Static Detection of Brittle Parameter Typing

Michael Pradel
Dept. of Computer Science
ETH Zurich, Switzerland

Severin Heiniger
Dept. of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Dept. of Computer Science
ETH Zurich, Switzerland

ABSTRACT

To avoid receiving incorrect arguments, a method specifies the expected type of each formal parameter. However, some parameter types are too general and have subtypes that the method does not expect as actual argument types. For example, this may happen if there is no common supertype that precisely describes all expected types. As a result of such *brittle parameter typing*, a caller may accidentally pass arguments unexpected by the callee without any warnings from the type system. This paper presents a fully automatic, static analysis to find brittle parameter typing and unexpected arguments given to brittle parameters. First, the analysis infers from callers of a method the types that arguments commonly have. Then, the analysis reports potentially unexpected arguments that stand out by having an unusual type. We apply the approach to 21 real-world Java programs that use the Swing API, an API providing various methods with brittle parameters. The analysis reveals 15 previously unknown bugs and code smells where programmers pass arguments that are compatible with the declared parameter type but nevertheless unexpected by the callee. The warnings reported by the analysis have 47% precision and 83% recall.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging

General Terms

Experimentation, Languages, Reliability

Keywords

Automatic bug finding, static analysis, anomaly detection

1. INTRODUCTION

In statically-typed programming languages, the type system ensures that method arguments have a type expected by the callee. This check is done under the assumption that all subtypes of the declared parameter type are legal argument types [11].[1] Unfortunately, method parameters may have subtypes that are not expected by the callee. We call this situation *brittle parameter typing* (or simply a *brittle parameter*), because the safety guaranteed by the type system is easily breakable. A type system may find arguments given to brittle parameters to be legal, but in fact they are incorrect because the callee does not expect them.

ISSTA '12, July 15–20, 2012, Minneapolis, MN, USA

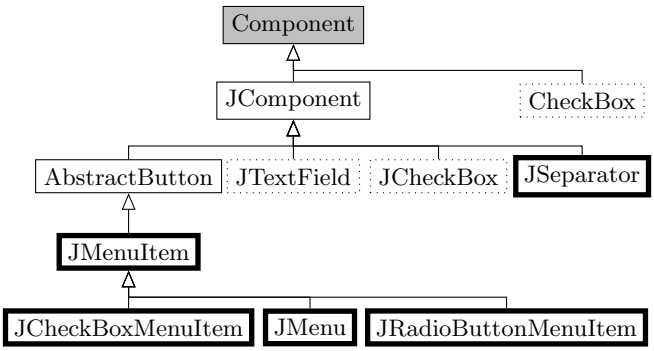

Figure 1: Excerpt of Swing's type hierarchy showing that `JMenu.add(Component)` has a brittle parameter. The declared parameter type has a gray background. Only the bold types should be passed as arguments. Dotted types are compatible but lead to incorrect behavior. Arguments of the remaining types may be correct or incorrect.

1.1 Example

For example, consider the method `JMenu.add(Component)` from the Java Swing API (Figure 1). The declared parameter type `Component` has various subtypes, all of which are valid argument types according to the method declaration of `add()`. However, the API documentation states that a menu can only contain `JMenuItem`s and `JSeparator`s, that is, a subset of all compatible argument types. If a programmer adds a `Component` with another compatible type, such as `CheckBox`, the call is incorrect because its behavior is undefined.

Figure 2 illustrates a bug caused by passing an unexpected argument to the brittle parameter of `JMenu.add(Component)`. We found this problem in a real-world program (nTorrent, a graphical user interface for a BitTorrent client[2]) and the

[1]We refer to formal parameters in a method declaration as *parameters* and to objects passed to methods at a call site as *arguments*.
[2]`http://code.google.com/p/ntorrent/`

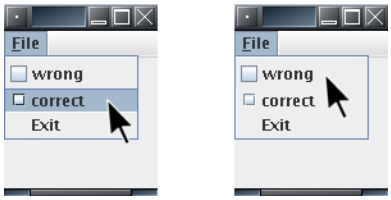

Figure 2: Demonstration of a real-world bug resulting from passing an unexpected argument to a brittle parameter.

developers confirmed it as a bug after receiving our report. The correct menu entry is of type `JCheckBoxMenuItem`. An item of this type is highlighted when hovering over it, and clicking it will close the menu. However, the programmer accidentally passes a `JCheckBox` to `JMenu.add(Component)`. In contrast to the correct menu item, the wrong item does not react on hovering, and clicking it does not give the expected behavior.

How can developers find errors caused by compatible but unexpected argument types? Finding errors related to brittle parameters is hard. First of all, traditional compilers and type systems are oblivious to the problem, because the argument type will type check given the declared parameter type. Furthermore, unexpected arguments may slip through traditional testing if they do not manifest through an exception. For example, the bug in Figure 2 does not raise an exception or any other obvious error. Instead, it leads to non-functional GUI elements as illustrated in Figure 2. Finally, programmers with little experience in programming against a particular API are prone to pass unexpected arguments, because the knowledge about expected argument types is often scarcely or not at all documented. For the above example, the expected types are listed in `JMenu`'s class documentation but not in the method documentation of `add()`.

1.2 Why Brittle Parameter Typing Exists

There are several reasons for brittle parameter typing in API methods. One reason is that the API designers had more functionality in mind when publishing the API and wanted to leave an easy way to later add this functionality without changing the method signature. Changing a parameter type to a more specific type after releasing an API is difficult, as it may break existing client code.

Brittle parameter typing also occurs when it is impossible to create a common supertype that precisely describes all expected types. If no such supertype exists, it may be possible to let all expected types implement a marker interface [1] and to use this interface as the parameter type. Unfortunately, this approach is infeasible if one or more of the expected types are declared outside of the API and therefore cannot be changed, for example, if `String` is among the expected argument types.

1.3 Our Approach

This paper presents a fully automatic, static analysis to (i) find brittle parameters and (ii) reveal unexpected arguments passed to methods with such parameters. The key idea is a simple one: We leverage existing API clients to infer the argument types that an API method expects and warn developers about apparently unexpected arguments. The analysis has two main steps. At first, it statically extracts argument types from call sites of API methods. Then, it searches for argument types that are unusual with respect the other arguments passed to the parameter. The second step assumes the analyzed client programs to be mostly correct, an assumption shared with other anomaly detection techniques [8, 3, 15, 23, 16].

Despite being simple, the approach is easy to apply and effective in practice. It is easy to apply, because it does not require any formal specification of expected argument types. Instead, all required information is automatically extracted from existing client code. As the analysis is independent of the API implementation, it is applicable to arbitrary third-party APIs. The approach is effective, because it reveals real programming errors. The price for being simple and effective is that the analysis is neither sound nor complete, that is, it might report spurious warnings and miss real errors. However, our results show both problems to be manageable in practice.

We evaluate the approach with 21 Java programs that use the Swing API. Swing has been developed on top of an existing API (AWT) and has a complex type hierarchy. As a result, Swing has various methods with brittle parameters, making it a good candidate for our evaluation. In total, the analysis reveals 15 previously unknown bugs and code smells in the analyzed programs, some of which have already been fixed in response to our bug reports. In its default configuration, the analysis has 47% true positives. The sensitivity of the analysis to unusual argument types can be tuned with threshold parameters, allowing developers to find an acceptable trade-off between precise warnings (few false positives) and finding many bugs. In addition to evaluating the approach with real bugs, we automatically seed bugs into existing programs. With its default parameters, the analysis finds 83% of these bugs.

1.4 Contributions

In summary, this paper contributes the following:

- We identify the problem of brittle parameter typing as a source of programming errors.
- We present a static analysis to detect unexpected arguments given to methods with brittle parameters. The analysis is fully automatic and requires no input except for the source code or byte code of API clients. In particular, the analysis does not rely on formal specifications or test cases.
- We present the results of applying the approach to real-world Java programs that use the Swing API. The results show that programming errors related to brittle parameters occur in practice and that the analysis detects them effectively.

2. APPROACH

The following section presents a static analysis to reveal method arguments that are unexpected by a callee despite having a type compatible with the declared parameter type. The approach is designed for analyzing calls from API clients to API methods. Figure 3 provides an overview of the analysis. As input, the analysis requires the source code or byte code of API clients. The first step is a static analysis that inspects all calls from the clients to API methods to extract information about the type of arguments passed to API methods. We call this information *argument type observations*. The second step is to search for anomalies in the

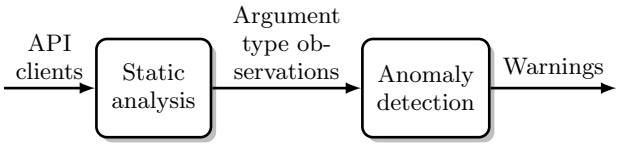

Figure 3: Overview of the approach.

argument type observations. Based on the assumption that most of the observations correspond to correct API usages, this step identifies calls where the observed argument type suggests an incorrect API usage. An anomaly occurs when (i) the API method has a brittle parameter, and (ii) the client passes an argument of an unexpected type.

2.1 Argument Type Observations

The goal of the first analysis step is to extract information about the types of arguments that clients pass to API methods. We represent this information as argument type observations.

Definition 1 (Argument type observation)
An argument type observation is a tuple $(m_{client}, line, m_{API}, pos, type)$*, where:*

- m_{client} *is the signature of the client method that calls the API method,*
- *line is the source code line of the call to the API method,*
- m_{API} *is the signature of the called API method,*
- $pos \in \mathbb{N}$ *is the position of the argument in the list of arguments passed to* m_{API} *(starting at 1), and*
- *type is a type that the argument at position pos may have.*

2.1.1 Points-to Analysis

To extract precise argument type observations, our analysis leverages points-to information obtained from a state of the art points-to analysis [9]. The points-to analysis statically reasons about the objects that may occur at runtime of a program and about the references that may point to each object. Potential runtime objects are represented by *abstract objects*. We access the results of the points-to analysis with a function $P2A$ that, given a reference r, returns the set $P2A(r)$ of abstract objects to which r may point. Each abstract object has an associated type. Unless specified otherwise, we use a context-insensitive points-to analysis with on the fly call graph construction. While context-sensitive analysis could provide even more precise argument type observations, we build upon a context-insensitive analysis to ensure that the approach scales to large programs.

Using a points-to analysis increases the precision of the extracted argument type observations, and therefore, the overall precision of our approach. However, the main idea of this work is independent of points-to analysis, and it can be pursued without any points-to information. In Section 3.7, we compare the results of our approach with and without points-to information.

Algorithm 1 Extract argument type observations from an API client.

Input: Program P that uses API A
Output: Set of argument type observations $\mathcal{O}$

```
1: O ← ∅
2: for all c ∈ allCalls(P) do
3:   if caller(c) ∈ P ∧ callee(c) ∈ A then
4:     G ← arguments(c)
5:     for pos ← 1, |G| do
6:       r ← reference(G(pos))
7:       if P2A(r) ≠ ∅ then
8:         for all o ∈ P2A(r) do
9:           obs ← (caller(c), line(c), callee(c), pos,
                    type(o), 1/|P2A(r)|)
10:          O ← O ∪ {obs}
11:        end for
12:      else
13:        obs ← (caller(c), line(c), callee(c), pos,
                  type(r), 1)
14:        O ← O ∪ {obs}
15:      end if
16:    end for
17:  end if
18: end for
```

2.1.2 Extraction Algorithm

Algorithm 1 summarizes how the analysis extracts argument type observations from a program. The analysis visits each method call in the program. If the callee of a call is an API method, it will be further analyzed (line 3). Calls to client types that inherit from API types are analyzed if the declaring type of the callee is an API type.

For a call to an API method, the algorithm analyzes each of its arguments. An argument is represented by a reference (line 6), which, for example, corresponds to a local variable or a field. The analysis is performed on an intermediate program representation that has an explicit reference for each argument, even if there is no such reference in the source code. For example, the statement `m2(m1())`, which passes the return value of a call to `m1()` as an argument to `m2()`, is represented by storing the result of `m1()` into a fresh local variable, and afterwards passing this variable to `m2()`.

For each argument reference r, the analysis checks whether the points-to analysis knows any abstract objects that r may point to (line 7). If so, then the analysis creates an argument type observation for each abstract object o that r may point to. The observation states that the argument passed to the API method can have the type $type(o)$, that is, the most specific type that the points-to analysis knows for o. If r may point to multiple abstract objects, the analysis creates a separate observation for each abstract object. Doing so naively can lead to many observations for a single call site, giving this call site a higher weight than other call sites. Since we want to give each call site the same weight when analyzing the API usage of a client, we extend the definition of argument type observations by adding a confidence value:

Definition 2 (Argument type observation, extended)
An argument type observation is a tuple $(m_{client}, line, m_{API}, pos, type, conf)$*, where:*

- m_{client} *is the signature of the client method that calls the API method,*

```
1  class API {
2    void m(Object o, Component c) { .. }
3  }
4
5  class Client {
6    API api;
7    void n() {
8      Foo f = new Foo();
9      Component c;
10     if (..) c = new JLabel();
11     else c = new Button();
12     api.m(f, c);              // call to API method
13   }
14 }
```

Listing 1: Example of extracting argument type observations.

- *line is the source code line of the call to the API method,*
- m_{API} *is the signature of the called API method,*
- $pos \in \mathbb{N}$ *is the position of the argument in the list of arguments passed to* m_{API} *(starting at 1),*
- *type is a type that the argument at position pos may have, and*
- $conf \in [0, 1]$ *indicates the confidence that the argument has this type.*

When creating multiple observations for a single argument reference r that may point to different abstract objects, the analysis sets the confidence of each such observation to $\frac{1}{|P2A(r)|}$. That is, the more types the analysis observes for a single call site, the less confidence it has into each individual observation. By dividing the confidence, the analysis gives the same weight to all call sites in the program.

The points-to analysis may not know any abstract object for an argument reference r (line 12). For example, this happens if r is a parameter obtained by the caller method $caller(c)$ and if there is no known call site where $caller(c)$ is called. However, $caller(c)$ may nevertheless be called, for example, in source code that is not part of the analyzed code base, such as sub-projects of a project or external plug-ins. Therefore, we also analyze calls where no abstract objects are known for the arguments. In this case, the analysis considers the declared type of the argument and creates an observation for this type (line 13). This observation has confidence one, because the analysis makes a single observation for the call site.

2.1.3 Example

We illustrate extracting argument type observations with the simple example in Listing 1. Class `Client` calls an API method and passes two arguments (line 12). The analysis extracts three observations from this call:

- $(Client.n(), 12, API.m(Object, Component), 1, Foo, 1)$
  This observation describes that the first argument for the API method has been observed to be of type `Foo`. Since this is the only possible type for this argument, the observation has confidence one.
- $(Client.n(), 12, API.m(Object, Component), 2, JLabel, 0.5)$
  This observation describes that the second argument passed to the API method has been observed to be of type `JLabel`. The local variable `c` may point to two abstract objects, which have type `JLabel` and `Button`, respectively. Therefore, the analysis splits the confidence and assigns confidence 0.5 to this observation.
- $(Client.n(), 12, API.m(Object, Component), 2, Button, 0.5)$
  This observation is similar to the previous observation, but for the argument type `Button`.

For the last two observations, the analysis relies on points-to information. Without it, the observed argument type for reference `c` is `Component` and the last two observations would be merged into a single observation with confidence one.

Algorithm 2 Find anomalies in argument type observations.

Input: Sets of argument type observations $\mathcal{O}_1, \ldots, \mathcal{O}_n$
Output: Set of warnings $\mathcal{W}$

1: $\mathcal{O}_{raw} \leftarrow merge(\mathcal{O}_1, \ldots, \mathcal{O}_n)$
2: $\mathcal{O} \leftarrow preprocessObs(\mathcal{O}_{raw})$
3: $M_{raw} \leftarrow param2Obs(\mathcal{O})$
4: $M \leftarrow preprocessParams(M_{raw})$
5: $\mathcal{W} \leftarrow \emptyset$
6: **for all** $(p, \mathcal{O}_p) \in M$ **do**
7: $\quad T \leftarrow histogram(\mathcal{O}_p)$
8: $\quad$ **if** $|\mathcal{O}_p| \geq \theta_{obs} \wedge |dom(T)| \leq \theta_{types}$ **then**
9: $\quad\quad \mathcal{T}_{deviant} \leftarrow \emptyset$
10: $\quad\quad$ **for all** $t \in dom(T)$ **do**
11: $\quad\quad\quad conf_{incl} \leftarrow \frac{|\mathcal{O}_p|}{|dom(T)|}$
12: $\quad\quad\quad conf_{excl} \leftarrow \frac{|\mathcal{O}_p| - T(t)}{|dom(T)|-1}$
13: $\quad\quad\quad$ **if** $conf_{excl} - conf_{incl} \geq \theta_{conf}$ **then**
14: $\quad\quad\quad\quad \mathcal{T}_{deviant} \leftarrow \mathcal{T}_{deviant} \cup \{t\}$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end for**
17: $\quad\quad$ **if** $\frac{\sum_{t \in \mathcal{T}_{deviant}} T(t)}{\sum_{t \in dom(T)} T(t)} \leq \theta_{deviant}$ **then**
18: $\quad\quad\quad \mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{O}_p$
19: $\quad\quad$ **end if**
20: $\quad$ **end if**
21: **end for**

2.2 Detecting Anomalies

The second step of the approach, anomaly detection, has two goals. First, we want to infer from argument type observations whether a method has brittle parameters. Second, we want to reveal calls to methods with a brittle parameter where the caller passes an argument of an unexpected type. As input, the anomaly detection takes sets of argument type observations, each obtained from a different API client. As output, it produces a set of warnings about observations of unexpected arguments. Algorithm 2 summarizes the anomaly detection.

2.2.1 Preprocessing Argument Type Observations

As a first step, the analysis merges the argument type observations from different API clients (line 1 of Algorithm 2). The *merge* function generalizes argument type observations to make them more comparable and then computes the union of the sets of observations. We generalize observations referring to client-specific types that are subtypes of API types.

```
1  interface APIItf1 { .. }
2  interface APIItf2 extends APIItf1 { .. }
3  interface APIItf3 extends APIItf1 { .. }
4  class API {
5    void m(APIItf1 i1) { .. }
6  }
7
8  class ClientType implements APIItf2, APIItf3 { .. }
9  class Client {
10   API api;
11   void n() {
12     api.m(new ClientType());   // call to API method
13   }
14 }
```

Listing 2: Example of generalizing observations by replacing client-specific types with their API supertypes.

The analysis checks for each observed, client-specific argument type t_{arg} whether it has a supertype t_{API} that is defined in the API and that is compatible with the type declared by the callee. If such a supertype exists, the analysis adapts the observation by replacing t_{arg} with the supertype t_{API}. If t_{arg} has multiple most specific API supertypes, the analysis replaces the observation with a set of observations, one for each most specific API supertype. In this case, the confidence of the new observations is the confidence of the old observation divided by the number of new observations.

For example, consider Listing 2. The analysis extracts the following observation:

$$(Client.n(), 12, API.m(APIItf1), 1, ClientType, 1)$$

From this observation, the analysis cannot draw any conclusions about other clients, because the argument type `ClientType` is client-specific. However, `ClientType` implements the API interfaces `APIItf2` and `APIItf3`, which both are compatible with the declared parameter type `APIItf1`. In this case, the analysis splits the observation into two new observations:

$$(Client.n(), 12, API.m(APIItf1), 1, APIItf2, 0.5)$$

$$(Client.n(), 12, API.m(APIItf1), 1, APIItf3, 0.5)$$

These generalized observations are useful for analyzing other API clients, because they refer to API types.

After merging observations from different API clients, the analysis removes observations from which we cannot infer any information (line 2). The *preprocessObs* function removes all observations where the declared parameter type is a primitive type, because primitive types have no subtypes in Java.

Next, the analysis groups observations by the declared parameter to which they refer (line 3). A declared parameter is defined by a callee and a position in the list of parameters of this callee. The map M_{raw} assigns to each parameter the set of observations made for the parameter. While considering each pair of callee and argument position as a parameter works reasonably well, we refine the notion of a parameter to consider overloaded API methods that allow clients to pass arguments of unexpected types. For example, consider the following API class:

```
class API {
  void m(Object o) { .. }
  void m(Foo o) { .. }
}
```

The overloaded method `m()` allows clients to pass any argument with a type being a subtype of `Object`. That is, from the client's perspective both methods can be considered a single method `m(Object)`. The analysis considers such cases by merging all parameters that are indistinguishable from the client's perspective into a single parameter. In the example, observations for both `m(Object)` and `m(Foo)` end up in a single group that refers to a parameter of type `Object`.

After grouping observations by the parameter they refer to, the analysis removes parameters where only a single argument type is observed. For such parameters, the analysis cannot reveal any unexpected arguments, because either all observed arguments are expected or all observed arguments are unexpected. If all arguments are unexpected, the analysis cannot find the problem, because there are no observations to learn from.

2.2.2 Type Histograms

The main part of the anomaly detection iterates over all parameters and their respective observations (line 6). For each parameter, the analysis builds a *type histogram* T showing how often each argument type is observed. T maps a type t to the summed up confidence values of all observations with argument type t.

Figure 4 shows four examples of type histograms extracted during the evaluation of this work. The examples illustrate kinds of histograms that occur again and again, allowing us to discuss how the anomaly detection should behave for each of them.

Figure 4a shows two frequently occurring types, `String` and `GridBagConstraints`, that are given to the second parameter of `Container.add(Component,Object)`. This parameter allows for specifying layout constraints for the component that is added to a container via the first argument. The expected type of the second argument depends on the layout the container uses. The layouts defined in Swing either expect a `String` describing constraints, or an instance of `GridBagConstraints`, that is, exactly the types prevalent in the histogram. The anomaly detection should not report any warnings for this histogram, because all observed argument types are expected by the callee.

Figure 4b is the histogram for `JMenu.add(Component)`, which is discussed in the introduction. The declared parameter type has many subtypes, but only a small number of argument types is observed. These observed argument types correspond to the expected types specified in the API documentation (Figure 1). In addition to the expected types, two argument types, `JCheckBox` and `JTextField` are observed a single time each. These argument types correspond to bugs. The anomaly detection should identify these anomalies and report warnings for them.

Figure 4c is an example for a "long-tail" histogram, where many different types occur as arguments. The parameter type has various subtypes, and they are all expected by the method. That is, the parameter is not brittle, and therefore the analysis should not report any warnings. It is a challenge to distinguish this kind of histogram from histograms such as Figure 4b.

Figure 4d illustrates a parameter with relatively few observations. In this case, the analysis should not draw any conclusions and should report no warnings.

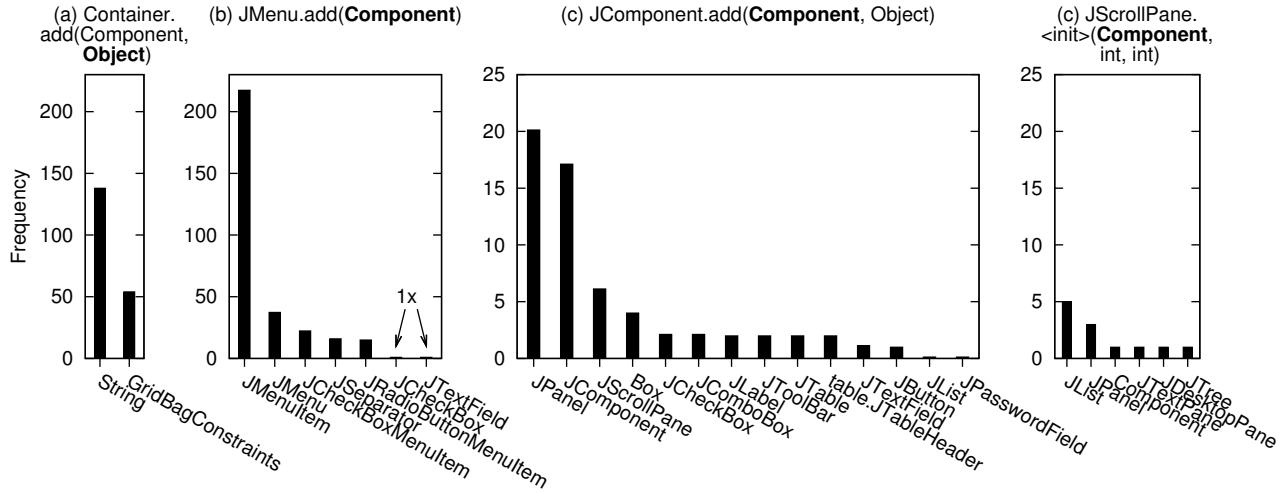

Figure 4: Four type histograms showing the frequency of argument types observed for a particular parameter.

2.2.3 Identifying Anomalies

Given type histograms such as those in Figure 4, how can an analysis find unexpected arguments given to brittle parameters while reporting as few false warnings as possible?

The analysis starts with the assumption that each observation is a potential anomaly and applies four filters to remove observations that would be false warnings. An observation that passes all filters results in a warning reported to the developer. In the following, we explain the four filters. Each filter has a threshold deciding which observations to filter. In Section 3.6, we discuss and compare values for these thresholds.

Minimum Number of Observations. The analysis ignores all parameters with a number of observations smaller than a threshold θ_{obs} (line 8 of Algorithm 2). This filter avoids drawing conclusions from a small number of observations, such as the histogram in Figure 4d. Setting θ_{obs} too high removes valid warnings for parameters with a fair number of observations, while choosing a value θ_{obs} that is too low leads to false warnings.

Maximum Number of Types. The analysis ignores all parameters for which the number of observed argument types exceeds a threshold θ_{types} (line 8). The rationale behind this filter is that parameters for which a large number of types is observed often are not brittle, and that instead all subtypes of the declared parameter type are expected. In particular, this filter removes long-tail histograms, such as Figure 4c. Setting θ_{types} too low removes valid warnings, while setting it too high introduces false warnings.

Minimum Confidence Drop. The analysis computes a confidence value indicating how confident it is that a given type histogram describes a brittle parameter. This confidence is the summed up confidence of all observations in the histogram divided by the number of observed argument types. If the confidence drops when adding a particular argument type to a histogram, then this argument type deviates from an otherwise accepted rule. The analysis compares the confidence with and without a type t for each type t in a histogram and ignores all types where the confidence drop is below a threshold θ_{conf} (line 13). Setting θ_{conf} too high removes valid warnings, while setting it too low leads to false positives.

Maximum Percentage of Anomalies. The analysis removes all warnings for a parameter if the percentage of supposedly noteworthy observations for this parameter exceeds a threshold $\theta_{deviant}$ (line 17). The rationale for this filter is to assume that most observations correspond to correct API usage. Setting $\theta_{deviant}$ too high removes valid warnings, while setting it too low leads to false warnings.

2.2.4 Clustering Warnings

The final step of the analysis is to cluster the warnings produced by Algorithm 2 to ease their manual inspection. A cluster of warnings contains all warnings for a particular parameter. We present warnings to developers in an interactive way, where the list of warnings to inspect depends on the developer's decisions on previous warnings. Confronted with a warning, the developer can indicate whether the parameter is brittle. If so, then all calls that pass arguments to this parameter are at risk to pass an unexpected argument without a notice from the type system. In this case, the analysis presents all warnings in the cluster, because they may correspond to bugs. Otherwise, the analysis knows that the cluster contains nothing but false warnings and therefore omits further warnings from the cluster.

3. EVALUATION

We evaluate our approach by analyzing 21 programs and their usage of the Java Swing API. The evaluation focuses on the following questions:

- *How effective is the approach in finding arguments of unexpected types?*
  The analysis reveals 15 previously unknown bugs and code smells. In its default configuration, 47% of all reported warnings are relevant problems. To measure how many unexpected arguments the analysis misses, we seed bugs into programs. The analysis detects 83% of them.
- *Does the approach scale to real-world programs?*
  Analyzing all 21 programs (650 KLoC) takes 23 minutes.

Table 1: Programs used for the evaluation.

Program	Lines of code		Argument
	All	API-rel.	type obs.
ArgoUML 0.34	156,305	79,950	7,726
Class Editor 2.23	10,121	5,643	1,796
Dublin Core Ousia 11-01-11	6,724	3,147	583
Figoo 2.6.0	12,911	11,965	5,758
File Renamer 0.1.1	1,328	726	236
FormLayoutMaker 8.2.1rc	4,239	4,152	814
hirudo 0.7	2,642	1,569	518
id3tidy 0.3beta	2,097	829	294
jEdit 4.5pre1	103,332	67,637	7,441
JFreeChart 1.0.14	93,460	66,733	7,039
JFtp 1.53	23,511	16,180	2,958
JGraph 1.9.0.2	45,768	28,484	2,135
JPropsEdit 1.0.2	3,374	3,005	883
myPomodoro 1.0	2,510	1,960	506
nTorrent 0.5.1	36,290	4,949	901
outliner 1.8.10.6	35,407	23,434	3,499
pdfsam 2.2.1	15,567	12,197	3,933
Protégé 3.3	68,383	29,812	3,146
Scrinch 1.1.1	13,122	9,193	3,677
Stringer 1.0beta1	8,947	5,438	1,366
uBlogger 20090914	3,206	2,922	1,024
Sum	649,244	379,925	56,233

- *What is the influence of the thresholds used for detecting anomalies?*
  We run the analysis with different values for each threshold, discuss their trade-offs, and propose a default configuration.

- *What is the influence of the points-to analysis?*
  Comparing the approach with and without points-to analysis shows that points-to analysis increases precision, but that it is not crucial for the approach.

3.1 Implementation

We implement our approach into a practical tool for analyzing Java programs. The implementation of the static analysis is based on the Soot framework 2.5.0 [22] and its implementation of the Paddle points-to analysis [9]. We enable Soot's option to consider all methods of program classes to be reachable and exclude classes in third-party libraries from the analysis.

3.2 Experimental Setup and Measurements

Table 1 lists the programs used in the evaluation. For each program, we give the number of non-comment, non-blank lines of Java source code, in total 650 KLoC. The third column shows an estimate of the number of lines of code related to the Swing API. We estimate this number by counting the source code lines of all classes that import from the `java.awt` or `javax.swing` packages. The last column shows how many argument type observations the analysis extracts from each program. In total, the analysis extracts 56,233 observations.

We inspect warnings manually and classify them into three categories [23, 7, 16]. *Bugs* are problems in the API usage that affect the correctness of the program. Since we focus on the Swing API, the bugs we find typically lead to visual glitches or they disable some functionality. *Code smells* are problems that affect performance or maintainability of the program but not its correctness. We say *true positives* to refer to both bugs and code smells. All other warnings are *false positives*.

Table 2: Classification of warnings in real programs.

Configuration	Warnings	Bugs	Smells	True pos.
Default	19	5	4	47%
Recall-focused	155	11	4	11%

```
1  class FilteredListModel extends AbstractListModel {
2    void setFilter(String filter) {
3      Runnable runner = new Runnable() {
4        public void run() {
5          // .. update internal state ..
6
7          // BUG: first argument must be a ListModel
8          fireContentsChanged(this, 0, getSize() - 1);
9        }
10     };
11     SwingUtilities.invokeLater(runner);
12   }
13 }
```

Listing 3: Bug in jEdit.

We quantify the effectiveness of the analysis by measuring precision and recall. *Precision* means the percentage of true positives among all reported warnings. *Recall* is the percentage of true positives that the analysis reports among all true positives in the program. Since we do not know all true positives in the analyzed programs (if we had a practical way to find them, this paper would be obsolete), we only report the recall of seeded bugs, where we know by construction where problems in a program reside.

The anomaly detection allows for controlling the trade-off between precision and recall with threshold parameters. In Section 3.6, we evaluate the influence of each threshold. For evaluating the effectiveness of the approach in finding anomalies, we use two configurations:

- *Default.* This configuration is a pragmatic compromise between maximizing true positives and minimizing false positives. The parameters are (using the notation from Algorithm 2) $\theta_{obs} = 30$, $\theta_{types} = 6$, $\theta_{conf} = 10$, and $\theta_{deviant} = 0.05$.

- *Recall-focused.* This configuration offers the possibility to reveal more true positives than the default configuration, but leads to significantly more false positives. The parameters are $\theta_{obs} = 25$, $\theta_{types} = 10$, $\theta_{conf} = 1$, and $\theta_{deviant} = 0.5$.

3.3 Anomalies in Real Programs

The analysis finds 15 previously unknown bugs and code smells in the programs from Table 1. Some of them have already been fixed as a result of our bug reports. Table 2 shows the number of reported warnings and their classification for both configurations. The default configuration gives a true positive rate of 47%, that is, about half of the reported warnings are relevant for a developer. The recall-focused configuration reveals six bugs that are not found in the default configuration. The price for finding these additional bugs is a lower true positive rate.

```
1  class InsomniacClient {
2    JPanel p = new JPanel();
3    JList list = new JList();
4    InsomniacClient() {
5      // ...
6      JScrollPane pane = new JScrollPane(list);
7      // BUG: Two nested scroll panes
8      p.add(new JScrollPane(pane));
9    }
10 }
```

Listing 4: Bug in JFtp.

Listing 3 shows a bug that the analysis finds in jEdit. The first parameter of `fireContentsChanged()`, called in line 8, is brittle: The declared parameter type is `Object`, but the documentation clearly states that it must be a `ListModel`. Our analysis infers this constraint from calls to this method and reports a warning because the first argument in Listing 3 is of type `Object`. The problem is that the programmer passes `this`, which refers to the `Runnable` and not to the surrounding class `FilteredListModel`. We reported this bug to the developers and they fixed it within a single day.[3]

Listing 4 is a bug found in JFtp.[4] The program wraps a list into a scroll pane to add scroll bars to it (line 6) and subsequently wraps this scroll pane into another scroll pane (line 8). The result are scroll bars surrounded by scroll bars—certainly undesired behavior. The analysis finds this problem because `JScrollPane` occurs only once as argument type of `JScrollPane`'s constructor, while several other types occur frequently.

The analysis finds a bug in nTorrent, which is illustrated in Figure 2. It has been confirmed as a bug in response to our bug report.[5] Five bugs (in Protégé, jEdit, and ArgoUML) are due to using look and feel-specific color classes, which lead to visual glitches if the programs run with another look and feel than expected by the programmers. The analysis finds these bugs because the look and feel-specific classes appear as argument types instead of the much more common type `Color`. Two bugs (in JFreeChart and Scrinch) pass unexpected arguments to `Graphics2D.setClip(Shape)`. According to the documentation only particular subtypes of `Shape` will lead to the expected behavior. Another bug (in jEdit) is a `JTextField` added to a `JMenu`, which is unexpected as shown in Figure 1.

The four code smells found by the analysis affect performance and maintainability. Two warnings (in JPropsEdit) are about passing a newly created `JLabel` as a message to `JOptionPane.showMessageDialog()`. The API documentation states that messages are converted to `String`s by the API implementation and afterwards wrapped into a `JLabel`. A client cloning this behavior creates useless labels. The analysis warns about Protégé passing a `JComponent` to a `BoxLayout`. The underlying problem is that a class in Protégé extends `JComponent` instead of the typically extended `JPanel`. Finally, we find a problem in Scrinch, where a `StringBuffer` is passed as the message of a dialog. Since the programmer obviously does not want the API to modify the string, passing a `String` instead of the mutable `StringBuffer` would be safer.

The false positives reported by the analysis have two main reasons. First, we get false positives because some argument types that occur infrequently are nevertheless correct. For example, the analysis extracts 77 observations for `GroupLayout.setHorizontalGroup(Group)`: 76 observations with `ParallelGroup` as argument type and a single observation with `SequentialGroup` as argument type. The parameter is not brittle, that is, both argument types are expected, but the analysis cannot distinguish this case from cases like Figure 4b. Second, we get false positives because the static analysis extracts imprecise observations. For example, the first argument of `ActionMap.put(Object,Action)` is typically a `String`, but the analysis warns about code where an `Object` is observed. Manual inspection of the source code shows that the argument will be a `String` for all possible program paths, but the static analysis fails to find it.

In summary, we find that errors related to brittle parameters exist in practice and that the analysis is effective in finding them. Many of the bugs that the analysis reveals are hard to detect with traditional testing techniques. These bugs do not raise an exception or trigger any other obvious misbehavior. Instead, many bugs lead to malfunctions of the user interface or to incorrectly displayed GUI elements.

3.4 Automated Evaluation with Seeded Bugs

In addition to evaluating the effectiveness of the analysis in finding real bugs, we seed bugs into programs. With seeded bugs we know by construction where problems in a program reside and do not require a human to inspect warnings. Seeding bugs not only allows us to evaluate the analysis with a large number of anomalies, but also to measure the recall of the analysis.

To seed bugs related to brittle parameters, we must know about argument types that are not expected by a callee but nevertheless compatible with the declared parameter type. To find those types, we manually search the documentation of the Swing API for descriptions of brittle parameters. The result is a list of 14 API methods, each declaring a parameter of type T but expecting a proper subset of T's subtypes as argument. Based on the list B of known brittle parameters, we seed bugs into the programs from Table 1. For each program P, we repeatedly do the following:

1. Randomly select a parameter p from B. The probability to choose parameter p is proportional to the number of calls of p's API method in P.

2. Randomly select an argument type t from all types that are compatible with p but that are not among the expected types for p. The probability to choose a type t is proportional to the number of references with type t in P.

3. Add an argument type observation stating that t is passed to p.

4. Run the analysis to check whether it finds the seeded bug.

The seeding technique chooses API methods and argument types according their frequency in the analyzed program to simulate errors that programmers could make.

[3]See issue 3477759 in jEdit's bug database.
[4]See issue 3484625 in JFtp's bug database.
[5]See issue 136 in nTorrent's bug database.

For each seeded bug, we run the analysis and measure its precision and recall:

$$\text{precision} = \frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false positives}}$$

$$\text{recall} = \begin{cases} 1 & \text{if the seeded bug is found} \\ 0 & \text{otherwise} \end{cases}$$

The number of true positives is one if the analysis finds the seeded bug and zero otherwise. Based on the assumption that the programs are correct except for the seeded bug, all other reported warnings are considered to be false positives. We make this assumption more realistic by ignoring the 15 known bugs and code smells described in Section 3.3. Yet, the measured precision is an under-approximation, because some of the warnings we count as false positives may in fact be true positives that we did not inspect manually.

We seed 100 bugs into each program from Table 1 and compute the average of the results from all runs of the analysis. In the default configuration, the analysis has a precision and a recall of 83%. The recall-focused configuration raises recall to 94% but reduces precision to 11%. These results show that the analysis reveals most of the seeded bugs, even in the default configuration.

3.5 Performance

On a standard PC (Intel Core 2 Duo with 3.16 GHz, using 2.5 GB of memory), analyzing all 21 programs takes 23 minutes. Analyzing the largest program, ArgoUML, takes eight minutes. Most of the time (over 99%) is spent extracting argument type observations. We consider these performance results to be acceptable for an automatic testing tool.

3.6 Thresholds of Anomaly Detection

Four thresholds control how strict the anomaly detection is when searching anomalies (Section 2.2.3). In the following, we analyze the sensitivity of the analysis to these thresholds and illustrate the trade-offs when selecting thresholds. We vary one threshold at a time, while leaving the others at default values. Initially, we varied all four thresholds and decided on the default configuration given in Section 3.2.

Figure 5 shows how precision and recall vary depending on each threshold. For each threshold, we give the results from analyzing the original programs (left) and from analyzing programs with seeded bugs (right). For seeded bugs, we report the F-measure, that is, the harmonic mean of precision and recall.

The minimum number of observations (Figure 5a) controls how many observations the analysis requires to give warnings. We experiment with values between two and 200. A high threshold leads to more warnings, increasing precision but decreasing recall. In contrast, a low threshold gives higher recall but a lower precision. The figure illustrates the typical trade-off between avoiding false positives and avoiding false negatives. Our default configuration is a compromise between both objectives.

The maximum number of types (Figure 5b) specifies the maximum number of different observed argument types for a parameter for which the analysis reports warnings. We experiment with values between two and 50 (Figure 5b shows only values up to 15 because there are no significant changes between 15 and 50). The figure shows that starting from a relatively small number (three for real bugs and four for seeded bugs), the threshold does not significantly influence

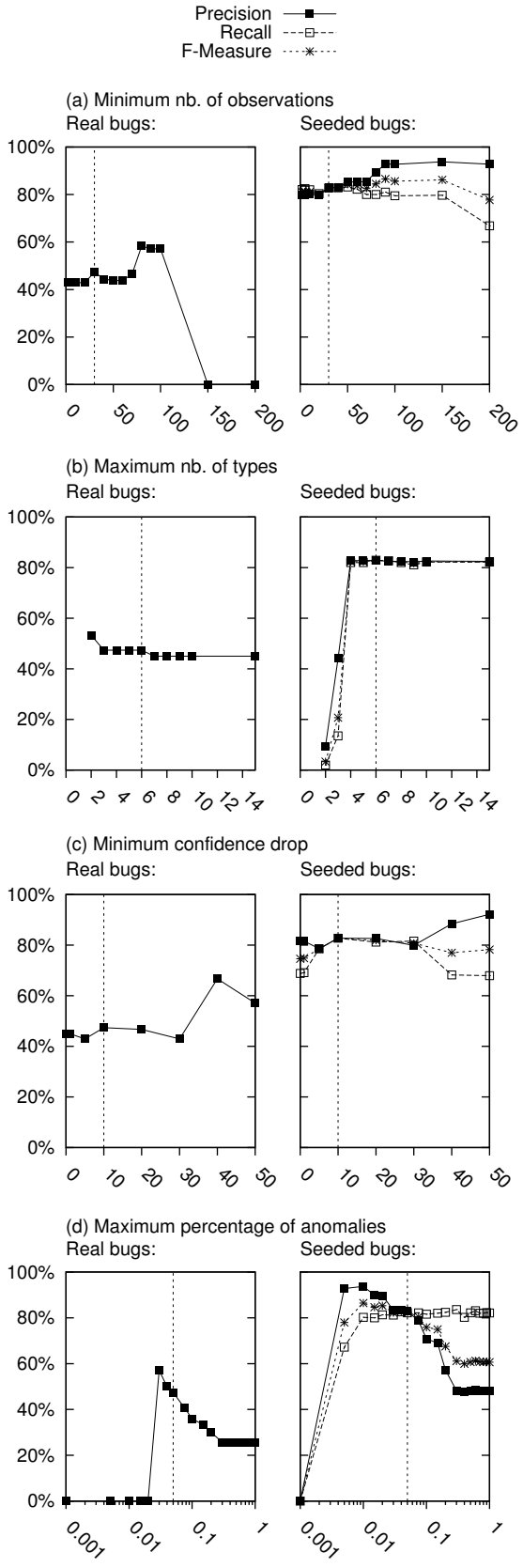

Figure 5: Influence of thresholds on anomaly detection. The dashed vertical line indicates the default configuration.

the results. Our default configuration is to allow up to six different argument types.

The minimum confidence drop (Figure 5c) controls how much the confidence in the brittleness of a parameter must decrease by adding an argument type to a histogram for the analysis to report a warning. We experiment with values between zero and 50. A small threshold leads to more warnings, and hence, higher recall but lower precision. In contrast, a larger threshold increases precision for the price of reducing recall. As a default, we select a threshold that maximizes the F-measure for seeding bugs.

The maximum percentage of anomalies (Figure 5d) limits the degree of incorrectness that the analysis expects in the programs. We experiment with values between 0.1% and 100%. For very small thresholds the analysis does not report any warnings. At some point (3% for real bugs and 0.5% for seeded bugs), the analysis begins to report warnings with high precision. Further increasing the threshold reduces precision while slightly increasing recall. We choose a default threshold that gives reasonable precision.

3.7 Influence of Points-to Analysis

Using a points-to analysis while extracting argument type observations has a significant performance impact. While points-to analysis can have benefits as illustrated by the example in Listing 1, it is unclear how these benefits manifest in practice. Therefore, we evaluate our approach without points-to analysis and compare the results to the approach as described in Section 2. To disable points-to analysis, the approach remains as in Section 2 except for the $P2A$ function, which always returns an empty set. That is, we assume that the points-to analysis never knows any abstract object that a reference points to, and therefore our analysis always considers the statically declared type of arguments.

The comparison shows that the precision of finding seeded bugs decreases from 83% to 76% when abandoning points-to information. In contrast, the precision of finding real bugs and the recall of seeded bugs remain the same. We conclude that points-to analysis has an influence, but that it is not crucial for the overall approach. Since the performance of our approach is reasonable when using a points-to analysis, we leverage the benefits of this analysis technique.

4. DISCUSSION

The problem of brittle parameter typing exists in various APIs. For example, the `Command` class in the Eclipse Platform/Core API has a method `compareTo(Object)` that expects a `Command` instance as the parameter. Another example is the Java XPath API: The `XPath` class provides a method `evaluate(String, Object)` that expects instances of `Node` as the second parameter. To adapt our analysis to an arbitrary API, it suffices to pass the appropriate API packages to Algorithm 1. The approach is particularly well-suited for APIs that have evolved over time, because this evolution often introduces brittle parameters. For such APIs, there typically exist various client programs to analyze, because otherwise there would be no reason to evolve the API.

To deploy our analysis, one could extract information on brittle parameters from many API clients and persist it in a knowledge base. This knowledge base can then be used to quickly check new API clients, which, once they reach some level of maturity, can contribute to the knowledge base.

5. RELATED WORK

The problem we address goes back to Liskov's substitution principle, which proposes that an instance of a subtype should behave in the same way as an instance of a supertype, when being used through the supertype's interface [11]. Statically-typed, object-oriented languages typically follow this principle and enforce it syntactically, for example, by restricting how subtypes can change method signatures. However, these syntactic restrictions do not guarantee that subtype instances are semantic substitutes for supertype instances, leading to the problem of brittle parameters.

Recent work on "related types" addresses the problem of calls that become trivial because the actual argument type is unrelated to the formal type of the receiver [24]. Their approach requires annotating API methods, whereas our analysis infers expected argument types. Another difference is that our approach can deal with expected argument types that are scattered over the type hierarchy, that is, argument types not precisely describable with a single supertype.

The "extract superclass" and "extract interface" refactorings [5] create new types that provide the common interface of a set of existing types. They can be used to remove brittle parameter typing by introducing a type that precisely describes the parameters expected by a method and by making this new type the declared parameter type. Unfortunately, this approach is only feasible if it is possible to add a new supertype to all the expected types, which is, for example, not true if an expected type is declared in another library. The "generalize type" refactoring [21] replaces the declared type of a variable or a parameter with a more general type to reduce coupling and to increase extensibility. The problem we address in this paper is the result of using too general types for method parameters.

Our work is an anomaly detection technique that reveals errors by searching parts of a program that deviate from the norm. Several other approaches following this idea have been proposed in the past. All of them address different kinds of bugs than errors related to brittle parameters. Engler et al. [3] propose a static analysis framework to identify and check system-specific rules based on rule-templates. Several approaches search for missing method calls by learning which calls usually happen in a particular context [10, 2, 19, 23, 15, 20, 14]. Other work finds violations of method call protocols through anomaly detection [6, 17]. Lu et al. search for concurrency-related errors by inferring access interleaving invariants [13] and variable access correlations [12]. Finally, we previously presented an anomaly detection technique to find bugs related to equally-typed method arguments that programmers may accidentally pass in an incorrect order [16]. All these approaches are complementary to this paper, because they target different kinds of bugs than this work.

Ernst et al. [4] and Hangal et al. [8] propose to extract invariants over variables and to use them for finding errors. Similar to our work, these approaches leverage the observation that variables are expected to have only a subset of all values possible according to their declared type. Our work differs by considering subtypes of declared reference types instead of concrete values of primitive types. Hangal et al. [8] propose a metric for the confidence that an invariant holds and use it for deciding whether to warn about a violation of the invariant. We adapt this metric to our anomaly detec-

tion and use it to filter observations based on a minimum confidence drop (Section 2.2.3).

APIs with multiple clients allow for cross-project analysis. Zhong et al. [25] mine API clients to derive recommendations of code snippets. In contrast, our approach reveals bugs in the analyzed client programs. Gruska et al. [7] analyze 6,000 projects with an anomaly detection technique. In previous work, we leverage API clients to extract API usage protocols and to check clients for protocol violations [18]. Both [7] and [18] cannot find bugs related to brittle parameters.

6. CONCLUSIONS

Methods that expect fewer argument types than those that are compatible with the declared parameter type pose a risk at its callers. If a caller passes an argument of an unexpected type, this error slips through the checks of the type system. Complex APIs often have methods with brittle parameters. Yet, to the best of our knowledge, no existing technique searches bugs related to such methods. In this paper, we propose a simple yet effective static analysis that warns developers about unexpected arguments given to brittle parameters. The approach finds various issues in real-world programs with a precision and a recall making it applicable in practice.

We draw three conclusions from this work. First, brittle parameter typing and the problems resulting from it exist in practice. Our results contain various examples of brittle parameters in a real-world API. Furthermore, we find that programmers are susceptible to errors related to brittle parameter typing. Second, an automatic bug finding technique can be effective despite being simple. Third, leveraging multiple clients of the same API reveals knowledge that may not be apparent from analyzing a single client, and this knowledge is useful for finding bugs.

Implementation and experimental results:

`http://mp.binaervarianz.de/issta2012`

7. ACKNOWLEDGMENTS

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

8. REFERENCES

[1] J. Bloch. *Effective Java (Second Edition)*. Addison-Wesley, 2008.

[2] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *ISSTA*, pages 163–173, 2007.

[3] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE T Software Eng*, 27(2):213–224, 2001.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE*, pages 15–24, 2010.

[7] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *ISSTA*, pages 119–130, 2010.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.

[9] O. Lhoták and L. J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM T Softw Eng Meth*, 18(1), 2008.

[10] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[11] B. Liskov. Data abstraction and hierarchy. In *OOPSLA*, 1987.

[12] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, pages 103–116, 2007.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[14] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *ECOOP*, pages 2–25, 2010.

[15] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, pages 383–392, 2009.

[16] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *ISSTA*, pages 232–242, 2011.

[17] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*, 2012.

[18] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE*, 2012.

[19] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE*, pages 283–294, 2009.

[20] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE*, pages 496–506, 2009.

[21] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, 2003.

[22] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, pages 125–135, 1999.

[23] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE*, pages 295–306, 2009.

[24] J. Winther and M. I. Schwartzbach. Related types. In *ECOOP*, pages 434–458. Springer, 2011.

[25] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343, 2009.