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 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. This paper reports on the first systematic effort to document and evaluate the sources of unsoundness in a static analyzer. We developed a code instrumentation that reflects the sources of deliberate unsoundness in the .NET static analyzer Clousot and applied it to code from six open-source projects. We found that 33% of the instrumented methods were analyzed soundly. In the remaining methods, Clousot made unsound assumptions, which were violated in 2–26% of the methods during concrete executions. Manual inspection of these methods showed that no errors were missed due to an unsound assumption, which suggests that Clousot's unsoundness does not compromise its effectiveness. Our 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, reduce the number of false positives or the annotation overhead, and achieve a modular analysis. As a result, such static analyzers become precise and efficient in detecting software bugs, but at the cost of making implicit, unsound assumptions about certain program properties. For example, ESC/Java uses bounded loop unrolling to reduce the overhead of writing loop invariants, and Spec# ignores exceptional control flow to speed up verification.

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 efficiency and precision of static analyzers by measuring, for instance, their performance and the number of false positives [2]. In this paper, we focus on a different perspective: 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 [10], an abstract interpretation tool for .NET and Code Contracts [9]. This instrumentation adapts

our earlier technique to make the unsound assumptions of a static analyzer explicit where they occur by automatically inserting annotations into the analyzed code [6]. Most of these assumptions are motivated by Clousot's design goal to analyze programs modularly without imposing an excessive annotation overhead. To evaluate the impact of Clousot's unsound assumptions, we instrumented code from six open-source projects, measured how often the unsound assumptions were violated during executions of the projects' test suites, and determined whether Clousot missed bugs due to unsound assumptions.

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.
- We present a code instrumentation that reflects the unsoundness in Clousot. Most 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 and whether they cause the analyzer to miss bugs.

In our experiments, 33% of the instrumented methods were analyzed soundly. In the remaining methods, Clousot made unsound assumptions, which were violated in 2–26% of the methods during concrete executions. Manual inspection of these methods showed that no errors were missed due to an unsound assumption, which suggests that Clousot's unsoundness does not compromise its effectiveness. We expect these results to guide users of static analyzers in using them fruitfully, for instance, in deciding how to complement static analysis with testing, and to assist designers of static analyzers in finding good trade-offs.

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. In Sect. 4, we present and discuss 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 elicited Clousot's unsound assumptions during the last two years by studying publications, extensively testing the tool, and having numerous discussions with its designers. Note that a formal proof that Clousot is sound modulo the issues we document here is beyond the scope of our paper.

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

Each unsound assumption in Clousot applies to a specific syntactic category such as a kind of statement or expression (for instance, because Clousot's abstract transformer does not soundly reflect the semantics of that syntactic category). We say that an explicit assumption *precisely* captures the unsound assumption for a syntactic category if for all elements e of that category and all executions τ of e, Clousot's analysis is sound iff the execution τ does not violate e's explicit assumption. Here, *sound* means that the concrete states of τ lie within the concretization of the corresponding abstract states. We say that an explicit assumption *over-approximates* the unsound assumption if there is an element e and an execution τ of e such that Clousot's analysis is sound, but the execution τ violates e's explicit assumption. Conversely, an explicit assumption *under-approximates* the unsound assumption if there is an element e and an execution τ of e such that Clousot's analysis is *not* sound, but the execution τ *does not* violate e's explicit assumption.

2.1 Heap Properties

Clousot treats the following aspects of the heap unsoundly: object invariants, aliasing, write effects, and method purity.

Object invariants. Code Contracts provide object (or class) invariants to express which objects are considered valid. Clousot checks the invariant of the receiver at the end of a method or constructor, and assumes it in the pre-state of a method execution and after a call. However, the checks are insufficient to justify these assumptions [8]. That is, Clousot makes the following unsound assumptions to facilitate modular checking: *Clousot assumes the invariant of the receiver object in the pre-state of instance methods, without checking it at call sites*; moreover *Clousot assumes the invariant of the receiver after a call to an inherited method on* `this`*, without fully checking it.*

The C# code on the right illustrates the first unsoundness. Method `M` violates the invariant of its receiver before calling `N`. (We use the keywords `invariant` and `assert` to denote Code Contracts' object invariants and assertions.) The gray boxes in the code are discussed later. 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, before the call to `N` in `M`. So Clousot emits no warning for the assertion in `N`, although it will not hold when `N` is called from `M`. The fact that there is no warning for the assertion in `M` is a consequence of the same unsoundness. Clousot checks the receiver's invariant in the post-state of method `N`; this check succeeds because of the same unsound assumption in `N`'s pre-state. The check in the post-state justifies assuming the invariant after the call.

```
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;
  }
}
```

We capture this unsoundness by introducing an `assumed` statement at the beginning of each instance method in classes that declare or inherit object invariants. As shown in the gray boxes in the code, these explicit assumptions use a predicate `invariant`(o, t), which holds iff object o satisfies the object invariants defined in class t in conjunction with all invariants inherited from t's super-classes. Here, type t is 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 assumption as "invariants at method entries" (**IE**). We will refer to such labels in our experimental evaluation.

This explicit assumption captures the first unsoundness precisely because any method execution in which the explicit assumption is violated (that is, where the receiver's invariant does not hold in the pre-state), will be analyzed with an unsound abstraction of the initial state (unless Clousot's abstract domains do not reflect the invariant anyway, which we ignore here). This does not necessarily mean that Clousot misses errors because the unsoundness might be irrelevant for the checks performed on the method body. Conversely, if the abstraction of the initial state is unsound because the receiver's invariant is violated, the explicit assumption will be false. Note that there are programs for which this will never happen; some explicit assumptions may always hold in these programs (and still be precise according to our definition).

The code on the right illustrates the second unsoundness. 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. However, since Clousot's analysis is modular, `Sub`'s invariant is not considered when analyzing `Super` and, therefore, Clousot does not detect this invariant violation. Nevertheless, Clousot assumes the invariant of `this` after the call to `N` in `M`, which is unsound. As a result, no warnings are emitted.

```
class Super {
  bool b;

  void N() { b = true; }
}

class Sub : Super {
  invariant !b;

  Sub() { b = false; }

  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`) and its super-classes. We label this kind of explicit assumption as "invariants at call sites" (**IC**).

Aliasing. To avoid the overhead of a precise heap analysis, *Clousot ignores certain side-effects due to aliasing.* For operations with side-effects, such as field updates, Clousot unsoundly assumes that heap locations not explicitly aliased in the code are non-aliasing and, thus, not affected.

As an example of this unsoundness, consider method `M` below. (We use the keyword `requires` to denote preconditions.) 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.

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

Clousot abstracts the heap by a *heap-graph*, which maintains equalities about access paths. The nodes of the heap-graph denote symbolic values, which 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. (Programmers may declare a method as pure to indicate that it makes no visible state changes.) All access paths in the heap-graph are rooted in a local variable or a method parameter. When two access paths lead to the same symbolic value, they represent the same concrete value, 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. Nevertheless, Clousot unsoundly assumes in this case that updating the heap through one path will not affect values read through the other.

We precisely capture this unsoundness by introducing an `assumed` statement before every side-effecting operation that unsoundly affects the values in the heap-graph, that is, when the side effect is reflected only on some symbolic values, although other symbolic values may represent the same heap locations. Specifically, for each field, property, or array update (side effects via calls are discussed below), we determine the set of symbolic values that are distinct from the symbolic value for the receiver r of the update, but may be aliases of r. This set is computed based on the heap-graph in the pre-state of the update and on type information. For each element s of this set, our explicit assumption has a conjunct expressing that the concrete values represented by r and s (and given by the access paths leading to the symbolic values) are non-aliasing.

In our example, Clousot's heap abstraction uses distinct symbolic values for the arrays `a` and `b` in the initial heap-graph. Thus, for the first array update, r represents `a` and the set of possible aliases consists of `b`. Hence, the explicit assumption expresses that `a` and `b` are not aliases. The explicit assumption for the second array update is analogous. Note that we call `ReferenceEquals` since the `==` operator may be overloaded in C#. We label this kind of explicit assumption 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. This assumption is unsound since the heuristics in general may not include all heap locations that are modified by a call.

The code on the right illustrates this unsoundness. Clousot assumes that the call to method `N` in `M` modifies only the fields of the receiver object, and leaves the elements of the array unchanged. As a result, it does not emit a warning for the assertion. Note that this unsoundness is caused by Clousot's heuristics for write effects, regardless of whether `a` and `b` are aliases.

```
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; }
  }
}
```

We capture this unsoundness by introducing an `assumed` statement after each call, stating that all heap locations in the heap-graph that Clousot assumes to remain unmodified by the call are indeed not modified. This is achieved by comparing all symbolic values in the heap-graph before and after the 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 stating that it is not contained in the actual write effect of the method for the last call. To obtain the actual write effect, we instrument the program to provide the function `writtenObjects`, which returns the set of objects that were modified by the most recently executed call (including any objects that were modified indirectly through method calls). We label this kind of explicit assumption as "write effects" (**W**). Note that this explicit assumption subsumes the aliasing unsoundness for calls because it covers all objects Clousot assumes to be left unchanged by a call, no matter whether this assumption is caused by ignoring certain aliasing situations or by the unsound heuristics for write effects. In method `M` above, `writtenObjects` returns the set consisting of array `a` and, since `a` and `b` refer to the same array, the explicit assumption is violated at runtime.

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 modified by the most recently executed call then our assumptions over- or under-approximate Clousot's unsoundness, respectively. In our implementation, `writtenObjects` is precise for methods that we instrument, but under-approximates the write effects of library methods (see Sect. 3).

Purity. Users may explicitly annotate a method with the Code Contracts attribute `Pure` to express 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 indeed make no visible state changes.* (We will refer to property getters and methods annotated with `Pure` as "pure methods".) Moreover, Clousot uses unsound heuristics to determine which heap locations affect the result of a pure method, that is, 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.

Method `M` on the right illustrates the second unsoundness. Clousot assumes that both calls to the pure method `Random` in `M` deterministically return the same value, and no warning is emitted.

```
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() {
    requires a != null && 0 < a.Length;
    var v = First();
    assumed v == First();
    a[0] = v + 1;
    assumed v == First();
    assert v == First();
    assumed v == First();
  }

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

Method `N` on the right illustrates another aspect of this unsoundness. Clousot assumes that the result of the pure method `First` depends only on the state of its receiver, but not on the state of array `a`. Therefore, no warning is emitted about the assertion in `N` even though `a[0]` is modified after the first call to `First`.

Clousot's heap-graph maintains information about which values may be retrieved by calling a pure method. For instance, after the first call to `Random` in `M`, the heap-graph maintains an equality of `r` and a call to `Random`. This information becomes unsound if (1) the pure method is not deterministic, (2) an object is modified, but Clousot unsoundly assumes that the pure method does not depend on that object, or (3) an object is modified, but Clousot does not reflect the modification correctly in the heap-graph. The latter case is covered by the explicit assumptions for aliasing and write effects. We capture the former two cases as follows: (1) We generate an explicit assumption after each call to a pure method stating that the method still yields the value stored in the heap-graph. This assumption under-approximates Clousot's unsoundness due to non-determinism since even a non-deterministic method might return the same result several times in a row. (2) Whenever the heap-graph retains a value for a pure method call across a statement that may modify the heap, we generate an explicit assumption stating that the method still yields the value stored in the heap-graph. This assumption precisely captures the case that Clousot may assume a too small read effect, as for method `First`. We label these explicit assumptions as "purity" (**P**).

2.2 Method-Local Properties

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

Integral-type arithmetic. To reduce the number of false positives, *Clousot ignores overflow in integral-type arithmetic operations and conversions*. That is, Clousot treats bounded integral-type expressions as unbounded (except for `checked` expressions, which raise an exception when an overflow occurs).

The code on the right illustrates the unsoundness for operations. Although the assertion fails when an overflow occurs, no warning is emitted.

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

We precisely capture this unsoundness by introducing an `assumed` statement before each bounded arithmetic operation that might overflow (and is not `checked`) stating that the operation returns the same value as its unbounded counterpart. We encode this unbounded counterpart by performing the operation on operands with types for which no overflow will occur, for instance, `long` instead of `int` as in the example above, or arbitrarily large integers (`BigInteger`) instead of `long`. We label this kind of explicit assumption as "overflows" (**O**).

The code on the right illustrates the unsoundness for conversions. 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;
assumed a == (short)a;
short b = (short)a;
assert (int)b == int.MaxValue;
```

We precisely 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 above. We label this kind of explicit assumption as "conversions" (**CO**).

Exceptional control flow. Exceptions add a large number of control-flow transitions and, thus, complicate static analysis. To avoid losing efficiency and precision, many static analyzers 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 corresponding* `try` *block has been reached.*

The code on the right illustrates the unsoundness for `catch` blocks. Since Clousot ignores 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 assumption as "catch blocks" (**C**).

The code on the right illustrates the unsoundness for `finally` blocks. Since Clousot assumes that the `finally` block is entered only when the `try` block executes normally, no warning is emitted about the assertion. (We use `*` to denote an arbitrary 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 terminates normally. This is expressed by introducing a fresh boolean variable for each `try` block, which is initially false and set to true at all non-exceptional exit points of the `try` block, as shown in the code. The `assumed` statement then states that this variable is true. We label this kind of explicit assumption as "finally blocks" (**F**).

2.3 Static Class Members

Here, we describe the sources of unsoundness for static fields and main methods.

Static fields. To avoid the complications of class initialization [5] and to reduce the annotation overhead and the number of false positives, *Clousot assumes that static fields of reference types contain non-null values.*

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

```
static int[] a;

void M() {
  assumed a != null;
  assert a != null;
}
```

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

Main methods. When a main method is invoked by the runtime system, the array of strings that is passed to the method and the array elements are never null. To relieve its users from providing preconditions for main methods, *Clousot assumes that the string array passed to a main method and its elements are non-null for all invocations of the method.*

As an example, consider the code on the right. Although method `M` calls `Main` with a null argument, no warning is 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 `forall` keyword to denote Code Contracts' universal quantifiers.) We label this kind of explicit assumption 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 without interference from other threads.
- *Reflection*: Clousot assumes that the analyzed method does not use reflection.
- *Unmanaged code*: Clousot checks memory safety for unmanaged code, but does not consider its effects on the analyzed method.
- *Static initialization*: Clousot assumes that the analyzed code runs without interference from a static initializer.

- *Iterators*: Clousot does not analyze iterator methods (C#'s `yield` statements).
- *Library contracts*: Clousot assumes that the contracts provided for libraries such as the .NET API are correct.
- *Floating-point numbers*: Under certain circumstances, Clousot assumes that operations on floating-point numbers are commutative.

A very coarse way of capturing the first five sources of unsoundness would be to introduce an `assumed false` statement at each program point that starts a thread, invokes reflection, or contains unmanaged code, as well as in each static initializer and for each `yield` statement. Such an instrumentation would grossly over-approximate Clousot's unsound assumptions (for instance, many static initializers do not interfere with the execution of the analyzed method). However, a more precise instrumentation is complicated and would require explicit assumptions for most statements, for instance, to detect data races. Incorrect library contracts could be detected by introducing an explicit assumption for the postcondition of each call into the library. We omit these assumptions because they are orthogonal to the design of the static analyzer. Finally, we do not instrument the unsoundness about floating-point numbers because we were not able to precisely determine where the assumptions occur.

Note that we do not consider Clousot's inference of method contracts and object invariants in this paper. In the presence of inference, an unsound assumption in a method m might affect not only the analysis of m but also of methods whose analysis assumes properties inferred from m, in particular, m's postcondition and the object invariant of the class containing m. One solution is to introduce an explicit assumption whenever Clousot assumes a postcondition or invariant that was inferred unsoundly; one can then determine easily which methods have been analyzed soundly by inspecting the instrumented method body. Another solution is to rely on the existing instrumentation, which is sufficient to reveal unsound inference during the execution of the program. If the postcondition of a method or constructor m was inferred unsoundly, we detect an assumption violation when executing a call to m, and analogously if m violates an inferred invariant.

3 Implementation

To evaluate whether Clousot's sources of unsoundness are violated in practice, we have implemented a tool chain that instruments code with explicit assumptions and checks them at runtime.

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 *Inspector-Clousot*, a wrapper around Clousot that uses the debug output emitted during the analysis to instrument the program (at the binary level).

Runtime checking. In the runtime checking stage, we first run the existing Code Contracts binary rewriter to transform Code Contracts specifications into runtime checks. We subsequently run a second rewriter, called *Explicit-*

Application	Description	CC	Analyzed methods	Methods with violations
BCrypt.Net[1]	Password-hashing library	no	21	1 / 12 (8.3%)
Boogie[2]	Verification language and engine	yes	299	2 / 119 (1.7%)
ClueBuddy[3]	GUI application for board game	yes	139	16 / 67 (23.9%)
Codekicker.BBCode[4]	BBCode-to-HTML translator	no	179	2 / 58 (3.4%)
DSA[5]	Data structures and algorithms library	no	213	26 / 99 (26.3%)
Scrabble (for WPF)[6]	GUI application for Scrabble	yes	127	8 / 41 (19.5%)

Tab. 1: Applications selected for our experiments. The first two columns describe the C# applications. The third column indicates whether the applications contain Code Contracts. The fourth column shows the number of analyzed methods per project. The fifth column shows how many of the methods with explicit assumptions that were hit at runtime contained assumption violations.

Assumption-Rewriter, that transforms all `assumed` statements of the instrumented program into logging operations. More specifically, this 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 P is violated. If P contains method calls, we do not 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 of a method consists of the objects that have been modified but are not newly allocated by the method. The set of written objects for a call to an uninstrumented (library) method is always empty, that is, our instrumentation under-approximates the objects actually modified by such a method.

4 Experimental Evaluation

In this section, we present our experiments for evaluating whether Clousot's unsound assumptions are violated in practice and whether these violations cause Clousot to miss errors.

For our experiments, we used code from six open-source C# projects (see Tab. 1) from different application domains. We selected only applications that come with a test suite so that the experiments achieve good code coverage. We chose three applications to contain Code Contracts specifications to evaluate the explicit assumptions about object invariants. We ran our tool chain on at least one substantial DLL from these applications to perform the instrumentation described in the previous sections. For invoking Clousot, we enabled all checks, set the warning level to the maximum, and disabled all inference options. We

[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`

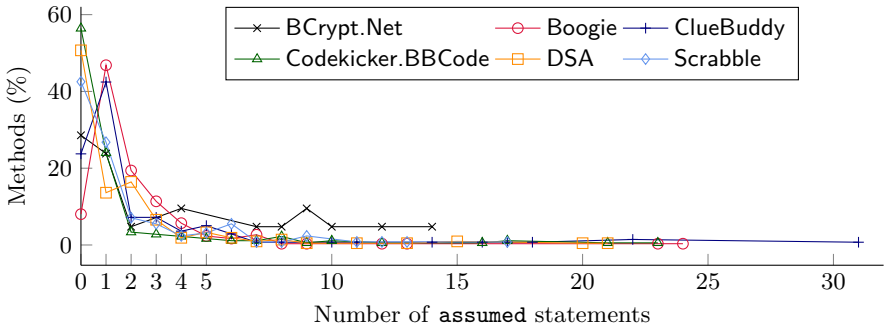

Fig. 1: The percentage of analyzed methods from each project versus the number of `assumed` statements in the methods.

subsequently ran tests from the test suite of each application and logged which explicit assumptions were hit at runtime and which of those were violated. Finally, we manually inspected a large number of methods to determine whether Clousot misses any errors because of its unsound assumptions.

4.1 Experimental Results: Instrumentation

Fig. 1 presents the percentage of analyzed methods from each project versus the number of `assumed` statements in the methods. An *analyzed* method is checked by Clousot but not necessarily hit at runtime by the test suite of a project. We analyzed a total of 978 methods with Clousot. As shown in the figure, the majority of these methods (860) contain less than 5 `assumed` statements, and a large number of those (326) are soundly checked, that is, do not contain any explicit assumptions. There are only 20 methods with more than 10 `assumed` statements. In these methods, the prevailing sources of unsoundness are "invariants at call sites" (**IC**), "write effects" (**W**), "purity" (**P**), and "overflows" (**O**).

Fig. 2 shows the average number of bytecode instructions in the analyzed methods versus the number of `assumed` statements in the methods. Notice that most methods that are soundly checked contain only a small number of bytecode instructions. A manual inspection of these methods showed that many of them are setters, getters, or (default) constructors. Our results indicate that methods with more instructions contain a larger number of `assumed` statements.

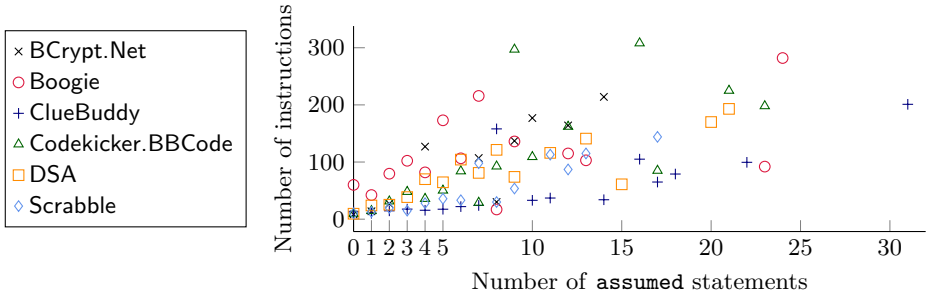

Fig. 2: The average number of bytecode instructions in the analyzed methods from each project versus the number of `assumed` statements in the methods.

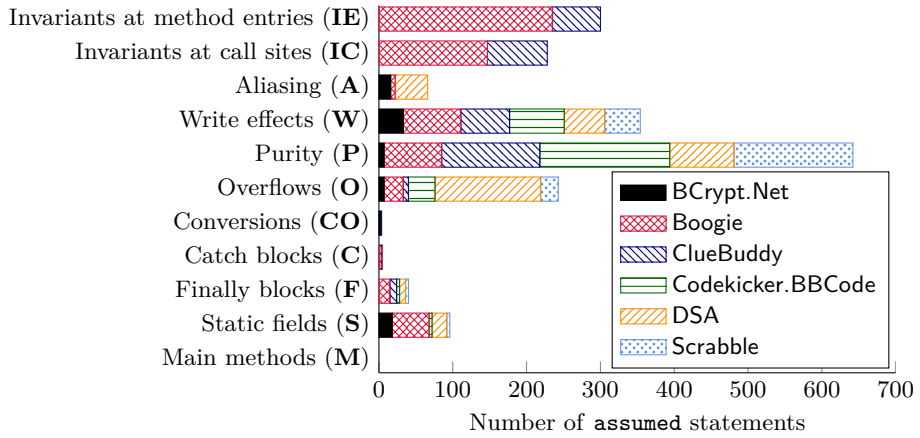

Fig. 3: Clousot's sources of unsoundness versus the number of `assumed` statements that are introduced in the analyzed methods of each project.

Fig. 3 shows Clousot's sources of unsoundness versus the number of `assumed` statements that are introduced in the analyzed methods of each project. The results are dominated by the assumptions that are introduced for each method (**IE**) or for common statements (**IC**, **W**, **P**). The unsound treatment of aliasing (**A**) affects relatively few methods, even though it could be introduced for each field, property, or array update. Assumptions about "main methods" (**M**) were not introduced because there are either no main methods at all (for instance, in libraries) or not in the portions of the code that we analyzed and instrumented.

4.2 Experimental Results: Runtime Checking

The experimental results for the instrumentation alone provide very limited insight into the impact of Clousot's unsoundness. For instance, while some explicit assumptions reflect details of the analysis (such as **A** and **W**, which are based on Clousot's heap-graph), others merely indicate the existence of a syntactic element (for instance, we generate one assumption of kind **C** per catch-block). Moreover, some explicit assumptions are not violated in any concrete program execution; for instance, the assumptions of kind **M** always hold if a program does not call a main method. To better understand the impact of Clousot's unsound assumptions, we measure how often the generated explicit assumptions are violated during concrete program executions.

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` statement. That is, different executions of the same `assumed` statement in different method invocations or loop iterations are counted separately. Tab. 3 shows the corresponding numbers when counting only per *occurrence* of an `assumed` statement rather than per execution. For example, in BCrypt.Net, the assumption violations shown in Tab. 2 occur in only 4 `assumed` statements (see Tab. 3), which are all in the body of the same loop.

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0/1694124 (0%)	275/27318 (1.01%)	-	-	-
IC	-	0/628448 (0%)	0/9759 (0%)	-	-	-
A	0/25844436 (0%)	0/24771 (0%)	-	-	131/992 (13.21%)	-
W	0/6419169 (0%)	0/372851 (0%)	0/3589 (0%)	82/11577 (0.71%)	0/613 (0%)	25/5011 (0.50%)
P	0/6405279 (0%)	27/108506 (0.02%)	12198/241385 (5.05%)	0/10311 (0%)	0/1008 (0%)	425/21580 (1.97%)
O	102488804/326722626 (31.37%)	0/569258 (0%)	0/547 (0%)	0/1196 (0%)	0/6053 (0%)	0/909 (0%)
CO	0/6633876 (0%)	-	-	-	-	0/2 (0%)
C	-	-	-	-	1/1 (100%)	-
F	-	0/53246 (0%)	0/325 (0%)	0/114 (0%)	0/43 (0%)	0/65 (0%)
S	0/708 (0%)	1/155080 (0%)	-	0/7 (0%)	129/640 (20.16%)	0/15 (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

Tab. 2: The number and percentage (rounded to two decimal places) of violated explicit assumptions per application and kind of assumption. These numbers include all executions of a single `assumed` statement. Cells with non-zero values are highlighted; the "-" indicates that no explicit assumptions are hit at runtime.

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0/108 (0%)	7/44 (15.91%)	-	-	-
IC	-	0/60 (0%)	0/59 (0%)	-	-	-
A	0/16 (0%)	0/1 (0%)	-	-	16/46 (34.78%)	-
W	0/30 (0%)	0/32 (0%)	0/43 (0%)	2/61 (3.28%)	0/51 (0%)	1/25 (4.00%)
P	0/7 (0%)	1/40 (2.50%)	10/81 (12.35%)	0/130 (0%)	0/86 (0%)	11/85 (12.94%)
O	4/11 (36.36%)	0/11 (0%)	0/5 (0%)	0/25 (0%)	0/134 (0%)	0/13 (0%)
CO	0/3 (0%)	-	-	-	-	0/1 (0%)
C	-	-	-	-	1/1 (100%)	-
F	-	0/3 (0%)	0/5 (0%)	0/3 (0%)	0/8 (0%)	0/2 (0%)
S	0/18 (0%)	1/31 (3.23%)	-	0/2 (0%)	16/18 (88.88%)	0/2 (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

Tab. 3: The number and percentage (rounded to two decimal places) of violated explicit assumptions per application and kind of assumption. These numbers are per occurrence of a single `assumed` statement. Cells with non-zero values are highlighted; the "-" indicates that no explicit assumptions are hit at runtime.

4.3 Manual Inspection

We manually inspected a large number of explicit assumptions, including all violated assumptions, and made the following observations.

- "Invariants at method entries" (**IE**): Only Boogie and ClueBuddy contain invariant specifications, and all violations are found in ClueBuddy. These violations are all caused by 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. Objects that escape from their constructors are a well-known problem; a possible solution is to annotate

methods that may operate on partially-initialized objects and, thus, must not assume their invariants [16].
- "Invariants at call sites" (**IC**): These assumptions are never violated because in all of our applications, sub-classes do not strengthen the object invariants of their super-classes such that calls to inherited methods could violate them.
- "Aliasing" (**A**): These assumptions are violated only in DSA. All violations occur in nine methods of two classes implementing singly and doubly-linked lists. For example, one violation occurs in method `AddAfter` when expressions `this.Tail`, `this.Head`, and the node to be added are aliased. The small number of these violations suggests that there is only a limited practical need for performing a sound, but expensive heap analysis. However, an analyzer could optionally allow users to run a sound heap analysis, for instance, for methods with violations of "aliasing" assumptions.
- "Write effects" (**W**): Tab. 3 shows that these assumptions are hardly ever violated. By inspecting assumptions of this kind that are not violated, we confirmed that the write effects assumed by Clousot are usually conservative.
- "Purity" (**P**): Most of these assumptions are violated for pure methods that return newly-allocated objects, that is, for non-deterministic methods. In applications without Code Contracts, these assumptions are introduced only in property getters, but are never violated.
- "Overflows" (**O**): These assumptions are 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.
- "Conversions" (**CO**): These assumptions are never violated. Our manual inspection showed that the value ranges of the converted expressions are sufficiently small such that no overflow may occur.
- "Catch blocks" (**C**): Only one assumption of this kind was introduced in a method that removes a value from an AVL tree in application DSA. An auxiliary method throws an exception when the AVL tree is empty. Catching this exception violates the assumption. This violation could be avoided by using an out-parameter instead of an exception to signal that the tree was empty.
- "Finally blocks" (**F**): Our instrumentation introduced only 39 assumptions about "finally blocks". The majority of these `finally` blocks are added by the compiler to desugar `foreach` statements. If the body of the `foreach` statement does not throw an exception, these assumptions are not violated.
- "Static fields" (**S**): The violations of these assumptions are, in some cases, due to static fields being lazily initialized, that is, being assigned non-null values after having first been read. Supporting lazy initialization via a language construct, such as Scala's "lazy val" declarations, could help avoid such violations. In other cases, the values of static fields are passed as arguments to library methods, which are designed to handle null arguments.

Missed errors. The violation of an explicit assumption does not necessarily mean that Clousot misses errors since the resulting unsoundness may be irrelevant for the subsequent checks. To determine whether the assumption violations

detected in our experiments might lead to missed errors, we manually inspected the containing methods of all 70 violations (computed from Tab. 3). We did not find any runtime errors or assertion violations that Clousot missed due to its unsound assumptions. With the exception of a few cases, it was fairly straightforward to determine whether an assumption violation could conceal an error. For instance, violations of explicit assumptions about "purity" (**P**) are harmless when there is only a single call to the pure method. The same holds for explicit assumptions about "aliasing" (**A**) when the updated field, property, or array element is not accessed after the update.

The fact that we did not find any missed errors due to assumption violations possibly indicates that providing slightly weaker soundness guarantees in certain situations in favor of performance, precision, and low annotation overhead does not compromise Clousot's effectiveness; its unsound assumptions are not problematic in the code and executions we investigated.

4.4 Threats to Validity

We identified the following threats to the validity of our experiments:

- *Instrumentation*: It is possible that we missed some of Clousot's unsound assumptions. Since we elicited the assumptions very diligently, it seems unlikely that we overlooked any major sources of unsoundness. There are several sources of unsoundness that we identified, but do not capture (see Sect. 2.4). For most of these sources, a syntactic check suffices to determine whether a program might be affected. Moreover, even though our instrumentation captures most of Clousot's unsound assumptions precisely, it under-approximates the unsound treatment of write effects for calls to uninstrumented (library) methods and of non-deterministic pure methods. As a result, it is possible that Clousot's analysis of a method is unsound even though all runtime checks for explicit assumptions pass (this is very unlikely for non-deterministic pure methods).
- *Runtime checking*: We measured assumption violations in executions of the projects' test suites. There were no failing tests, that is, any errors detected by the test suites have been fixed. This explains in part why we did not find any errors missed by Clousot. However, in our manual inspection of the violated assumptions, we checked the entire method, that is, all execution paths of the method for all its input states, not just the code covered by the test suite. Thus, we could have detected errors that the tests missed.
- *Project selection and sample size*: The projects in our experiments were chosen from different application domains. All projects were required to include a test suite. We selected projects with and without Code Contracts. Since Clousot analyzes each method modularly, we were able to pick those DLLs that have the most comprehensive test suites. We ran Clousot on 978 methods; `assumed` statements were added in 652 methods, 396 out of which were hit during the execution of the projects' test suites. Therefore, we believe that our projects are representative for a large class of C# code bases.

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 [14], over interactive and automatic proofs [3,4], to less formal techniques, such as "smoke checking" [1].

Many static analyzers compromise soundness to improve on other qualities such as precision or efficiency (see Cousot and Cousot [7] for an overview), and there is existing work on evaluating these other qualities of analyzers in practice. For instance, Sridharan and Fink [15] evaluate the efficiency of Andersen's pointer analysis, and Liang et al. [11] evaluate the precision of different heap abstractions. We show that such evaluations are also possible for the unsoundness in static analyzers, and propose a practical approach for doing so.

Our explicit assumptions could be used to express semantic environment conditions inferred from a base program, as in VMV [13]; a new version of the program could then be instrumented with these inferred conditions (in the form of assumptions) to reduce the number of warnings reported by Clousot. Moreover, our technique could be applied in "probabilistic static analyzers" [12] to determine the probabilities of their judgments about analyzed code. Specifically, one could estimate the probability that an unsound assumption holds (or is violated) based on its value along a number of concrete executions.

Finally, we refer the reader to `http://soundiness.org` for the "soundiness" movement in static program analysis, which brings forward the ubiquity of unsoundness in static analyzers, draws a distinction between analyzers with specific, well-defined soundness trade-offs and tools that are not concerned with soundness at all, and issues a call to the research community to clearly identify the nature and extent of unsoundness in static analyzers.

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.

Our work can help designers of static analyzers in finding good trade-offs. We encourage them to document all compromises of soundness and to motivate them empirically. Such a documentation facilitates tool integration since other static analyzers or test case generators could be applied to compensate for the explicit assumptions. Information about violated assumptions (for instance, collected during testing) could also be valuable in identifying methods that require special attention during testing and code reviews. Finally, our results could be used to derive programming guidelines and language designs that mitigate the unsoundness of a static analyzer.

Acknowledgments. We are especially grateful to Francesco Logozzo for numerous discussions and his active support; this work would not have been possible without his help. We also thank Mike Barnett, Manuel Fähndrich, and Herman Venter for their valuable help and feedback, and the reviewers for their constructive comments.

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. 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.
3. 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.
4. 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.
5. M. Christakis, P. Emmisberger, and P. Müller. Dynamic test generation with static fields and initializers. In *RV*, volume 8734 of *LNCS*, pages 269–284. Springer, 2014.
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. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.
9. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.
10. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.
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. B. Livshits and S. K. Lahiri. In defense of probabilistic static analysis. In *APPROX*, 2014.
13. F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, pages 294–304. ACM, 2014.
14. 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.
15. 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.
16. A. J. Summers and P. Müller. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA*, pages 1013–1032. ACM, 2011.