An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer

Maria Christakis, Peter Müller, and Valentin Wüstholz

Department of Computer Science
ETH Zurich, Switzerland
{maria.christakis, peter.mueller, valentin.wuestholz}@inf.ethz.ch

Abstract. Many practical static analyzers are not completely sound by design. Their designers trade soundness in order to increase automation, improve performance, and reduce the number of false positives or the annotation overhead. However, the impact of such design decisions on the effectiveness of an analyzer is not well understood. In this paper, we report on the first systematic effort to document and evaluate the sources of unsoundness in a static analyzer. We present a code instrumentation that reflects the sources of deliberate unsoundness in the .NET static analyzer Clousot. We have instrumented code from several open source projects to evaluate how often concrete executions violate Clousot's unsound assumptions. In our experiments, this was the case in 8–29% of all analyzed methods. Our approach and findings can guide users of static analyzers in using them fruitfully, and designers in finding good trade-offs.

1 Introduction

Many practical static analyzers are not completely sound by design. Their designers often trade soundness in order to increase automation, improve performance, and reduce the number of false positives or the annotation overhead. As a result, such static analyzers become precise and efficient in detecting software bugs, but at the cost of making compromises, such as making implicit, unsound assumptions about certain program properties. For example, consider a sound static analyzer that determines all possible values of global variables. Such an analyzer may implement a sophisticated, inter-procedural, and potentially inefficient pointer analysis that over-approximates the values of these variables. On the other hand, an unsound analyzer may assume that the values of global variables can only change through direct assignments (and not through pointers), which simply requires a linear scan of the program [10]. Note that we use "compromise", "unsoundness", and "unsound assumption" interchangeably here.

Despite how common such design decisions are, their practical impact on the effectiveness of static analyzers is not well understood. There are various approaches in the literature that study the precision and efficiency of static analyzers by measuring, for instance, their performance and determining whether the number of false positives or lines of annotations is below a certain threshold [3]. In this paper, however, we investigate the compromise of soundness for

better precision and efficiency from a novel perspective, the perspective of the unsoundness in a static analyzer. In particular, we report on the first systematic effort to document and evaluate the sources of deliberate unsoundness in a static analyzer. We present a code instrumentation that reflects the sources of unsoundness in the static analyzer Clousot [9], a modular abstract interpretation tool for .NET and Code Contracts [8]. We subsequently use this instrumentation to evaluate how often concrete runs of several open source applications violate Clousot's unsound assumptions.

For our purposes, we adapt a technique we recently proposed for collaborative verification and testing [6] that aims at providing definite guarantees about program correctness by making compromises of static analyzers explicit. To evaluate the unsoundness in Clousot, we automatically insert annotations that make its assumptions explicit where they occur. We then attempt to evaluate whether these assumptions reflect good design decisions. We do so by running the test suites of open source projects and logging which of Clousot's unsound assumptions are violated. Note that it is not our intention in this work to determine whether bugs are missed by the tool. Instead, our goal is to evaluate Clousot's choice of compromises by discovering whether they can be empirically justified. The contributions of this paper are the following:

- We report on the first systematic effort to document all sources of unsoundness in an industrial-strength static analyzer. We focus on Clousot, a widely used, commercial static analyzer (at the time of this writing, more than 125K external downloads since February 2009).
- We present a code instrumentation that reflects the unsoundness in Clousot and discuss its precision. We over-approximate two sources of unsoundness, under-approximate one, and do not handle four. All other sources of unsoundness in Clousot are precisely captured by our encoding.
- We perform an experimental evaluation that, for the first time, sheds light on how often the unsound assumptions of a static analyzer are violated in practice. In our experiments, this was the case in 8–29% of all analyzed methods. Moreover, three sources of unsoundness were never violated in our evaluation.

We expect our results to guide users of static analyzers in using them fruitfully, for instance, in deciding how to complement static analysis with testing, and assist designers in finding good trade-offs. Our results can also facilitate collaboration of static analyzers; new analyzers can now focus on advanced features and rely on existing tools for those properties that are already handled in a sound way.

Outline. Sect. 2 explains all sources of unsoundness in Clousot and how we instrument most of them. Sect. 3 gives an overview of our implementation, and Sect. 4 presents our experimental results. We review related work in Sect. 5 and conclude in Sect. 6.

2 Unsoundness in Clousot

In this section, we present a complete list of Clousot's sources of deliberate unsoundness and demonstrate how most of these can be expressed through simple annotations. We have determined this list of Clousot's sources of unsoundness

by trying the tool on numerous examples, reading the publications on Clousot's analyses, and confirming its unsound assumptions with the designers of the tool.

We make the unsoundness of a static analyzer explicit in its output via annotations in the form of `assumed` statements, also called *explicit assumptions*. An `assumed` statement is of the form `assumed` P, where P is a boolean property, and denotes that a static analyzer *unsoundly* assumed property P at this point in the code, that is, assumed P without checking that it actually holds. Note that our `assumed` statements are different from the classical `assume` statements, which express properties that the user *intends* the static analyzer to take for granted.

In the rest of this section, we assume the program to be instrumented such that function `writtenObjects`() returns the set of objects that were modified by the most recent method call along the concrete execution, including any objects that were modified indirectly through method calls. We refer to this set of objects as the set of *written objects*.

We also assume a predicate `invariant`(o, t) that takes an object o and a type t, which is o's dynamic type or any of its super-types. This predicate returns true if and only if o satisfies the object invariant defined in t in conjunction with all invariants inherited from t's super-types.

We now present all sources of unsoundness in Clousot divided into four categories: those related to (1) the heap, (2) properties local to a method, (3) static state, and (4) those that we do not instrument.

2.1 Heap Unsoundness

We further categorize the sources of unsoundness in Clousot that are related to the heap into those that refer to (1) object invariants, (2) aliasing, (3) write effects, and (4) purity.

Object invariants. Object (or class) invariants express which instances of a class are considered valid. Object invariants can be specified in .NET programs using Code Contracts. A sound technique for checking object invariants never assumes an invariant to hold without justifying this assumption. More specifically, sound techniques need to check that the invariant indeed holds along all concrete executions reaching the point where the assumption is made.

To reduce the number of false positives, Clousot does not reason about object invariants in a sound way. In particular, *Clousot assumes that the invariant of the receiver object holds in the pre-state of instance methods without checking it at call sites*. Moreover, *Clousot assumes that, after a call to an inherited method on the current receiver (that is, on* `this`*), the invariant of the receiver holds without checking it.*

As an example of the first unsoundness, consider the C# code on the right, in which

```
class C {
  bool b;

  invariant !b;

  void M() {
    assumed invariant(this, typeof(C));
    b = true;
    N();
    assert !b;
  }

  void N() {
    assumed invariant(this, typeof(C));
    assert !b;
  }
}
```

method `M` violates the invariant of its receiver before calling `N`. (We use the special `invariant` and `assert` keywords to denote Code Contracts' object invariants and assertions.) The gray boxes in the code should be ignored for now. Clousot assumes the invariant of the receiver in the pre-state of method `N`, which is unsound since it does not check this invariant at call sites of `N`, in particular, at the call in `M`. Clousot also assumes the invariant of the receiver after the call to `N` in method `M`, but this assumption is sound because the invariant is checked in the post-state of `N` (`N` is not an inherited method). As a result, no warnings are emitted.

We capture this unsoundness by introducing an `assumed` statement at the beginning of each instance method in classes that declare or inherit object invariants. The explicit assumption states that the invariant of the method's receiver holds. Our `assumed` statements are shown in the gray boxes in the code. Note that this explicit assumption is expressed using the predicate $\texttt{invariant}(o, t)$ that we defined at the beginning of this section. Type t is, in this case, the type of the class in which the method is defined; the corresponding type object is retrieved with the `typeof` expression in C#. We label this kind of explicit assumptions as "invariants at method entries" (**IE**). We will refer to such labels in our experimental evaluation.

We precisely capture this unsoundness because (1) we introduce the `assumed` statements exactly where Clousot's assumptions occur, and (2) these assumptions are always unsound as the invariant of the receiver is never checked at call sites.

As an example of the second unsoundness, consider the code on the right. Method `M` of the sub-class calls the inherited method `N` of the super-class on the current receiver, and `N` violates the invariant declared in the sub-class. Clousot assumes the invariant of the current receiver after the call to `N` in `M`, which is unsound since it does not check this invariant in the post-state of `N` (in Clousot's modular analysis, `Sub`'s invariant is not considered when analyzing `Super`). As a result, no warnings are emitted.

```
class Super {
  bool b;

  void N() { b = true; }
}

class Sub : Super {
  invariant !b;

  void M() {
    N();
    assumed invariant(this, typeof(Sub));
    assert !b;
  }
}
```

We precisely capture this unsoundness by introducing an `assumed` statement after each call to an inherited method on the current receiver in classes that declare or inherit object invariants. The explicit assumption states that the object invariant of `this` holds for the enclosing class, here `Sub`. We label this kind of explicit assumptions as "invariants at call sites" (**IC**).

Aliasing. To avoid the overhead of a precise heap analysis, *Clousot uses an optimistic heap abstraction that ignores certain side-effects due to aliasing.* Namely, for operations that may modify certain objects, such as field updates and method calls, Clousot assumes that heap locations not explicitly aliased in the code are non-aliasing and will, thus, not be affected.

As an example of this unsoundness, consider method `M` on the right. (We use the special `assume` keyword to denote Code Contracts' assumptions, such as preconditions.) In this case, Clousot assumes that array `a` is not modified by the update to array `b`, although `a` and `b` might point to the same array in some calls to `M`. As a result, no warning is emitted about the assertion.

```
void M(int[] a, int[] b) {
  assumed a == null || !object.ReferenceEquals(a, b);
  assume a != null && b != null;
  assume 0 < a.Length && 0 < b.Length;
  a[0] = 0;
  b[0] = 1;
  assert a[0] == 0;
}
```

Clousot abstracts the heap by a graph, the *heap-graph*, which maintains equalities about access paths. More specifically, the nodes of the heap-graph denote symbolic values, which in turn represent concrete values, such as object references and primitive values. An edge of the heap-graph denotes how the symbolic value of the target node is retrieved from the symbolic value of the source node, for instance, by dereferencing a field or calling a pure method. (A method is called *pure* when it makes no visible state changes.) Therefore, all access paths in the heap-graph are rooted either in a local variable or a method parameter. When two access paths lead to the same symbolic value, they must represent the same concrete value along all executions, that is, must be aliases. However, when two access paths lead to distinct symbolic values, they may represent the same or different concrete values, that is, may or may not be aliases.

We conservatively introduce an `assumed` statement at the beginning of a method body (since Clousot assumes that method parameters are non-aliasing) and after every program statement that affects the aliasing information in the heap-graph, for instance, by creating, merging, or splitting nodes. For simplicity, we do not introduce `assumed` statements solely before operations that may have differing side-effects depending on aliasing. Consequently, we may introduce them at program points at which Clousot does not make any unsound assumption, for instance, when the effect of a statement on the heap-graph precisely reflects its concrete effect. Our explicit assumption has a conjunct for each pair of distinct symbolic values of aliasing-compatible reference types. Each conjunct states that the concrete values represented by the two distinct symbolic values (and given by the access paths leading to the symbolic values) are non-aliasing. In particular, the conjunct states that the concrete values are not equal unless they are null, as shown in method `M` above. Note that we use reference equality since the `==` operator may be overloaded in C#. We ensure that all explicit assumptions are *well-defined*, that is, insusceptible to runtime errors, such as null dereferences in access paths. We label this kind of explicit assumptions as "aliasing" (**A**).

Write effects. To avoid a non-modular, inter-procedural analysis or having to provide explicit write effect specifications, *Clousot uses unsound heuristics to determine the set of heap locations that are modified by a method call.* Clousot then assumes that all other heap locations are not modified by the method call. This assumption is unsound since Clousot does not check whether the heuristics actually determine all heap locations that are modified by the call.

```
class C {
  int[] a;

  void M() {
    var b = new int[1];
    a = b;
    N();
    assumed b == null || !writtenObjects().Contains(b));
    assert b[0] == 0;
  }

  void N() {
    if (a != null && 0 < a.Length) {
      a[0] = 1;
    }
  }
}
```

As an example of this unsoundness, consider the code on the right. Clousot assumes that the call to method `N` in `M` modifies only the current receiver (that is, `this`), and, thus, assumes that the elements of array `a` cannot be modified by the call. As a result, even though arrays `a` and `b` are explicitly aliased in the code (and in the heap-graph) before the call to method `N`, Clousot does not expect that, after the call, the elements of `b` have changed. No warning is, therefore, emitted about the assertion at the end of method `M`. Note that Clousot does expect that `a` and `b` might no longer be aliases since `N` is assumed to modify field `a` of the receiver.

Since Clousot reflects the write effects of method calls on the heap-graph, we capture this unsoundness by inspecting the graph before and after each call. More specifically, after a method call, we introduce an `assumed` statement stating that all heap locations in the graph that remained unmodified by the call are indeed not modified by the call. This is achieved by comparing all symbolic values in the heap-graph before and after each call and using their access paths to retrieve the concrete values they represent. The explicit assumption has a conjunct for each unmodified concrete object reference declaring that, when the reference is non-null, it is not contained in the actual write effect of the method for the last call (that is, in the method's set of written objects), as shown in method `M` above. Note that the assumption is expressed using the function `writtenObjects`() that we defined at the beginning of this section, which returns the set of objects that were modified by the last method call. We label this kind of explicit assumptions as "write effects" (**W**).

How precisely we capture this unsoundness depends on the definition of function `writtenObjects`(). If the function returns an over- or under-approximation of the set of heap locations actually modified by the last call, then our assumptions also over- or under-approximate Clousot's unsoundness. Otherwise, our assumptions precisely express that Clousot assumes the write effects to be correct without checking them. Note that, in our implementation, `writtenObjects`() precisely captures the set of modified heap locations except in the case of calls to uninstrumented (library) methods, as we discuss in Sect. 3.

Purity. Users may explicitly annotate a method with the Code Contracts' attribute `Pure` to denote that the method makes no visible state changes. To avoid the overhead of a purity analysis, *Clousot assumes that all methods annotated with the* `Pure` *attribute as well as all property getters are indeed pure.* (We will refer to property getters and methods annotated with `Pure` simply as "pure methods".) Moreover, Clousot uses unsound heuristics to determine which heap

locations affect the result of a pure method, the method's *read effect. Clousot then assumes that all pure methods deterministically return the same value when called in states that are equivalent with respect to their assumed read effects.*

We capture the first unsoundness with the explicit assumptions about write effects described above. After each call to a pure method, we introduce an `assumed` statement stating that all heap locations (in the heap-graph) remained unmodified. Here, we only discuss how we instrument the second unsoundness.

As an example of the second unsoundness, consider method `M` on the right. Clousot assumes that both calls to `Random` in `M` deterministically return the same value, and no warning is emitted about the assertion at the end of method `M`.

As another example of this unsoundness, consider method `N`. Clousot assumes that the result of method `FirstOrZero` depends only on the state of its receiver, but not the state of array `a`. Therefore, no warning is emitted about the assertion in `N` even though `a[0]` is modified after the call.

As we previously mentioned, Clousot's heap-graph maintains information about which values may be retrieved by calling a pure method. For instance, after the first call to method `Random` and `FirstOrZero` in `M` and `N`, respectively, the heap-graph maintains an equality of `r` to a call to `Random`, and of `v` to a call to `FirstOrZero`. We, therefore, capture this unsoundness by determining whether, at any program point, the heap-graph assumes a value previously returned by a pure method to still be retrievable via a call to this method. We introduce an `assumed` statement after every program statement for which this is the case. The explicit assumption has a conjunct for every such assumption in the heap-graph, as shown in the code above. We label this kind of explicit assumptions as "purity" (**P**).

```
class C {
  void M() {
    var r = Random();
    assumed r == Random();
    assert r == Random();
    assumed r == Random();
  }

  [Pure]
  int Random() {
    return (new object()).GetHashCode();
  }
}

class D {
  int[] a;

  void N() {
    assume a != null && 0 < a.Length;
    var v = FirstOrZero();
    assumed v == FirstOrZero();
    a[0] = v + 1;
    assumed v == FirstOrZero();
    assert v == FirstOrZero();
    assumed v == FirstOrZero();
  }

  [Pure]
  int FirstOrZero() {
    return a != null &&
      0 < a.Length ? a[0] : 0;
  }
}
```

These explicit assumptions under-approximate Clousot's unsoundness due to non-deterministic methods. For example, assume it was possible that method `Random` in the example above returns the same value the first two times it is called, and a different value the third time. (This is not possible with the implementation of `Random` above.) In this case, our explicit assumption before the assertion in method `M` evaluates to true, but the assertion fails. Consequently, this

unsoundness is not precisely captured by our instrumentation since we cannot express determinism of method calls.

2.2 Local Unsoundness

We now present the sources of unsoundness in Clousot that are related to properties local to a method. We divide them into two categories: (1) integral-type arithmetic operations and conversions, and (2) exceptional control flow.

Integral-type arithmetic operations and conversions. To reduce the number of false positives, *Clousot ignores overflow in integral-type arithmetic operations and conversions*. In other words, Clousot treats bounded integral-type expressions as unbounded. Note, however, that Clousot's treatment of these operations and conversions is sound within `checked` expressions, which raise an exception when an overflow occurs.

As an example of this unsoundness, consider the code on the right, where `a` is of type `int`. Although the assertion fails when an overflow occurs, no warnings are emitted.

```
a++;
assumed (long)((int)(a + 1)) == (long)a + (long)1;
assert int.MinValue < a;
```

We capture this unsoundness by introducing an `assumed` statement after each bounded arithmetic operation that might overflow (and is not `checked`) stating that the operation returns the same value as its unbounded counterpart. More specifically, the explicit assumption states that the operation returns the same value as if it were performed on operands with types for which no overflow can occur, as shown in the code above. Note that primitive types with bounded precision are sufficient to express this unsoundness except in the case of `long` integers. For these, we use arbitrarily large integers (`BigInteger`s in C#) as the type for which no overflow can occur. We label this kind of explicit assumptions as "overflows" (**O**).

As another example of this unsoundness, consider the code on the right. Even though the assertion fails due to an overflow that occurs when converting `a` to a `short` integer, Clousot does not emit any warnings.

```
int a = int.MaxValue;
short b = (short)a;
assumed a == (short)a;
assert b == int.MaxValue;
```

We capture this unsoundness by introducing an `assumed` statement for each integral-type conversion to a type with smaller value range stating that the value before the conversion is equal to the value after the conversion, as shown in the code above. We label this kind of explicit assumptions as "conversions" (**CO**).

Our explicit assumptions over-approximate this source of unsoundness in Clousot only when the tool has enough information about the possible values of an integral-type arithmetic operation or conversion in an abstract domain, like an interval domain, to know that an overflow is impossible. When this is not the case, our assumptions precisely capture this unsoundness.

Exceptional control flow. Reasoning about exceptions requires control-flow transitions to exception-handling code at all program points where an exception might be thrown. Consequently, to avoid losing efficiency and precision, static analyzers typically ignore exceptional control flow. *Clousot ignores* `catch` *blocks*

and assumes that the code in a `finally` *block is executed only after a non-exceptional exit point of the* `try` *block has been reached.*

As an example of this unsoundness, consider the code on the right. Since Clousot ignores the existence of the `catch` block, no warning is emitted about the assertion.

```
try {
  throw new Exception();
} catch (Exception) {
  assumed false;
  assert false;
}
```

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each `catch` block stating that the block is unreachable, as shown in the code above. We label this kind of explicit assumptions as "catch blocks" (**C**).

As another example of this unsoundness, consider the code on the right. Since Clousot assumes that the `finally` block is only entered when the `try` block executes normally, no warning is emitted about the assertion. (We use `*` to denote any boolean condition.)

```
bool b = false;
bool $noException$ = false;
try {
  if (*) { throw new Exception(); }
  b = true;
  $noException$ = true;
} finally {
  assumed $noException$;
  assert b;
}
```

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each `finally` block stating that the block is entered only when the `try` block executes normally. This is expressed by introducing a fresh boolean ghost variable before each `try` block, which we set to true at all non-exceptional exit points of the `try` block, as shown in the code above. The `assumed` statement in the `finally` block then states that this ghost variable is true. We label this kind of explicit assumptions as "finally blocks" (**F**).

2.3 Static-State Unsoundness

Here, we describe the sources of unsoundness in Clousot that are related to static fields and main methods.

Static fields. Users can specify properties about static fields only with method pre- and postconditions; there are no static class invariants in Code Contracts. To reduce the annotation overhead and the number of false positives, *after a read operation from a static field of reference type, Clousot assumes that the static field is non-null.*

As an example of this unsoundness, consider the code on the right, for which no warnings are emitted.

```
static int[] a;

void M() {
  int[] b = a;
  assumed a != null;
  assert b != null;
}
```

We precisely capture this unsoundness by introducing an `assumed` statement after each read operation from a static field of reference type stating that the static field is non-null, as shown in the code. We label this kind of explicit assumptions as "static fields" (**S**).

Main methods. When a main method is invoked by the runtime system, the array of strings that is passed as an argument to the method and the elements of the array are never null. To spare its users from having to provide preconditions to main methods and to avoid emitting false positives in case there are no such preconditions, *Clousot assumes that the string array passed to every main method*

(that takes a parameter) and its elements are non-null for all invocations of the method.

As an example of this unsoundness, consider the code on the right. Although method `M` calls `Main` with a null argument, no warnings are emitted about the assertions in `Main`.

```
void M() {
  Main(null);
}

public static void Main(string[] args) {
  assumed args != null && forall arg in args | arg != null;
  assert args != null;
  assert args.Length == 0 || args[0] != null;
}
```

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each main method stating that the parameter array and its elements are non-null, as shown in the code above. (We use the special `forall` keyword to denote Code Contracts' universal quantifiers.) We label this kind of explicit assumptions as "main methods" (**M**).

2.4 Uninstrumented Unsoundness

In the rest of this section, we give an overview of the remaining sources of unsoundness in Clousot, which we do not instrument:

- *Concurrency*: Clousot does not reason about concurrency and assumes that the analyzed code runs in a single thread.
- *Static initialization*: Clousot assumes that the execution of the analyzed code is not interrupted by the execution of a static initializer.
- *Floating-point numbers*: Clousot may (occasionally) assume that operations on floating-point numbers are commutative.
- *Iterators*: Clousot does not analyze iterator methods (indicated with the `yield` keyword in C#).

Instrumenting the first two sources of unsoundness requires monitoring almost every program instruction, which is too expensive in practice. We do not instrument the unsoundness about floating-point numbers because, without access to Clousot's source code, we cannot be certain about which operations are assumed to be commutative and in which cases. At the beginning of each iterator method, we introduce an `assumed false` statement, which means that we consider all calls to iterators to be violating an unsound assumption made by Clousot. We decided not to include these results in our experimental evaluation since Clousot simply does not analyze iterators.

3 Implementation

To evaluate whether the sources of unsoundness of Sect. 2 are violated in practice, we have implemented a tool chain consisting of two stages: instrumentation and runtime checking. We describe these stages in this section.

Stage 1: Instrumentation. The instrumentation stage runs Clousot on a given .NET program, which contains code and optionally specifications expressed in Code Contracts, and instruments the sources of unsoundness of the tool as described in the previous section. For this purpose, we have implemented a wrapper around Clousot, which we call Inspector-Clousot, that inspects the debug output

emitted during the analysis. Based on this output, Inspector-Clousot rewrites the program (at the binary level) to instrument the unsoundness of the static analyzer. When Inspector-Clousot determines that the analyzer makes an unsound assumption at a certain program point, it introduces an `assumed` statement at that point in the code.

Stage 2: Runtime checking. In the runtime checking stage of the tool chain, we first run the Code Contracts binary rewriter to transform any Code Contracts specifications of the program into runtime checks. For example, postconditions of a method, which are specified at the beginning of the method body, are transformed into runtime checks occurring at every return point of the method.

We subsequently run a second rewriter, which we call Explicit-Assumption-Rewriter, that transforms all `assumed` statements of the program into logging operations. More specifically, the Explicit-Assumption-Rewriter replaces each explicit assumption `assumed` P by an operation that logs the program point of the `assumed` statement, which kind of unsoundness it expresses, and whether the assumed property is violated. Since property P may contain method calls, we take special care not to further log assumed properties in the callees.

The Explicit-Assumption-Rewriter also instruments each method to compute its set of written objects by keeping track of all object allocations and updates to instance fields and array elements. The set of written objects is then computed by determining which objects have been modified but are not newly allocated. Each method returns its set of written objects through static state, instead of having an additional return value (that is, an `out` parameter in C#), so that the method interface remains unmodified. This allows us to handle uninstrumented (library) methods for which the interface cannot be changed. The set of written objects for a call to an uninstrumented method is always empty. Our explicit assumptions could state that calls to uninstrumented methods modify the entire heap, but this would significantly over-approximate Clousot's unsoundness about write effects.

Predicate `invariant`(o, t) is implemented as a public, non-virtual instance method that returns a boolean value. Each class is extended to define such a method. In the method body, we first evaluate whether the receiver, that is, object o, satisfies the object invariants inherited from its super-classes. If this is the case, we evaluate whether the receiver satisfies the invariant of its own class, otherwise we return false. Note that if a class does not declare an object invariant and does not have any super-classes that do, the method simply returns true.

Rewriting the program with both the Code Contracts binary rewriter and the Explicit-Assumption-Rewriter is necessary to transform all annotations into executable code.

4 Experimental Evaluation

In this section, we present our experiments for evaluating whether Clousot's unsound assumptions are violated in practice. We present which kinds of assumptions were violated in our evaluation and how often this was the case.

For our experiments, we used code from six open source C# projects of different application domains. We selected only applications that come with a test

Application	Description	CC	Methods w/violations		total
BCrypt.Net[1]	Password-hashing library	no	1	(9.1%)	11
Boogie[2]	Verification language and engine	yes	19	(15.7%)	121
ClueBuddy[3]	GUI application for board game	yes	25	(28.7%)	87
Codekicker.BBCode[4]	BBCode-to-HTML translator	no	9	(7.6%)	119
DSA[5]	Data structures and algorithms library	no	46	(24.1%)	191
Scrabble (for WPF)[6]	GUI application for Scrabble	yes	10	(13.9%)	72

Table 1: Summary of results. The first and second columns describe the C# applications. The third column indicates whether the applications contain Code Contracts. The fourth column shows the number of methods in which explicit assumptions were violated, their percentage over the total number of methods with explicit assumptions that were hit at runtime, and the total number of methods with explicit assumptions that were hit at runtime.

suite so that the experiments achieve good code coverage of the applications. We chose three of these applications to contain specifications expressed with Code Contracts in order to evaluate the impact of these specifications on our results. To collect our results, we ran our tool chain on code from these applications such that all our explicit assumptions were instrumented exactly as described in Sect. 2. We subsequently ran tests from the test suite of each application. With our runtime-checking stage, we were able to log which kinds of explicit assumptions were hit at runtime and which of those were violated.

Tab. 1 describes the applications we used in our experiments, indicates which of these applications contain Code Contracts, and shows the number of methods in which explicit assumptions were violated out of the total number of methods with explicit assumptions that were hit at runtime. In the applications with Code Contracts, 19.29% of the total number of methods (shown in Tab. 1) contain explicit assumptions that were violated at runtime. For the applications without Code Contracts, this percentage is 17.45%. This small difference may be caused by explicit assumptions about "invariants at method entries" and "invariants at call sites", which can be violated only in applications with Code Contracts. Moreover, in applications without Code Contracts, explicit assumptions about "purity" can be violated only for property getters since no methods are annotated with the `Pure` attribute.

Tab. 2 shows the number and percentage of violated explicit assumptions per application and kind of assumption. These numbers include all *executions* of a single `assumed` P statement. That is, different executions in different method executions or loop iterations are counted separately. Tab. 3 shows the corre-

[1] `http://bcrypt.codeplex.com`, rev: `d05159e21ce0`
[2] `http://boogie.codeplex.com`, rev: `8da19707fbf9`
[3] `https://github.com/AArnott/ClueBuddy`, rev: `c1b64ae97c01fec249b2212018f589c2d8119b59`
[4] `http://bbcode.codeplex.com`, rev: `80132`
[5] `http://dsa.codeplex.com`, rev: `96133`
[6] `http://wpfscrabble.codeplex.com`, rev: `20226`

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0 (0%)	275 (0.06%)	-	-	0 (0%)
IC	-	0 (0%)	0 (0%)	-	-	0 (0%)
A	0 (0%)	192,025 (14.97%)	671 (43.07%)	641 (11.64%)	629 (34.30%)	150 (12.03%)
W	0 (0%)	6,236 (1.45%)	35 (0.96%)	0 (0%)	0 (0%)	10 (0.13%)
P	0 (0%)	27 (0.03%)	12,198 (5.06%)	0 (0%)	0 (0%)	425 (2.89%)
O	102,508,372 (30.58%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
CO	0 (0%)	-	-	-	-	0 (0%)
C	-	-	-	-	1 (100%)	-
F	-	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S	0 (0%)	1 (0%)[7]	-	0 (0%)	129 (20.16%)	0 (0%)
M	-	-	-	-	-	-

IE : invariants at method entries
IC : invariants at call sites
A : aliasing
W : write effects
P : purity
O : overflows
CO : conversions
C : catch blocks
F : finally blocks
S : static fields
M : main methods

Table 2: The number and percentage of violated explicit assumptions per application and kind of assumption. These numbers include all executions of a single `assumed` P statement. The "-" indicates that no explicit assumptions of a particular kind were hit at runtime.

sponding numbers when counting only per *occurrence* of an `assumed` statement rather than per execution. For example, in BCrypt.Net, all assumption violations shown in Tab. 2 occur in only four `assumed` statements (see Tab. 3), which are all in the body of the same loop.

As shown in Tab. 2 and 3, three kinds of assumptions were not violated in our experiments, namely, "invariants at call sites" (**IC**), "conversions" (**CO**), and "finally blocks" (**F**). Assumptions about "invariants at call sites" were not violated because, in these applications, sub-classes do not strengthen the object invariants of their super-classes such that the called inherited methods violate them. After manually inspecting all assumptions about "conversions", we realized that these assumptions indeed cannot be violated in practice. Finally, our instrumentation introduced only 21 assumptions about "finally blocks". The majority of these 21 `finally` blocks were added by the compiler to desugar `foreach` statements. Our results indicate that these three compromises of soundness reflect good design decisions in Clousot. Note that assumptions about "main methods" (**M**) were either not introduced (because there were no main methods in the portions of the code we instrumented) or not hit during our experiments.

Here are our observations from manually inspecting the remaining kinds of explicit assumptions that were introduced in the code of these applications:

- Assumptions about "invariants at method entries" (**IE**): All violations of these assumptions were found in application ClueBuddy. These violations were all caused by object constructors that call property setters in their body. The object invariants are, therefore, violated on entry to the setters since the constructors have not yet established the invariants. This, perhaps, indicates a need for a Code Contracts' attribute for annotating methods that do not rely

[7] We use two-decimal precision in this table. The exact percentage is $\frac{100}{155428}\%$.

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0 (0%)	7 (9.59%)	-	-	0 (0%)
IC	-	0 (0%)	0 (0%)	-	-	0 (0%)
A	0 (0%)	26 (22.41%)	13 (46.49%)	18 (33.33%)	44 (38.94%)	4 (14.81%)
W	0 (0%)	1 (2.38%)	1 (2.22%)	0 (0%)	0 (0%)	2 (6.25%)
P	0 (0%)	1 (2.94%)	10 (12.99%)	0 (0%)	0 (0%)	11 (16.42%)
O	4 (8.16%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
CO	0 (0%)	-	-	-	-	0 (0%)
C	-	-	-	-	1 (100%)	-
F	-	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S	0 (0%)	1 (3.23%)	-	0 (0%)	16 (88.88%)	0 (0%)
M	-	-	-	-	-	-

IE : invariants at method entries
IC : invariants at call sites
A : aliasing
W : write effects
P : purity
O : overflows
CO : conversions
C : catch blocks
F : finally blocks
S : static fields
M : main methods

Table 3: The number and percentage of violated explicit assumptions per application and kind of assumption. These numbers are per occurrence of a single `assumed` P statement. The "-" indicates that no explicit assumptions of a particular kind were hit at runtime.

on the invariant of their receiver (similarly to the `NoDefaultContract` attribute in Spec# [2]).

- Assumptions about "aliasing" (**A**): These assumptions were violated in all applications except for BCrypt.Net. This is because the code of BCrypt.Net is written in a single class containing mostly static methods that manipulate strings and arrays.
- Assumptions about "write effects" (**W**): Tab. 3 shows that these assumptions were hardly ever violated. By inspecting assumptions of this kind that were not violated, we confirmed that the method write effects assumed by Clousot are usually conservative.
- Assumptions about "purity" (**P**): Most of these assumptions were violated in methods that return newly-allocated objects. In applications without Code Contracts, these assumptions were only introduced in property getters, but were never violated.
- Assumptions about "overflows" (**O**): These assumptions were violated only in BCrypt.Net. All violations occur in an `unchecked` block, which suppresses overflow exceptions. This indicates that, in this application, overflows are actually expected to occur, or even intended.
- Assumptions about "catch blocks" (**C**): Only one assumption of this kind was introduced (and violated) in a method that removes a node from an AVL tree in application DSA. The method returns true when the node is successfully removed from the tree. The body of the method is wrapped in a `try` block, and, in case an exception is raised, the `catch` block returns false.
- Assumptions about "static fields" (**S**): The violations of these assumptions were, in some cases, due to static fields being lazily initialized, that is, being assigned non-null values after having first been read. In other cases, null static fields were passed as arguments to library methods (that are designed to handle null parameters).

5 Related Work

To the best of our knowledge, there is no existing work on experimentally evaluating sources of deliberate unsoundness in static analyzers.

There are, however, several approaches for ensuring soundness of static analyzers and checkers, ranging from manual proofs (e.g., in [12]), over interactive and automatic proofs (e.g., in [5] and [4]), to less formal techniques, such as "smoke checking" in the Boogie verification engine [1].

Many static analyzers compromise soundness to improve on other qualities (see [7] for an overview), such as precision or efficiency, and there is existing work on evaluating these other qualities of static analyzers in practice. For instance, Sridharan and Fink [13] evaluate the efficiency of Andersen's pointer analysis, and Liang et al. [11] evaluate the precision of different heap abstractions. In this work, we show that such evaluations are also possible for the unsoundness in static analyzers, and propose a practical approach for doing so.

6 Conclusion

In this paper, we report on the first systematic effort to document and evaluate the sources of deliberate unsoundness in a widely used, commercial static analyzer. Our technique is general and applicable to any analyzer whose unsoundness is expressible using a code instrumentation. In particular, we have explained how to derive the instrumentation by concretizing relevant portions of the abstract state (in our case, the heap-graph). We believe that this approach generalizes to a large class of assumptions made by static analyzers.

We consider our work to be an important first step in discovering good tradeoffs for the design of static analyzers. We encourage designers of static analyzers to document all compromises of soundness they choose to make and to motivate them empirically. Such a documentation facilitates tool integration since static analyzers or test case generators could be applied to compensate for the explicit assumptions. Moreover, analyzers could use runtime information about violated explicit assumptions (for instance, collected during testing) to re-analyze parts of the code under different assumptions.

Acknowledgments. We thank Mike Barnett, Manuel Fähndrich, Francesco Logozzo, and Herman Venter for their valuable help and feedback.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
2. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.
3. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *CACM*, 53:66–75, 2010.
4. F. Besson, P.-E. Cornilleau, and T. P. Jensen. Result certification of static program analysers with automated theorem provers. In *VSTTE*, volume 8164 of *LNCS*, pages 304–325. Springer, 2013.

5. S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013.
6. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.
7. P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with ASTRÉE. In *TASE*, pages 3–20. IEEE Computer Society, 2007.
8. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.
9. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.
10. D. Jackson and M. C. Rinard. Software analysis: A roadmap. In *ICSE*, pages 133–145. ACM, 2000.
11. P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, pages 411–427. ACM, 2010.
12. J. Midtgaard, M. D. Adams, and M. Might. A structural soundness proof for Shivers's escape technique: A case for Galois connections. In *SAS*, volume 7460 of *LNCS*, pages 352–369. Springer, 2012.
13. M. Sridharan and S. J. Fink. The complexity of Andersen's analysis in practice. In *SAS*, volume 5673 of *LNCS*, pages 205–221. Springer, 2009.