:                     b_1:
 print_vehicle() C       Car() C                 Car(int) C               Bus() C
 print_vehicle() R       Vehicle() C             Vehicle() C              Vehicle() C
 ~Vehicle() C            Vehicle() R             Vehicle() R              Vehicle() R
 ~Vehicle() R            Car() R                 Car(int) R               Bus() R
                             Vehicle()               Vehicle()                Vehicle()
g_1:                         GPS()                   GPS()                print_bus() C
 GPS() C                 print_car() C           print_car() C            print_bus() R
 GPS() R                 print_car() R           print_car() R            ~Bus() C
 ~GPS() C                ~Car() C                ~Car() C                 helper() C
 ~GPS() R                ~Vehicle() C            ~Vehicle() C             helper() R
                         ~Vehicle() R            ~Vehicle() R             ~Vehicle() C
g_2:                     ~Car() R                ~Car() R                 ~Vehicle() R
 GPS() C                     ~GPS()                  ~GPS()               ~Bus() R
 GPS() R                     ~Vehicle()              ~Vehicle()               helper()
 ~GPS() C                                                                     ~Vehicle()
 ~GPS() R
```

Fig. 2. Object-traces for the example program. The records in the return-only suffixes are underlined.

sequence is also available in the object-trace. Concretely, an object-trace for an object O is a sequence of object-trace records. Each object-trace record has the following form,

$$\langle \mathit{method}, C \mid R, \mathit{calledMethods} \rangle,$$

where *method* denotes a method that was called with O as the receiver. Because we are dealing with binaries, methods are represented by their effective addresses, and so *method* is an effective address. C denotes a call event for *method*; R denotes a return event. *calledMethods* denotes the set of effective addresses of methods directly called by *method*. Each method in *calledMethods* can have any receiver object (not necessarily O). Object-traces are the key structure used for recovering class hierarchies and composition relationships.

In the rest of this section, when we use the term "method" in the context of object-traces or recovered classes, we are referring to the effective address of the method. However, to make our examples easier to understand, we will use method names rather than method effective addresses.

Fig. 2 shows the set of object-traces obtained from executing our example binary with Lego. In the figure, the objects encountered by Lego are denoted by appending instance numbers to the source-code object names: `c_1` and `c_2` correspond to different objects in two different activations of method `foo`, and `g_1` and `g_2` correspond to the instances of the `GPS` class in those objects.

We now describe how Lego obtains the class hierarchy and composition relationships from the set of object-traces. Lego computes a *fingerprint* for each object-trace. The fingerprint is a string obtained by concatenating the methods that constitute a return-only suffix of the object-trace. For our example, the fingerprint for the object-trace of `v_1` is `~Vehicle()`, and for the object-trace of `c_1`, it is `~Vehicle() ~Car()`. The object-trace records that are underlined in Fig. 2 contribute to fingerprints. A fingerprint represents the methods that were involved in the cleanup of an object. A fingerprint's length indicates the

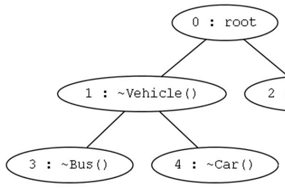


Fig. 3. Trie constructed by Lego using the object-trace fingerprints for the example program

Table 1. Methods in the set of recovered classes

Trie node	Methods in the recovered class
1	`Vehicle(), print_vehicle(), ∼Vehicle()`
2	`GPS(), ∼GPS()`
3	`Bus(), print_bus(), ∼Bus(), helper()`
4	`Car(), Car(int), print_car(), ∼Car()`

possible number of levels in the inheritance hierarchy from the object's class to the root. The methods in a fingerprint represent the potential finalizers in the class and its ancestor classes.

Next, Lego constructs a trie by inserting the fingerprints into an empty trie and creating a new trie node for each new method encountered. For the fingerprints of the object-traces in Fig. 2, the constructed trie is shown in Fig. 3. Each node's key is a finalizer method. Event order (i.e., left-to-right reading order in Fig. 2) corresponds to following a path down from the root of the trie (cf. Fig. 3).

Lego links each object-trace *ot* to the trie node N that "accepts" *ot*'s fingerprint. In particular, N's key is the last method in *ot*'s fingerprint. In our example, the object-trace of `v_1` is linked to node 1 of Fig. 3, the object-traces of `g_1` and `g_2` to node 2, the object-trace of `b_1` to node 3, and the object-traces of `c_1` and `c_2` to node 4.

Using the linked object-traces, Lego computes, for each trie node, the *methods set* and the *called-methods set*. For a trie node N and a set of object-traces OT_N linked to N, N's methods set is the set of methods that appear in some object-trace record in OT_N; N's called-methods set is the set union of the *calledMethods* field of the last object-trace record in each object-trace in OT_N. For instance, node 4's methods set is {`Car()`, `Car(int)`, `Vehicle()`, `print_car()`, `∼Car()`, `∼Vehicle()`}, and its called-methods set is {`∼GPS()`, `∼Vehicle()`}. If methods present in the methods set of ancestor nodes are also present in the methods set of descendants, Lego removes the common methods from the descendants. The resulting trie nodes and their methods sets constitute the recovered classes, and the resulting trie constitutes the recovered class hierarchy. The methods of each recovered class are shown in Table 1.

To determine composition relationships between recovered classes, for all pairs of trie nodes m and n, where neither is an ancestor of the other, Lego checks if n's key is present in the called-methods set of m. If so, the recovered class corresponding to m has a member whose class is the one corresponding to n, and thus there exists a composition relationship between m and n. For instance, in our example, the objects `c_1` and `c_2` (associated with node 4) both call `∼GPS()`, which is the key of node 2; consequently, Lego reports a composition relationship between nodes 4 and 2.

In this example, the recovered classes exactly match the class definitions from the source code. However, this example illustrates an idealized case, and for real applications an exact correspondence may not be obtained.

Threats to validity. There are five threats to the validity of our work.

1. The binaries given as input to Lego must come from a language that uses the *this*-pointer idiom.
2. Lego assumes that every class has a unique finalizer method that is called at the end of an object's lifetime. If a class has no finalizer or multiple finalizers, the information recovered by Lego might not be accurate. Lego also assumes that a parent-class finalizer is called only at the end of a child-class finalizer.

   In C++, the class destructor acts as the finalizer. Even if the programmer has not declared a destructor, in most cases, the compiler will generate one. A C++ base-class's destructor is called at the very end of a derived-class's destructor. The C++ compiler will sometimes create up to three versions of the class destructor in the binary [14]. Information that certain methods are alternative versions of a given destructor can be passed to Lego. However, our experiments show that there is little change in the results when such information is not provided to Lego (Fig. 9).
3. If the binary has stand-alone methods that do not belong to any class, but have an object pointer as the first argument, Lego might include those stand-alone methods in the set of methods of some recovered class. Although the recovered classes will not match the source-code class structure, it is arguable that they reflect the "actual" class structure used by the program.

   In addition, stand-alone methods that have a non-object pointer as the first argument may end up in stand-alone classes that are not part of any hierarchy.
4. Lego relies on the ability to observe a program's calls and returns. Ordinarily, these actions are implemented using specific instructions—e.g., `call` and `ret` in the Intel x86 instruction set. Code that is obfuscated—either because it is malicious, or to protect intellectual property—may deliberately perform calls and returns in non-standard ways.
5. Inlining of method calls also causes methods to be unobservable. In particular, if a method has been uniformly inlined by the compiler, it will never be observed by Lego.

For real software systems, these issues are typically not completely avoidable. Our experiments are based on C++, which uses the *this*-pointer idiom, and issues 4 and 5 were deemed out of scope. The experiments show that, even if issues 2 and 3 are present in an executable, Lego recovers classes and a class hierarchy that is reasonably accurate.

3 Algorithm

Lego needs to accomplish two tasks: (i) compute object-traces, and (ii) identify class hierarchies and composition relationships. In this section, we describe the algorithms used during these two phases of Lego.

Algorithm 1. Algorithm to compute full object-traces

```
Input: Currently executing instruction I
 1. if InstrFW.isCall(I) then
 2.     m ← InstrFW.eaOfCalledMethod(I)
 3.     ShadowStack.top().calledMethods.insert(m)
 4.     ID ← InstrFW.firstArgValue(I)
 5.     expectedRetAddr ← InstrFW.eaOfNextInstruction(I)
 6.     ShadowStack.push(⟨ID, expectedRetAddr, ∅⟩)
 7.     OTM[ID].append(m, C, ∅)
 8. else if InstrFW.isReturn(I) then
 9.     if not IgnoreReturn(I) then
10.         m ← InstrFW.eaOfReturningMethod(I)
11.         ⟨ID, expectedRetAddr, calledMethods⟩ ← ShadowStack.top()
12.         ShadowStack.pop()
13.         OTM[ID].append(m, R, calledMethods)
14.     end if
15. else
16.     // Do Nothing
17. end if
```

3.1 Phase 1: Computing Object-Traces

The input to Phase 1 is a stripped binary; the output is a set of object-traces. The goal of Phase 1 is to compute and emit an object-trace for every unique object allocated during the program execution. This ideal is difficult to achieve because Lego works with a stripped binary and a runtime environment that is devoid of object types. We start by presenting a naïve algorithm; we then present a few refinements to obtain the algorithm that is actually used in Lego. In the algorithms that follow, a data structure called the Object-Trace Map (OTM) is used to record object-traces. The OTM has the type: OTM:$ID \rightarrow ObjectTrace$, where *ID* is a unique identifier for a runtime object that Lego has identified.

3.1.1 Base Algorithm

A naïve first cut is to assume that every method in the binary belongs to some class, and to treat the first argument of every method as a valid *this* pointer (address of an allocated object). When Lego encounters a `call` instruction, it obtains the first argument's value, treats it as an *ID*, and creates an object-trace call-record for the called method. It then appends the record to *ID*'s object-trace in the OTM. (It creates a new object-trace if one does not already exist.) The highlighted lines of Alg. 1 show this strawman algorithm.

The algorithms of Phase 1 work in the context of a dynamic binary-instrumentation framework. They use the framework to answer queries (represented as calls to methods of an `InstrFW` object) about static properties of the binary ("Is this instruction a `call`?") and the dynamic execution state. ("What is the value of the first argument to the current call?") In this version of the algorithm, an *ID* is a machine integer. An *ID* for which there is an entry in the

Algorithm 2. Algorithm IgnoreReturn

```
Input: Instruction I
Output: true or false
 1. actualRetAddress ← InstrFW.targetRetAddr(I)
 2. ⟨firstArgValue, expectedRetAddr, calledMethods⟩ ← ShadowStack.top()
 3. if actualRetAddr $\neq$ expectedRetAddr then
 4.    if ShadowStack.matchingCallFound(actualRetAddr) then
 5.       ShadowStack.popUnmatchedFrames(actualRetAddr)
 6.    else
 7.       return true
 8.    end if
 9. end if
10. return false
```

OTM corresponds to the value of the first argument of some method called at runtime.

To enable the strawman algorithm to append an object-trace return-record for a method *m*, Lego must remember the value of *m*'s first argument to use as *ID* when it encounters *m*'s `return` instruction. To accomplish this, Lego uses a shadow stack. Each shadow-stack frame corresponds to a method *m*; a stack frame is a record with a single field, *firstArgValue*, which holds the value of *m*'s first argument. At a call to *m*, Lego pushes the value of *m*'s first argument on the shadow stack. At a return from *m*, Lego obtains the value at the top of the shadow stack, treats it as the *ID*, creates an object-trace return-record for *m*, and appends it to *ID*'s object-trace in the OTM. It then pops the shadow stack.

Due to optimizations, or obfuscations that use calls or returns as obfuscated jumps [21], some binaries may have calls with unmatched returns, and returns with unmatched calls. Unmatched calls and returns would make Lego's shadow stack inconsistent with the runtime stack, leading to incorrect object-traces. To address this issue, Lego does call-return matching. The actions taken are those of line 9 of Alg. 1, and Alg. 2.

To record the methods called by a method in an object-trace record, we add another field, *calledMethods*, to each shadow-stack frame. For a frame corresponding to method *m*, *calledMethods* is the set of methods that are directly called by *m* (dynamically). The basic algorithm that computes full object-traces along with call-return matching is shown in Alg. 1 (both the highlighted and non-highlighted lines). Note that the *calledMethods* set is empty for call-records.

3.1.2 Blacklisting Methods

Alg. 1 records the necessary details that we want in object-traces. However, because Alg. 1 assumes that all methods receive a valid *this* pointer as the first argument, stand-alone methods and static methods, such as the following would end up in object-traces:

```
void foo();
static void Car::setInventionYear(int a);
```

The algorithm actually used in Lego tries to prevent methods that do not receive a valid *this* pointer as their first argument from appearing in object-traces. Because inferring pointer types at runtime is not easy, when the instrumentation framework provides the first argument's value v for a method m, Lego checks whether v could be interpreted as a pointer to some allocated portion of the global data, heap, or stack. If so, Lego heuristically treats v as a pointer (i.e., it uses v as an object *ID*); if not, Lego *blacklists* m. Once m is blacklisted, it is not added to future object-traces; moreover, if m is present in already computed object-traces, it is removed from them.

The metadata maintained by Lego is only an estimate. For example, Lego keeps track of the stack bounds by querying the instrumentation framework for the value of the stack pointer at calls and returns. If the estimates are wrong, it is possible for a method that receives a valid *this* pointer to be blacklisted. If the estimates are correct, methods that receive a valid *this* pointer are unlikely to ever be blacklisted. In contrast, methods that do not receive a valid *this* pointer are likely to be blacklisted at some point, and thereby prevented from appearing in any object-trace. One final point is worth mentioning: methods that expect a valid pointer as their first argument, but not necessarily a valid *this* pointer, will not be blacklisted (threat 3 to the validity of our approach).

3.1.3 Object-Address Reuse

§3.1.2 presented a version of the algorithm to compute object-traces that, on a best-effort basis, filters out methods that do not receive a valid *this* pointer as the first argument. However, there are several possible ways for the methods of two unrelated classes to appear in the same object-trace. Consider the example shown in Fig. 4. Assuming standard compilation and runtime environments, `a` and `b` will be allocated at the same address on the stack (but in two different activation-record instances). As a consequence, `printA()` and `printB()` will end up in the same object-trace. Methods of unrelated classes can also end up in the same object-trace when the same heap address is reused for the allocation of different objects of different classes.

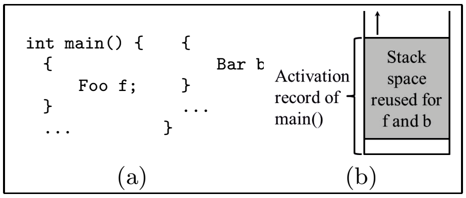
Lego detects reuse of the same object address by versioning addresses. When Lego treats the value v of a method's first argument as a valid *this* pointer, Lego associates a version number with v. If v is deallocated (i.e., if it is freed in the heap, or if the method in whose activation record v was allocated returns), Lego increments the version number for v. An *ID* now has the form $\langle Addr, n\rangle$, where *Addr* is the object address and n is the version number.

3.1.4 Spurious Traces

Even with address versioning, it is possible for methods of two unrelated classes to end up in the same object-trace. This grouping of unrelated methods in the same object-trace is caused by the idiosyncrasies of the compiler in reusing stack space for objects in the same activation record (as opposed to reusing stack space

```
class A {       void foo() {
   . . .            A a;        int main() {
   printA();        a.printA();     foo();
};              }                   bar();
class B {       void bar() {        return 0;
   . . .            B b;        }
   printB();        b.printB();
};              }
```

Fig. 4. Example program to illustrate reuse of stack space for objects in different activation records



```
int main() {     {
 {                    Bar b
     Foo f;      }
 }                    ...
 ...           }
```

Fig. 5. (a) Example to illustrate reuse of stack space for objects in the same activation record; (b) a stack snapshot

in different activation records, which §3.1.3 dealt with). We call such traces *spurious traces.* Consider the example program and its stack snapshot shown in Fig. 5. Because `f` and `b` are two stack-allocated objects in disjoint scopes, the compiler could use the same stack space for `f` and `b` (at different moments during execution). Note that object-address versioning does not solve this issue because an object going out of scope within the same activation record cannot be detected by a visible event (such as a method return or a heap-object deallocation).

To handle this issue, once the object-traces have been created by Alg. 1, Lego computes a set of potential initializers and finalizers by examining each object-trace *ot*. It adds the method of *ot*'s first entry to the set of potential initializers, and the method of *ot*'s last entry to the set of potential finalizers. It then scans each object-trace, and splits a trace at any point at which one of the potential finalizers is immediately followed by one of the potential initializers. This scheme breaks up spurious traces into correct object-traces. Note that if a class does not have an initializer or a finalizer, many methods of that class might end up in the set of potential initializers and the set of potential finalizers. As a consequence, non-spurious object-traces of objects of that class might be split. We examine the effects of splitting and not splitting spurious traces in our experiments (§4.4).

3.2 Phase 2: Computing Class Hierarchies

If the application does not use inheritance, the object-trace of an object will contain only the methods of the object's class. However, if the application uses inheritance, the object-trace of an object will contain methods of the object's class, plus those of the class's ancestors. In this section, we describe how Lego teases apart methods of different classes in a hierarchy. The input to this phase is a set of object-traces from Phase 1. The output is the recovered hierarchy.

3.2.1 Identifying Candidate Classes

A common semantics in object-oriented languages is that a derived class's finalizer cleans up the derived part of an object, and calls the base class's finalizer just before returning (to clean up the base part of the object). This behavior is visible in the object-traces that Lego gathers. Consider the example program and object-trace snippet of a `D` object shown in Fig. 6. The snippet covers all of the records between and including the last return record and its matching

```
class A {         class B:          class C:          class D:
   ~A();          public A {        public B {        public C {
};                    ~B();             ~C();             ~D();
                  };                };                };
```

(a)

```
~D() C
~C() C
~B() C
~A() C
```

$\underline{\texttt{\sim A()\ R}}$
$\underline{\texttt{\sim B()\ R}}$
$\underline{\texttt{\sim C()\ R}}$
$\underline{\texttt{\sim D()\ R}}$

(b)

Fig. 6. (a) Example program, and (b) object-trace snippet to illustrate an object-trace fingerprint (underlined returns)

Algorithm 3. Algorithm to populate candidate classes

```
Input: OTM, Trie T
Output: Trie T with candidate classes populated with methods
 1. for each object-trace ot in OTM do
 2.     lastRec ← ot.getLastRecord()
 3.     m ← lastRec.method
 4.     c ← T.getCandidateClassWithFinalizer(m)
 5.     c.calledMethods ← lastRec.calledMethods
 6.     for each object-trace record r in ot do
 7.         m′ ← r.method
 8.         c.methods.insert(m′)
 9.     end for
10. end for
```

call record. (The values of *calledMethods* fields of the object-trace records are omitted.)

We construct a string by concatenating the *method* fields that appear in the return-only suffix of an object-trace. We call such a string the *fingerprint* of the object-trace. We can learn two useful things from the fingerprint.

1. Because the fingerprint contains the methods involved in the cleanup of the object and its inherited parts, a fingerprint's length indicates the number of levels in the inheritance hierarchy from the object's class to the root.
2. The methods in the fingerprint correspond to potential finalizers in the class and its ancestor classes.

Lego computes a fingerprint for every computed object-trace, and creates a trie from the fingerprints (see §2). Every node in the trie corresponds to a *candidate class*, with the node's key constituting the candidate class's finalizer.

3.2.2 Populating Candidate Classes

Every computed object-trace *ot* is linked to the trie node (candidate class) that accepts *ot*'s fingerprint. Every candidate class has a *methods* set and a *called-methods* set. The methods set represents the set of methods in the object-traces linked to the candidate class, and is used in the computation of the final set of methods in each recovered class (see §3.2.3). The called-methods set represents the methods called by the finalizer of the candidate class, and is used to find composition relationships between recovered classes. The algorithm to populate the sets is given as Alg. 3.

Algorithm 4. Algorithm to find composition relationships

```
Input: Trie T
Output: Set of candidate class pairs $\langle A, B\rangle$ such that A has a member whose class is B
1. compositionPairs = $\emptyset$
2. for each pair of non-ancestors $\langle c, c'\rangle$ in T do
3.    if $c'$.finalizer $\in$ c.calledMethods then
4.       compositionPairs $\leftarrow$ compositionPairs $\cup$ $\langle c, c'\rangle$
5.    end if
6. end for
```

3.2.3 Trie Reorganizations

Some methods may appear both in the methods set of a candidate class C and candidate classes that are descendants of C. To remove this redundancy, Lego processes the candidate classes in the trie from the leaves to the root, and eliminates the redundant methods from the methods sets of candidate classes of descendants.

If two candidate classes C_1 and C_2, neither of which is an ancestor of the other, have a common method m in their methods sets, m is removed from the methods sets of C_1 and C_2, and put in the methods set of their lowest common ancestor. This reorganization handles cases where a class C was never instantiated during the program's execution, but its descendants C_1 and C_2 were, and the descendants had methods inherited from C in their object-traces.

After these two transformations, if a candidate class has no methods in its methods set, its trie node is removed from the trie. The resulting candidate classes and their corresponding methods sets constitute the final set of classes recovered by Lego. The final trie represents the recovered class hierarchy.

3.2.4 Composition Relationships

A composition relationship is said to exist between two classes A and B if A has a member whose class is B. The instance of the member is destroyed when the enclosing object is destroyed. However, unlike inheritance, A and B do not have an ancestor-descendant relationship. The algorithm for determining composition relationships is shown in Alg. 4.

Certain relationships between classes exist only at the source level. At the binary level, they become indistinguishable from other relationships. Lego cannot distinguish between certain composition relationships and inheritance. Consider the example shown in Fig. 7. Because the member `g` is the first member of a `Car` object, it might result in the `Car` object having the same object address as `g`. Methods of `g` end up in the object-trace of the `Car` object, and Lego would recover a hierarchy in which `GPS` becomes the base class of `Car`.

Because Lego operates at the binary level, Lego sees multiple inheritance as a combination of single inheritance and composition. Consider the example shown in Fig. 8(a). For the object layout shown in Fig. 8(b), Lego would recover

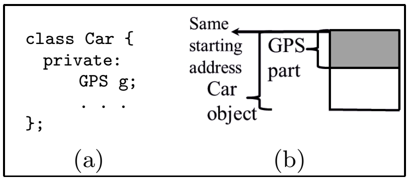


Fig. 7. (a) Example class-definition snippet; (b) a possible object layout to illustrate a composition relationship

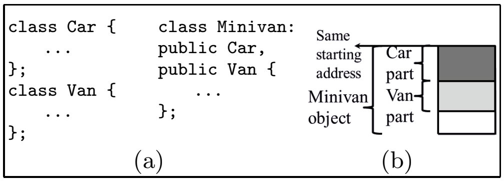


Fig. 8. (a) Example class-definition snippet; (b) a possible object layout to illustrate multiple inheritance

a class hierarchy in which `Car` is the base class, `Minivan` is derived from `Car`, and `Minivan` has a member whose class is `Van`.

4 Experiments

This section describes Lego's implementation, the scoring scheme used to score the conformance of Lego's output with ground-truth, and the experiments performed.

4.1 Implementation

Lego uses Pin [20] for dynamic binary instrumentation, and Phase 1 of Lego is written as a "Pintool". Pin can instrument binaries at the instruction, basic-block, routine, and image level. (Lego mainly uses instruction instrumentation for the algorithms of Phase 1; it uses image instrumentation for instrumenting routines for dynamic memory allocation and deallocation.) Pin executes the binary for each given test input, while performing Lego's Phase 1 instrumentation and analysis actions. Object-traces are computed and stored in memory, and emitted at the end of the execution of the program. A post-processing step of Phase 1 reads the object-traces, removes spurious traces, and emits the final set of object-traces. Phase 2 reads the final object-traces and emits four output files:

1. The set of recovered classes: each class is a set of methods; each class is uniquely identified by an *ID*.
2. The recovered class hierarchy: a trie with every node (except the root) having a class's *ID* as its key.
3. The recovered finalizers: a set of methods in which each method is identified as the finalizer of some class recovered by Lego.
4. The recovered composition relationships: a set of class *ID* pairs. Each pair $\langle A, B\rangle$ indicates that class A has a member whose class is B.

4.2 Ground Truth

We used C++ applications to test Lego. To score the outputs created by Lego, we collected ground-truth information for our test suite. For each application, the methods in each class and the set of destructors were obtained from the unstripped, demangled binary. The class hierarchy and composition relationships were obtained from source-code class declarations. We refer to this information as Unrestricted Ground Truth (UGT). We removed classes and methods of libraries that were not included in the source code (for example, the C++ standard library) from the UGT (even if they were statically linked to create the executable) because common library functions could potentially occupy the bulk of UGT for all our test applications, thereby skewing our scores.

We cannot use UGT to score Lego's outputs because it contains *all* the methods and classes in the program, whereas Lego's outputs contain only the subset of classes and methods that was *exercised* during Phase 1. We give the UGT files to Lego as an additional input—used only to prepare material for scoring purposes—and Lego emits "exercised" versions of the ground-truth files at the end of Phase 1. We refer to these files as Partially-Restricted Ground Truth (PRGT). Only methods that were exercised, and only classes that had at least one of their methods exercised, appear in the PRGT files. (For example, the destructors file now has only the set of exercised destructors, and the composition-relationships file contains only pairs $\langle A, B\rangle$ for which methods of A and methods of B were exercised.)

Lego tries to group only methods that receive a *this* pointer, and it expects every class in the binary to have a unique finalizer that should be called whenever an instance of the class is deallocated. However, PRGT does not comply with Lego's goals and restrictions. Some classes in PRGT might contain static methods, and some might not have a finalizer. (Even if they did, the finalizer might not have been exercised during Phase 1.) To see how Lego performs in the ideal case where the ground-truth complies with Lego's goals and restrictions, we create another set of ground-truth files called Restricted Ground Truth (RGT). RGT is a subset of PRGT: RGT is PRGT with all static methods removed, and all classes removed that lack a destructor, or whose destructors were not exercised during Phase 1. When Lego's results are scored against RGT, we are artificially suppressing threats 2 and 3 to the validity of our study. Note that the set of exercised destructors is the same for PRGT and RGT.

Scoring against RGT corresponds to the ideal case, whereas scoring against PRGT corresponds to the more realistic case that would be encountered in practice. We report Lego's results for both PRGT and RGT in §4.4.

4.3 Scoring

This section describes the algorithms used to score Lego's outputs against ground-truth files. In this section, when we say "ground-truth" we mean RGT or PRGT.

4.3.1 Scoring Finalizers

This output is the easiest to score because the ground-truth and Lego's output are both sets of methods. We merely compute the precision and recall of the recovered set of destructors against ground-truth.

4.3.2 Scoring the Class Hierarchy

It is not straightforward to score recovered classes because we are dealing with sets of sets of methods, which are related by inheritance relationships. We do not want to match ground-truth classes against recovered classes because a perfect matching may not always be possible. (For example, due to spurious traces, Lego may coalesce methods of two ground-truth classes into one recovered class.) Thus, as our general approach to scoring, we see if any of the recovered classes match a ground-truth class, both in terms of the set of methods, as well as its position in the hierarchy.

A naïve way to score would be as follows: Compare the set of methods in each ground-truth class against the set of methods in each recovered class to determine the maximum precision and maximum recall obtainable for each ground-truth class. Note that different recovered classes can contribute to maximum precision and maximum recall, respectively, for the ground-truth class. However, this simple approach treats classes as flat sets, and does not account for inheritance relationships between classes. As a consequence, the penalty for a recovered class having an extra method from an *unrelated* class will be the same as having an extra method from an *ancestor* class.

The scoring scheme used below addresses the inheritance issue. For every class in the ground-truth hierarchy and in the recovered hierarchy (except the dummy root nodes), we compute the *extended-methods set*. The extended-methods set of a class is the set union of its methods and the methods of all of its ancestors. For every ground-truth class, we compare the extended-methods set against every recovered class's extended-methods set to determine a maximum precision and maximum recall for the ground-truth class. This scoring scheme incorporates inheritance into scoring, by scoring with respect to paths of the inheritance hierarchy, rather than with respect to nodes. For every unique path in the inheritance hierarchy, it measures how close are the paths in the recovered hierarchy.

Scoring could also be done in the converse sense—comparing the extended-methods set of each recovered class with the extended-methods sets of all ground-truth classes—to determine a maximum precision and maximum recall for each recovered class. However, recovered classes may contain classes and methods not present in ground-truth (for example, library methods). For this reason, we do not score in this converse sense.

We can also view our scoring problem as one of computing an appropriate similarity measure. For this task we make use of the Jaccard Index. The Jaccard Index for a pair of sets A and B is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

For every ground-truth class, we compare the extended-methods set against every recovered class's extended-methods set to determine the recovered class with the maximum Jaccard Index for the ground-truth class. In contrast, when computing maximum precision and maximum recall for a ground-truth class, the respective maxima might be associated with the extended-methods set of two independent recovered classes.

To obtain the precision, recall, and Jaccard Index for the entire ground-truth hierarchy, we compute the weighted average of, respectively, the maximum precision, maximum recall, and maximum Jaccard Index computed for each ground-truth class, using the number of methods in each ground-truth class as its weight. We compute a weighted average because we want classes with a larger number of methods to contribute more to the overall score than classes with a smaller number of methods.

4.3.3 Scoring Composition Relationships

For each ground-truth composition pair and each recovered composition pair, we compute the *composed-methods* set. The composed-methods set of a pair of classes is the set union of the methods of the two classes. We compare the composed-methods set of each ground-truth composition pair against the composed-methods sets of recovered composition pairs to determine the maximum precision, maximum recall, and maximum Jaccard Index. (We compute the Jaccard Index for scoring composition pairs as well because two different recovered composition pairs might contribute to maximum precision and maximum recall, respectively, for one ground-truth composition pair.) Finally, we compute the weighted-average precision, recall, and Jaccard Index for all ground-truth composition pairs, using the size of the composed-methods set of each pair as its weight.

4.4 Results

We tested Lego on ten open-source C++ applications obtained from SourceForge [25], the GNU software repository [13] and FreeCode [12]. The characteristics of the applications are listed in Table 2. The applications were compiled using the GNU C++ compiler. The test suite that came with the applications was used to create test inputs for the binary for Phase 1. The experiments were run on a system with a dual-core, 2.66GHz Intel Core i7 processor; however, all the applications in our test suite and all the analysis routines in Lego are single-threaded. The system has 4 GB of memory, and runs Ubuntu 10.04.

Our experiments had three independent variables:

1. Partially-restricted ground-truth (PRGT) vs. restricted ground-truth (RGT): See §4.2.
2. Destructor versions provided (Destr) vs. destructor versions not provided (NoDestr): Recall that some compilers produce up to three versions of a single declared destructor. In one set of experiments, for each destructor D we supplied all compiler-generated versions of D as additional inputs to Phase 1. This information was used to compute object-traces as if each class had a unique destructor in the binary. In another set of experiments, we did not coalesce the different destructor versions, and generated object-traces based on multiple destructors per class.
3. Split spurious traces (SST) vs. do not split spurious traces (NoSST): We described the additional pass to remove spurious traces from the object-traces emitted at the end of Phase 1 in §3.1.4. In one set of experiments

Table 2. Characteristics of our test suite. The applications are sorted by increasing method coverage.

Software	KLOC	No. of classes in program	No. of methods in program	No. of classes with multiple destructor versions	No. of classes in PRGT	No. of methods in PRGT (Method coverage)	No. of methods in PRGT belonging to classes with unexercised destructors	No. of classes in RGT	No. of methods in RGT
TinyXML - XML Parser	5	16	302	13	16	236 (78.14%)	19	13	203
Astyle - source-code beautifier	10.5	19	350	14	12	195 (55.71%)	3	10	192
gperf - perfect hash function generator	5.5	25	207	16	20	109 (52.65%)	37	13	72
cppcheck - C/C++ static code analyzer	121	77	1354	46	62	657 (48.52%)	31	54	567
re2c - scanner generator	7.5	36	257	29	32	119 (46.30%)	54	16	57
lshw - hardware lister	18.5	13	161	4	6	61 (37.88%)	2	4	59
smartctl - SMART disk analyzer	50.5	34	192	30	18	36 (18.75%)	16	8	19
pdftohtml - pdf to html converter	52.5	131	1693	126	57	314 (18.54%)	37	50	267
lzip - LZMA compressor	3.2	12	74	0	6	11 (14.86%)	7	2	4
p7zip - file archiver	122	372	2461	216	105	365 (14.83%)	38	74	327

(SST), we executed this pass and used the resulting object-traces for Phase 2. In another set of experiments (NoSST), we did not execute this pass.

The first set of experiments measured the conformance of the *recovered class hierarchy with the ground-truth hierarchy.* Fig. 9 shows the weighted-average precision, recall, and Jaccard Index obtained for different combinations of independent variables. The applications in the figure are sorted by increasing method coverage.

The aggregate precision, aggregate recall, and aggregate Jaccard Index reported for the entire test suite is the weighted average of the reported numbers, with the number of methods in the corresponding ground-truth as the weight. (The number of methods in PRGT is used as the weight in computing PRGT aggregates, and the number of methods in RGT is used as the weight in computing RGT aggregates.) One observation is that there is only a slight variation in precision, recall, and Jaccard Index in the Destr vs. NoDestr case. This tells us that the destructor versions are not essential inputs to recover accurate class hierarchies. Also, we can see that there is very little difference between precision, recall, and Jaccard Index numbers for the RGT vs. PRGT case. This tells us that even if we do not know if the binary came from clean object-oriented source-code, Lego's output can generally be trusted.

Another observation is that for some applications like TinyXML, cppcheck, etc., comparing against PRGT causes an increase in precision numbers compared to RGT (which seems counter-intuitive). This increase in precision is because of the fact that the recovered classes corresponding to the extra classes present in PRGT (and absent in RGT) get fragmented, with each fragment containing very

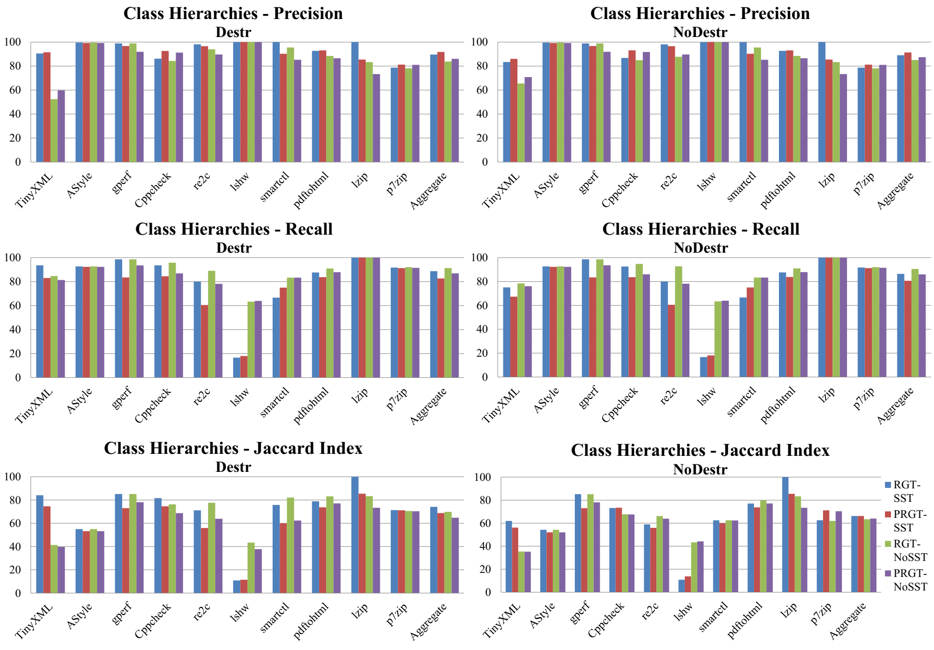


Fig. 9. Weighted-average precision, recall, and Jaccard Index for recovered class hierarchies

few methods of the class and they are not mixed with other recovered classes' methods. Because we compute weighted-average precision, this fragmentation causes an increase over the RGT weighted-average precision. However, if the methods of the extra classes get mixed with other recovered classes' methods, we see the intuitive decrease in weighted-average precision for PRGT (cf. lzip)

With SST, the precision increases or stays the same compared with NoSST. The increase is more pronounced if the source-code heavily uses code blocks within the same method—for example, TinyXML—à la Fig. 9. The recall for the SST case is better only if destructor versions were provided and if the source-code heavily uses code blocks (TinyXML). If, say, by inspecting and testing the binary, we suspect that code blocks are used, we could ask Lego to run the split-spurious-traces pass before recovering classes.

The second set of experiments measured the conformance of *recovered composition relationships with ground-truth composition relationships*. Lego detects a composition relationship by looking for finalizers called from the enclosing class's finalizer. It makes the most sense to use only RGT as the ground truth while scoring recovered composition relationships because all classes in the composition pairs of RGT have their destructors exercised during Phase 1. (All classes in PRGT may not satisfy this property.) Fig. 10 shows the results. Note that applications that do not have any composition relationships between classes in RGT are not shown in the figure. One of the applications (TinyXML) had a

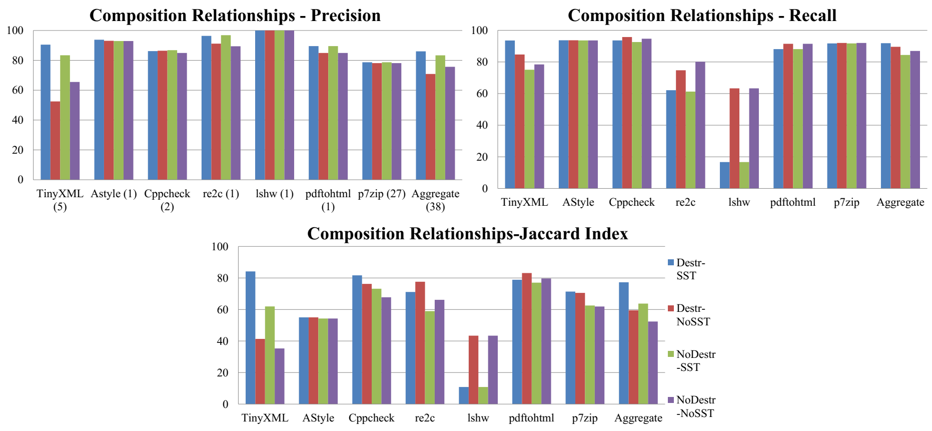


Fig. 10. Weighted-average precision, recall, and Jaccard Index for recovered composition relationships

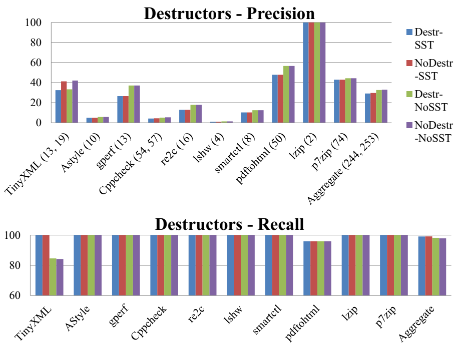


Fig. 11. Precision and recall for recovered destructors

Table 3. Time measurements (seconds). SD indicates slowdown.

Software	pin NULL	pinINSTR (SD)	I/O	Phase 2
tinyxml	3.62	30.54 (8.43x)	7.24	0.49
astyle	6.61	25.04 (3.78x)	2.74	0.88
gperf	2.05	6.51 (3.17x)	0.66	0.085
cppcheck	17.41	60.08 (3.45x)	5.14	1.04
re2c	3.31	7.44 (2.24x)	0.05	0.02
lshw	4.55	20.63 (4.53x)	0.46	0.33
smartctl	5.35	80.58 (15.06x)	1.78	0.15
pdftohtml	4.26	60.60 (14.22x)	15.22	6.40
p7zip	6.88	54.66 (7.94x)	9.73	0.04
lzip	1.47	3.70 (2.51x)	0.02	0.03

composition pair in RGT, in which the enclosing class's first member was a class. Because Lego sees this composition relationship as single inheritance (as described in §3.2.4), we removed this pair from the set of composition relationships (and added it as single inheritance in the RGT class hierarchy). The number of composition relationships between classes for each application is listed below its label in the precision graph. The aggregate precision, aggregate recall, and aggregate Jaccard Index for the entire test suite is the weighted average of the computed precision, recall, and Jaccard Index values with the sum of the sizes of all the composed-methods sets (§4.3.3) of an application as its weight.

The third set of experiments measured the conformance of *recovered destructors with ground-truth destructors*. Fig. 11 shows the results. Recall that RGT and PRGT have the same set of destructors (§4.2), so we report the results only for the RGT case. The number of destructors in each application is below its label in the precision graph. (Applications with different numbers of destructors

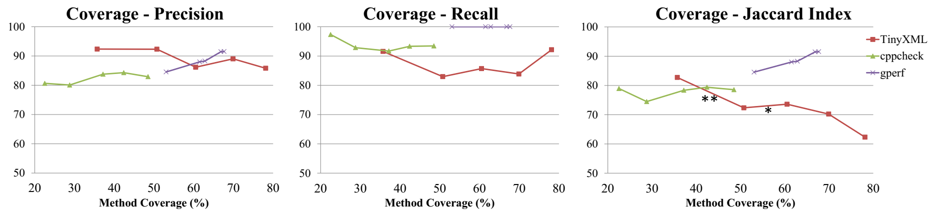


Fig. 12. Weighted-average precision, recall, and Jaccard Index for class hierarchies at different method coverages

for the Destr and NoDestr cases have both numbers listed.) The aggregate precision and recall for the entire test suite is the weighted average of the computed precision and recall values, with the number of destructors in ground-truth as the weight for each application. Lego identifies all of the destructors in most cases. In TinyXML, NoSST fails to expose a few destructors that are trapped in the middle of spurious object-traces. In pdftohtml, a few destructors get blacklisted by Lego and never end up in object-traces. Although Lego succeeds in identifying most of the destructors (high recall), the overall precision is low because destructors of classes in libraries—which are not present in the ground truth—are also reported by Lego.

Table 3 shows the timing measurements for our test suite. pinNULL represents the execution time of the application on pin, with Lego's analysis routines commented out. pinINSTR represents the execution time of the application on pin, with Lego's analysis routines performing dynamic analysis. The instrumentation and analysis overhead can be seen in the slowdown reported for each application. I/O represents the time taken to do file I/O in Phase 1 (reading ground-truth and destructor versions, writing object-traces, and exercised ground truth). pinINSTR + I/O represents the total running time of Phase 1 of Lego. Phase 2 reports the wall-clock time for Phase 2.

The fourth set of experiments aimed to study the *impact of code coverage* on the scoring metrics. For three applications (tinyxml, gperf, and cppcheck) we aggregated the object-traces from 5 test runs: just run 1; 1 and 2; 1 through 3; 1 through 4; and 1 through 5, resulting in five different amounts of method coverage for each application, and fed those object traces to Phase 2 of Lego. The combination of independent variables used in this set of experiments was Destr-RGT-NoSST. The results are shown in Fig. 12. We did not observe any global trends for precision, recall, or Jaccard Index with respect to increasing coverage. We observed that any of the following might happen when there is additional method coverage in Phase 1:

1. The additional coverage covers methods of a new class, thereby boosting the overall score for the test suite (see plots for gperf).
2. The additional coverage covers inherited methods of a class that is a sibling of a class already explored by Lego. This results in common inherited methods being hoisted to the parent class (see §3.2.3), thereby boosting similarity (see the line segment marked "*" in the Jaccard Index plot for TinyXML).

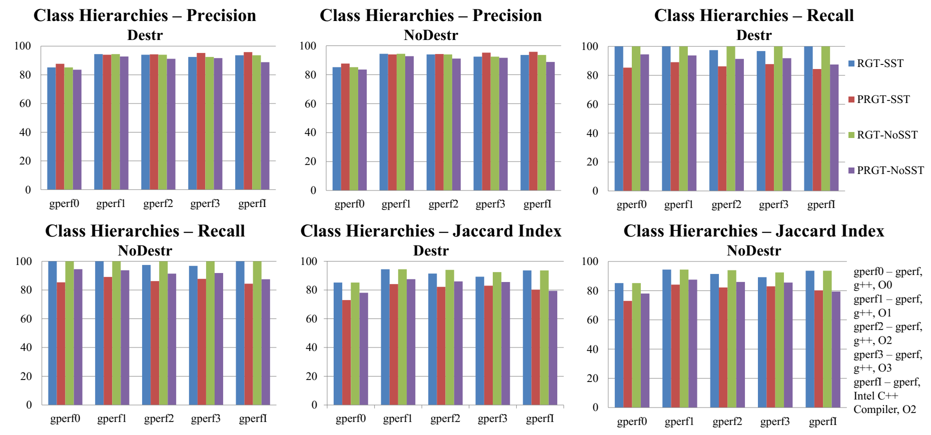


Fig. 13. Weighted-average precision, recall, and Jaccard Index for recovered class hierarchies measured for binaries generated from different compilers and at different optimization levels

3. The additional coverage causes Lego to encounter an object of a class A, but not objects of A's ancestors or siblings. This results in methods of A's ancestors ending up in the class recovered for A, thereby lowering the the similarity (see the line segment marked "**" in the Jaccard Index plot for TinyXML).

We also varied the compiler and optimization levels that were used to generate the binaries used in our experiments, and tested Lego on the newly generated binaries. Fig. 13 summarizes the results. We chose gperf as a representative application and compiled it using the Intel C++ compiler. We also compiled it using the GNU C++ compiler with different optimization levels. (We used the optimization flags -O0, -O1, -O2, and -O3.) The PRGT of the binaries generated using the -O1, -O2, and -O3 flags had 82 fewer methods than that of the binary generated using the -O0 flag. Methods being optimized away led to recovered classes that had fewer inherited base-class methods (which causes the slight increase in precision that one sees by comparing the gperf0 case to the other cases in Fig. 13).

To test Lego on binaries generated from a different object-oriented language, we collected applications written in the D programming language [7]. Lego did poorly in this experiment because D is garbage-collected, and thus many classes in the collected applications lacked a destructor. This failed experiment illustrates threat 2 to the validity of our approach, listed in §2.

5 Related Work

Reverse-Engineering Low-Level Software Artifacts from Binaries. Many prior works have explored the recovery of lower-level artifacts from binaries. Balakrishnan and Reps use a conjunction of Value Set Analysis (VSA) and Aggregate Structure Identification (ASI) to recover variable-like entities from stripped binaries [1]. Lee et al. describe the TIE system that recovers types from executables [16]. TIE uses VSA to recover variables, examines variable-usage patterns to generate type constraints, and then solves the constraints to infer a sound and most-precise type for each recovered variable. Dynamic analysis has also been used to reverse engineer data structures from binaries [24,18,6]. Such approaches can be used in conjunction with Lego to recover high-level types for recovered classes. Fields in recovered classes can either be of primitive type or user-defined type (composition or aggregation). While the tools and techniques described in the papers mentioned above can be used to recover primitive types, Lego can be used to recover composition relationships. (Recovering aggregation relationships is possible future work – see §6.)

Jacobson et al. describe the idea of using semantic descriptors to fingerprint system-call wrapper functions and label them meaningfully in stripped binaries [15]. Bardin et al. use Value Analysis with Precision Requirements (VAPR) for recovering a Control Flow Graph (CFG) from an unstructured program [3]. Schwartz et. al. describe the semantics-preserving structural-analysis algorithm used in Phoenix, their x86-to-C decompiler [22], to recover control structures. Fokin et al. describe techniques for decompilation of binaries generated from C++ [11]. During decompilation, they use run-time-type information (RTTI) and virtual function tables in conjunction with several analyses to recover polymorphic parts (virtual methods) of class hierarchies. The artifacts recovered by Lego complement those recovered by the aforementioned tools and techniques.

Recovering Protocol/File Formats from Executables. Prior works have also explored recovering higher-level abstractions from binaries. Cho et al. use concolic execution in conjunction with the L* learning algorithm to construct and refine the protocol state machine from executables that implement protocols [5]. Lim et al. describe recovering output file formats from x86 binaries using Hierarchical Finite State Machines (HFSMs) along with information from VSA and ASI [17]. Driscoll et al. use Finite Automata (FA) and Visibly Pushdown Automata (VPA) to infer I/O format of programs and check conformance of producer and consumer programs [9].

Modularizing Legacy Code. Formal Concept Analysis (FCA) has been extensively used for software-reengineering tasks [26,19,23]. Siff and Reps used FCA to modularize C code [23]. They used types and def/use information as attributes in a context relation to create a concept lattice, which was partitioned to obtain a set of concepts. Each concept was a maximal group of C functions that acted as a module. Bojic and Velasevic describe using dynamic analysis in conjunction with FCA to recover a high-level design for legacy object-oriented systems [4]. The high-level goals of Lego and these works are the same—namely, to recover a modular structure. However, Lego works at the binary level, where types are either absent or difficult to precisely obtain.

6 Conclusion and Future Work

In this paper, we described Lego, a tool that uses dynamic analysis to recover class hierarchies and composition relationships from stripped object-oriented binaries. We presented the algorithms used in Lego, and evaluated it on ten open-source C++ software applications by comparing the class hierarchies recovered by Lego with ground truth obtained from source code. Our experiments show that the class hierarchies recovered by Lego have a high degree of agreement—measured in terms of precision and recall—with the hierarchy defined in the source code. On average, the precision is 88% and the recall is 86.7%.

One possible direction for future work would be to use concolic execution to generate more inputs to achieve better coverage. For the Lego context, a concolic-execution engine should aim to maximize method coverage, not merely path coverage. A second direction would be to see how run-time-type information and virtual-function-table information could be used to improve the class hierarchy produced by Lego. When such information is available, it allows a portion of the source-code class hierarchy to be recovered exactly. (The hierarchy is incomplete because it contains only the program's virtual functions.) A third direction would be to use Lego's object-traces and recovered classes to infer temporal invariants on method-call order.

Another direction for future work is to use the information maintained by Lego about objects allocated during program execution to find aggregation relationships between inferred classes.

References

1. Balakrishnan, G., Reps, T.: DIVINE: DIscovering Variables IN Executables. In: VMCAI 2007. LNCS, vol. 4349, pp. 1–28. Springer, Heidelberg (2007)
2. Balakrishnan, G., Reps, T.: WYSINWYX: What You See Is Not What You eXecute. TOPLAS 32(6) (2010)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from unstructured programs. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 54–69. Springer, Heidelberg (2011)
4. Bojic, D., Velasevic, D.: A Use-case driven method of architecture recovery for program understanding and reuse reengineering. In: CSMR (2000)
5. Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: MACE: Model inference assisted concolic exploration for protocol and vulnerability discovery. In: USENIX Sec. Symp. (2011)
6. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: OSDI (2008)
7. D Programming Language, `http://dlang.org`
8. DMCA §1201. Circumvention of Copyright Protection Systems, `www.copyright.gov/title17/92chap12.html#1201`
9. Driscoll, E., Burton, A., Reps, T.: Checking compatibility of a producer and a consumer. In: FSE (2011)
10. ElWazeer, K., Anand, K., Kotha, A., Smithson, M., Barua, R.: Scalable variable and data type detection in a binary rewriter. In: PLDI (2013)
11. Fokin, A., Derevenetc, E., Chernov, A., Troshina, K.: SmartDec: Approaching C++ decompilation. In: WCRE (2011)
12. Freecode, `www.freecode.com`
13. GNU Software Repository, `www.gnu.org/software/software.html`

14. Itanium C++ ABI, `refspecs.linux-foundation.org/cxxabi-1.83.html`
15. Jacobson, E.R., Rosenblum, N., Miller, B.P.: Labeling library functions in stripped binaries. In: PASTE (2011)
16. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: NDSS (2011)
17. Lim, J., Reps, T., Liblit, B.: Extracting output formats from executables. In: WCRE (2006)
18. Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: NDSS (2010)
19. Lindig, C., Snelting, G.: Assessing modular structure of legacy code based on mathematical concept analysis. In: ICSE (1997)
20. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI (2005)
21. Roundy, K.A., Miller, B.P.: Binary-code obfuscations in prevalent packer tools. ACM Computing Surveys 46(1) (2013)
22. Schwartz, E.J., Lee, J., Woo, M., Brumley, D.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: USENIX Sec. Symp. (2013)
23. Siff, M., Reps, T.: Identifying modules via concept analysis. TSE 25(6) (1999)
24. Slowinska, A., Stancescu, T., Bos, H.: Howard: A Dynamic excavator for reverse engineering data structures. In: NDSS (2011)
25. SourceForge, `http://sourceforge.net`
26. Tonella, P.: Concept analysis for module restructuring. TSE 27(4) (2001)