Users Beware: Preference Inconsistencies Ahead

Farnaz Behrang
College of Computing
Georgia Tech
Atlanta, GA, USA
behrang@gatech.edu

Myra B. Cohen
Dept. of Comp. Sci. and Eng.
Univ. of Nebraska-Lincoln
Lincoln, NE, USA
myra@cse.unl.edu

Alessandro Orso
College of Computing
Georgia Tech
Atlanta, GA, USA
orso@cc.gatech.edu

ABSTRACT

The structure of preferences for modern highly-configurable software systems has become extremely complex, usually consisting of multiple layers of access that go from the user interface down to the lowest levels of the source code. This complexity can lead to inconsistencies between layers, especially during software evolution. For example, there may be preferences that users can change through the GUI, but that have no effect on the actual behavior of the system because the related source code is not present or has been removed going from one version to the next. These inconsistencies may result in unexpected program behaviors, which range in severity from mild annoyances to more critical security or performance problems. To address this problem, we present *SCIC* (*Software Configuration Inconsistency Checker*), a static analysis technique that can automatically detect these kinds of inconsistencies. Unlike other configuration analysis tools, SCIC can handle software that (1) is written in multiple programming languages and (2) has a complex preference structure. In an empirical evaluation that we performed on 10 years worth of versions of both the widely used Mozilla Core and Firefox, SCIC was able to find 40 real inconsistencies (some determined as severe), whose lifetime spanned multiple versions, and whose detection required the analysis of code written in multiple languages.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Verification

Keywords

Configurable systems, software evolution

1. INTRODUCTION

Many modern software systems are highly customizable, providing a flexible environment for adding, removing and modifying functionality. In these systems, users typically have access to hundreds or even thousands of preferences that they can modify to form an individual configuration (or instance) of the application.[1] The mechanisms for the manipulation of preferences consist of multiple layers of access. These mechanisms can include (1) the user-interface menu, which normally gives access to only a subset of the available preferences, (2) the actual application's code, which usually has complete control over the existing configurations, and (3) direct access to a persistent database (often based on plain files) that contains the configurations and can be directly modified. At the user interface level, some systems also provide access to a fuller set of preferences reflecting all of those within the underlying database (*e.g.,* Firefox, through the `about:config` mechanism [23]). In addition, at the source code level, preferences are often set and manipulated using more than one programming language [23].

This multi-layered approach for specifying preferences has the potential to introduce inconsistencies between layers. This is especially true during software evolution, where modifications made in one layer might not be suitably reflected in the other layers involved. For example, a preference may be modifiable in the external user view, but the corresponding source code may not be present—either because it was never implemented or because it has been removed going from one version to the next. In this scenario, which was first described by Rabkin [32], users might believe that they are modifying the application's behavior, yet no changes will occur. To make things worse, this could all happen silently, without clearly observable failures occurring.

Although some of these inconsistencies may result in nothing more than mild annoyances for the users, some of them may also cause more serious security or performance problems. Consider, for instance, a security preference in Firefox that we discovered in our empirical evaluation. In earlier versions of Firefox, a user had the ability to set for how long the website passwords repository remained unlocked after the users entered their master password; that is, if a user had visited a website for which a password was stored in Firefox's repository within *n* minutes from entering the master password, Firefox would have automatically populated the password entry without asking the user to re-enter it. This preference was removed from the code of Version 4.0 of Firefox, yet the option was modifiable, with no effect, using the `about:config` mechanism until Version 34.0. Users accustomed to using this setting in Firefox, were therefore led to falsely believe that they were adding a level of security to their browser, whereas they were not.

Despite a large body of recent research on testing (and specifically regression testing) of configurable systems, most of the previous work focuses on the problem of better functional fault detection and assumes that the preference model is known (*e.g.,* [21,

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786869

[1]In this paper, we use the term *preference* to indicate a specific configuration option. A set of preferences and the value assigned to them therefore identifies and corresponds to a configuration.

25, 29, 30, 33, 41]). Some researchers have begun to extract or dynamically track configuration options from the source code (*e.g.,* [26, 27, 32, 36, 43]). However, most of these analyses are limited to single-language programs with a flat (*i.e.,* name-value) preference structure and have been evaluated on programs with a limited number of options. Therefore, they cannot be straightforwardly applied to the highly-configurable, complex programs that we target, which typically have an extremely large configuration space (*e.g.,* Firefox has about 2,000 options, without considering plug-ins) with a hierarchical (*i.e.,* tree-based) preference structure system and which tend to be written in multiple languages.

To address some of the limitations of these existing approaches, we propose *SCIC (Software Configuration Inconsistency Checker)*, a technique and tool for finding configuration inconsistencies in modern, complex, highly-configurable, multi-language software systems. More precisely, SCIC's goal is to detect inconsistencies between the way configurations are handled in different layers (*e.g.,* user interface and code), with a focus on program evolution—but the technique can also be used for a single version of a program.

SCIC uses a custom analysis for each individual programming language and merges the results before identifying mismatches. To assess whether a mismatch is related to changes performed during evolution, SCIC also checks its current results against its results for prior versions of the system. This approach allows us to plug-in different analyses (*e.g.,* for additional languages) and identify the lifespan of inconsistency occurrences. SCIC's analyses work both on flat and on tree-based preference structures. We have identified this latter type of structure in several modern systems, including both the Mozilla [9] and the OpenOffice [28] application suites.

To assess the effectiveness of SCIC, we applied it to a 10-year version history of the open-source Mozilla Core modules, which form the basis for the entire family of Mozilla applications; that is, any inconsistencies within this core will propagate to all applications. We also applied SCIC to the 35 Firefox versions that were released during the same time period. In this empirical evaluation, SCIC was able to find 40 real inconsistencies, whose lifetime spanned multiple versions, and whose detection actually required the analysis of code written in multiple languages. We found unique inconsistencies in both Mozilla Core and Firefox and reported them to the Mozilla team, which acknowledged some of the issues and is currently investigating them. In our evaluation, we also classified the preferences involved in these inconsistencies and found that they are of different types, ranging from performance to security, and can cause non-negligible issues.

The contributions of this work are:

- A technique for extracting configuration options and finding configuration inconsistencies in evolving, highly-configurable, multi-language, modern software systems. The technique is able to handle systems with a hierarchical preferences structure.
- An implementation of the technique that can handle complex preference structures and systems written using multiple languages (*i.e.,* C, C++, JavaScript, and various markup languages).
- An empirical study performed on numerous versions of a large, widely used code base that shows that our technique can find real configuration inconsistencies that can have problematic consequences for users. Our experimental data is available at `http://www.cc.gatech.edu/~orso/software/scic-data`.

The rest of this paper is structured as follows. We next present some background on preference systems. Section 3 motivates inconsistency checking. We present our technique in Section 4 and discuss its implementation in Section 5. We then present our empirical evaluation, in Section 6, and related work, in Section 7. Finally, we conclude and discuss future work in Section 8.

2. BACKGROUND

Most modern highly-configurable applications have different levels of user interfaces (UIs), through which users can modify application preferences. In these applications, the top level UI is typically the menu system. However, as previous studies show (*e.g.,* [23]), for many of these applications the menu options only account for a subset of the whole preference set. For instance, these studies reported that the Firefox menu provides only 126 out of more than 1900 available preferences—the versions of Firefox that we analyzed have more than 2,000 preferences. Other preferences that are not readily available to the users through menu options are typically accessible through an intermediate level UI. The Google Chrome browser, Firefox, and the Opera browser, for instance, all provide a settings page that is accessible by typing `about:flags`, `about:config`, and `opera:config` into the address bar, respectively. Users can search for a particular option on the resulting page, and once they find it, they can modify the current value, which will update the current application configuration as well as write the value to the persistent preference files.

At the lowest level, the preferences are mapped to source code variables that turn on or off particular pieces of code and control the specific preference behavior. The lowest level holds the truth for the application when it comes to system behavior, but most systems do not provide a direct mapping between the UI levels and the code level. Initialization modules in the application set the preference values at application startup and store these in memory (*e.g.,* in a hash table or some other database) during runtime [23]. Many applications also provide a dynamic API (the previously mentioned `about:config` uses this API) to manipulate preferences at runtime. These APIs interact with the source code preference variables.

In order to identify mismatches between the preferences in different levels of UIs and source code, we first need to obtain a complete set of preferences from both the source code and the UIs and then map the different levels to one another. Rabkin and Katz [32] studied seven open-source Java programs and concluded that using key-value configuration APIs is a common and widespread approach for programmers to interact with configuration options. Each configuration API takes the name of a configuration and an optional default value as its parameter. In [23], however, Jin and colleagues noted that this convention does not hold for some classes of highly-configurable applications, such as those we target in this work. An alternative way to handle preferences is to create a map from option names to values, but these options may be grouped in a hierarchy as a tree structure. Applications that use the Windows Registry, open source office suites such as OpenOffice and LibreOffice, and Mozilla-based applications are examples of systems that use this type of preference structure [23, 28, 32]. Figure 1 shows an example of a hierarchical tree structure in which related configurations share the same prefix. At the highest level, there is the *browser* prefix, to which we can append either *chrome* or *chromeURL*. The *browser.chrome* preference name can then be the prefix for either *favicons* or *site_icons*.

In a tree structured preference system, the argument that is passed as a preference name to a configuration API is not necessarily the exact configuration name. When the API interacts with the preference tree, there is a possibility that such interaction occurs on branches other than at the root. In this case, the configuration name is divided into two parts: (1) the branch where the interaction occurs and (2) the argument that is passed to the API.

To illustrate, consider the code snippet in Figure 2, which we extracted from Firefox Version 34.0. (Other tree-structured preference systems behave in a similar way, but may be implemented

```
+-- browser
|  |
| +-- chrome
|  |  |
|  | +-- favicons (browser.chrome.favicons)
|  |  |
|  | +--site_icons (browser.chrome.site_icons)
|  |
|  +--chromeURL (browser.chromeURL)
```

Figure 1: Example of tree structured configuration.

```
1451  #define NS_BRANCH_DOWNLOAD "browser.download."
        ...
1453  #define NS_PREF_DIR   "dir"
        ...
1467  nsCOMPtr<nsIPrefBranch> prefBranch;
1468  rv = prefService->GetBranch(NS_BRANCH_DOWNLOAD,
1469                    getter_AddRefs(prefBranch));
        ...
1497  prefBranch->GetComplexValue(NS_PREF_DIR,
1498              NS_GET_IID(nsIFile),
1499              getter_AddRefs(customDirectory));
```

Figure 2: Example of tree structure configuration in the code.

differently.) In this code, the configuration API is *GetComplex-Value* (line #1497). If we use the existing analyses to extract the preference from this code, the first parameter would be the configuration name, which is *dir* (line #1453). However, the correct configuration name consists of two parts: *browser.download* (line #1451), as the branch name, and *dir*, as the configuration API argument. We need to identify and concatenate these two parts to return the correct configuration name. This is a common way of implementing configurations because, when developers want to interact (read/write) with a group of preferences that share the same branches in the tree, they can retrieve the shared branch once, and use that to traverse the rest of the tree. In the context of the code shown above, preference *browser.download.folderlist* shares prefix *browser.download* with preference *browser.download.dir*. In our analysis, we support both flat and tree structured preference systems.

3. MOTIVATING EXAMPLE

Our motivating example is derived from a real bug report in the Bugzilla database. The report [3] is for the core modules of Mozilla, which means that the issue with the preference code is shared across the entire family of Mozilla applications and impacts more than just Firefox (the application mentioned by the user in this report). In the report, a user complains about preference (*browser.history_expire_sites*), which was (intentionally) removed from the source code during software evolution but still existed in the UI. This preference can be used to limit the number of URLs kept in a browser history. The user indicated in the bug report that, although he set the value of this preference to 40,000 in Firefox, he still had more than 61,000 sites in his history. This bug was later marked by the developers as invalid since the preference, having been removed, did not exist anymore, which triggered the following followup comment by the reporting user:

"It's hard to guess if something is a bug or an intended change. If this preference is no longer used:

- *it should be removed automatically (just like browser.history_expire_days)*
- *all relevant information should be updated*
- *you should make sure that users are aware of this change. E.g. by mentioning it in the release notes."*

Note that this particular issue might have impacted the privacy of users who were unaware that *browser.history_expire_sites* was no longer a valid preference and used it to limit the amount of history maintained. We investigated the cause of this issue and found the following code segment, extracted from Firefox Version 3.0 (file nsNavHistory.cpp), where preference *browser.history_expire_sites* was introduced.

```
#define PREF_BROWSER_HISTORY_EXPIRE_SITES
"history_expire_sites"
 ...
pbi->AddObserver(
PREF_BROWSER_HISTORY_EXPIRE_SITES,this,false);
 ...
if(NS_FAILED(mPrefBranch->GetIntPref(
    PREF_BROWSER_HISTORY_EXPIRE_SITES,
    &mExpireSites)))
    mExpireSites = EXPIRATION_CAP_SITES;
```

This code was removed in Version 4.0. Before removing the code, the *History* component itself was responsible for expiration management. However, due to drawbacks such as a lag in navigation and other performance issues, a new JavaScript (JS) component with an adaptive algorithm was introduced instead. This change resulted in the replacement of three history related preferences-*browser.history_expire_days*, *browser.history_expire_days_min*, and *browser.history_expire_sites*—with the single preference *places.history.enabled*. When these three preferences were replaced with the new preference, one would expect them to also be removed from the UIs. However, whereas *browser.history_expire_days* and *browser.history_expire_days_min* were correctly removed from the UI in the next release, *history_expire_sites* was not. This preference was actually removed from the UI later on, in Version 8.0 of the Mozilla core, and no longer exists in the Mozilla preference interface, which shows that it was indeed an inconsistency and was recognized as such by the developers.

With thousands of preferences, and multiple layers that have to be kept synchronized, providing and maintaining consistency between the different levels at which preferences are accessed is not a trivial task. During software evolution, in particular, new preferences might be added to a version, and some old preferences might be removed. If any inconsistency is introduced between the actual and available functional preferences, this is likely to affect the users of the system, or at least the ones that rely on those preferences. To make things worse, these inconsistencies are often hard to detect, especially when their effect is not immediately observable, which is often the case for performance or security related preferences. The browser-history inconsistency we just described is a typical example, as it may not be easy to catch without specifically checking the number of entries kept in the history.

4. OUR TECHNIQUE: SCIC

Figure 3 shows an overview of SCIC. For each new application version, SCIC first performs a static analysis to extract the preferences from the source code. This is done independently for each of the different programming languages contained in the application (leftmost part of the figure). Using language-dependent analyses allows us to (1) handle different combinations of programming languages in different systems and (2) add new language-specific analyses (or modify and improve individual ones) based on the target application.

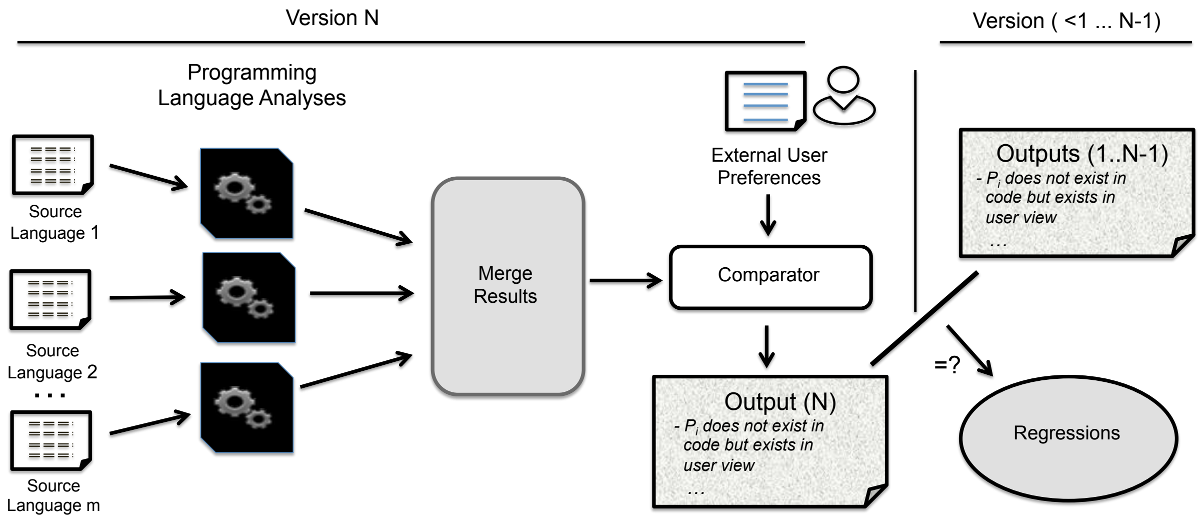
Version N
Version (<1 ... N-1)
Programming Language Analyses
Source Language 1
Source Language 2
...
Source Language m
Merge Results
External User Preferences
Comparator
Output (N)
- P_i does not exist in code but exists in user view
...
Outputs (1..N-1)
- P_i does not exist in code but exists in user view
...
=?
Regressions

Figure 3: Overview of SCIC.

Algorithm 1 Strategy to match preferences at different levels during software evolution.

```
    matchingPrefs()
 1: $highLevelPrefs \leftarrow extractHighLevelPrefs()$
 2: $sourceCodePrefs \leftarrow extractSourceCodePrefs()$
 3: $inconsistentPrefs \leftarrow newList < Pair < pref, version >>$
 4: for each $version$ in $availableSoftwareVersions$ do
 5:     for each $pref$ in $highLevelPrefs$ do
 6:         if $sourceCodePrefs$ contains $pref$ then
 7:             continue
 8:         else
 9:             $inconsistentPrefs.Add(Pair(pref, version))$
10:         end if
11:     end for
12: end for
```

Algorithm 2 Algorithm to extract configuration options.

```
    Auxiliary functions:
    constructCallGraph(): returns call graph
    getPref(c): returns the preference argument from method c
    getBranch(pref): returns the branch of configuration API
    concatenate(prefValues, branchValues): returns list of strings
    from joining two arguments

    extractSourceCodePrefs()
 1: $c \leftarrow constructCallGraph()$
 2: $confAPIs \leftarrow newMap <Method, ArgumentIndex>$
 3: $confAPIs \leftarrow identifyConfAPIs()$
 4: for each $c$ in $confAPIs$ do
 5:     $pref \leftarrow getPref(c)$
 6:     $prefValues \leftarrow findPrefValues(pref)$
 7:     $branch \leftarrow getBranch(pref)$
 8:     $branchValues \leftarrow newList <branchValue>$
 9:     $trackBranch(branch, branchValues)$
10:     $prefNames \leftarrow concatenate(prefValues, branchValues)$
11: end for
```

After performing these analyses, SCIC merges their results to form a single set of unique preferences and compares these merged results against the preferences that are available to users through higher level APIs. The UI preferences can be obtained either through static analysis (our approach) or using dynamic API methods that some applications provide. SCIC then reports inconsistent preferences, that is, preferences that are not in the source code, but are in the API. Finally, SCIC compares this report with those it produced for the prior (1, ..., N-1) versions of the system. This last step provides the inconsistencies' lifespan and allows for distinguishing new inconsistencies from inconsistencies that have already been identified. Algorithm 1 shows SCIC's approach to matching preferences.

As the algorithm shows, SCIC starts by extracting the high level preferences from the UI layer (line #1) and then extracts the preferences from the source code modules (line #2). On Line #3, it finds inconsistencies by checking whether each preference in the higher-level UI matches a preference in the source code using regular expression matching. If not, SCIC reports an inconsistency. As we just discussed, SCIC also identifies inconsistencies from all of the available prior versions of the software (Lines #4-12). The two main methods used in this algorithm are *extractHighLevelPrefs* and *extractSourceCodePrefs*. Method *extractHighLevelPrefs* extracts the UI preferences statically from the preference and configuration files.

4.1 Analysis Framework

Algorithm 2 describes the general analysis performed by method *extractSourceCodePrefs()*, which is called from Algorithm 1. This analysis builds on the work of Rabkin and Katz [32] by adding to it the ability to handle tree-structured preferences. The first step of the analysis is to identify the methods that read or write preferences (we call these methods configuration APIs). To achieve this, the analysis first constructs the static call graph (line #1), and then identifies the configuration APIs (lines #2-3). For many highly configurable systems, configuration APIs are used to interact with preferences. These are either stand-alone, documented APIs or are provided via specific classes that expose a set of methods to the rest of the program to enable interaction with configurations. There may also be other APIs that act as wrappers around explicit APIs. In order to extract all possible configurations, method *identifyConfAPIs* (line #3) accounts for both types of APIs. It examines each method in the call graph and checks if any of its arguments are

Algorithm 3 Algorithm to track branches.

```
  Auxiliary functions:
  getBranchDecl(branch): returns declaration of branch
  getBranchArgument(c): returns branch argument in call site c
  Add(branchValues, branchValue): adds an element to list

  trackBranch(branch, branchValues)
 1: $branchDecl \leftarrow getBranchDecl(branch)$
 2: if branchDecl is an argument of method m then
 3:     for each call site $c$ of method $m$ do
 4:         $branchArgument \leftarrow getBranchArgument(c)$
 5:         $trackBranch(branchArgument, branchValues)$
 6:     end for
 7: else
 8:     $branchValue \leftarrow findBranchValue(branch)$
 9:     $branchValues.add(branchValue)$
10: end if
```

passed to a configuration API. If so, it considers that method to be a configuration API as well.

Once it has collected all of the configuration APIs, SCIC proceeds as follows. For each API, it first retrieves the configuration argument, which is of type string (line #5). On line #6, method *findPrefValues* is responsible for finding the possible values of the configuration argument. Most of the configuration arguments are built as constants at compile-time. If the argument is passed explicitly as a string literal to the API, the method returns the string as the value of the configuration. Otherwise, it uses a points-to analysis to find possible values of the configuration. Sometimes parts of the configurations are built dynamically. For instance, if the branch name is `browser.contentHandlers.types. + nums[i] + "."`, where the value of nums[i] is decided at runtime, SCIC uses regular expression .*, instead of nums[i], to match any string at that point.

So far, SCIC has found the value of the API's configuration argument. However, as shown earlier, this does not necessarily mean that it has found the exact name of the configuration. Therefore, SCIC needs to check whether the retrieved configuration name starts from the root of the tree. If not, the algorithm has to find the corresponding branch with which the configuration API interacts (line #7). For instance, at line 1497 in the example of figure 2, the branch for *GetComplexValue* configuration API is *prefBranch*. Branches do not always get generated or modified at the call site of their corresponding configuration API, as it happens in the example in figure 2. It is possible for them to be passed as an argument as well. Method *trackBrach* on line #9, which is shown in Algorithm 3 (see below), tracks the branch at different call sites until it finds the earliest point in the program where the branch gets generated or modified. Then, the value of the branch is found on those particular call sites. This approach allows the analysis to use context-sensitivity on demand, only for the relevant call sites, rather than for the whole program. After finding the possible values of the branch, since SCIC already has the possible values of configuration arguments, the algorithm calls method *concatenate* to find the configuration names (line #10).

Algorithm 3 illustrates the general technique used by SCIC to find branches and their values. For every branch, it first finds the declaration of the branch (line #1). Then, it checks whether the declaration is an argument of its method. If so, for every call site of the method it recursively calls itself until it finds the call site where the value of the branch is generated (lines #2-7). This is the point where method *findBranchValue* is called to find the value of the

Algorithm 4 Strategy to find branches.

```
  Auxiliary functions:
  getBranchDecl(branch): returns declaration of branch
  findArg(m, N): finds Nth argument of method m
  findBranchValue(branch)
 1: $DefUse \leftarrow constructDefUseChains()$
 2: $branchDecl \leftarrow getBranchDecl(branch)$
 3: if $CPP$ then
 4:     $branchIsRoot \leftarrow true$
 5:     $uses \leftarrow DefUse(branchDecl)$
 6:     for each $use$ in $uses$ do
 7:         if $use$ is an argument of $GetBranch$ then
 8:             $branchArg \leftarrow findArg(GetBranch, 1)$
 9:             $values \leftarrow findValuesUsingPointsTo(branchArg)$
10:             $branchIsRoot \leftarrow false$
11:         end if
12:     end for
13:     if $branchIsRoot$ then
14:         $values \leftarrow null$
15:     end if
16: else if $JS$ and $branchDecl$ points to $getBranch$ then
17:     $branchArg \leftarrow findArg(getBranch, 1)$
18:     $values \leftarrow findValuesUsingPointsTo(branchArg)$
19: else
20:     $values \leftarrow null$
21: end if
```

branch in the call site (line #8). This value is then stored in the list of possible branch values (line #9). The list is passed by reference to this method initially. Depending on how branches are stored and used in a preference system, method *findBranchValue* could be implemented differently. In the next section, we discuss how we can instantiate this method for the preference system of one family of modern, highly-configurable software-system, the Mozilla family of applications.

4.2 Example Instances

In this subsection, we show how we can implement method *findBranchValue* for Mozilla-based applications. We then discuss an instantiation of the approach for analyzing markup languages.

4.2.1 Calculating Branch Values

For calculating branch values, we first determine if the branch is at the root. If so, the branch value is `null`. Otherwise, we have to find where branch was initiated and what are its possible values.

Algorithm 4 shows an instance of method *findBranchValue* (Line #8 of Algorithm 3). Note that we want to find the possible values of a branch that is an argument of a method (possibly a configuration API) at a particular call site. As we mentioned earlier, Mozilla-based applications are written in multiple programming languages, the two most used of which are C++ and JavaScript. The preference system uses *XPCOM* (Cross Platform Component Object Model) interfaces, in which each language provides its own APIs to access these interfaces (we discuss this in more detail in Section 5). In C++, branch values are passed around by method `GetBranch(const char *aPrefRoot, nsIPrefBranch **_retval)`, where the first argument is the branch value, and the second argument is the branch itself. However, in JavaScript, branch access is provided by method `getBranch(in string aPrefRoot)`, in which the argument is the branch value. Based on these two different APIs, we separated some parts of the strategy for C++ and JavaScript.

```
1 <preferences>
2  <preference id="xulschoolhello-message-count-pref"
3    name="extensions.xulschoolhello.message.count"
4  type="int" />
5  <!-- More preference elements. -->
6 </preferences>
7
8 <textbox
9  preference="xulschoolhello-message-count-pref"
10 type="number"/>
```

Figure 4: Example of how preferences are handled in XUL.

First, the algorithm constructs def-use chains (line #1), which consist of all the uses for each definition. On line #2, it finds the declaration of the branch. In the case of C++, we need to find the uses of the declaration and check whether the branch ever gets passed to the branch API. If so, the algorithm gets the corresponding argument and finds its value by using points-to information (line #5-11). Otherwise, it means that the branch has not been changed and is at the root. In case of JavaScript, we need to know whether the result of the *getBranch* method ever gets assigned to the branch. In other words, we need to know if the branch declaration points to the *getBranch* method. If this happens, the algorithm uses point-to information to find the possible values of the branch. If not, also in this case, it means that the branch is at the root.

4.2.2 Extracting Configurations from Markup Languages

In order to enable users to modify some of the preferences at the menu level, developers use markup languages, such as XML and XUL, to build the UIs.

Preference handling in XUL is facilitated by using *prefwindow*, which acts as a container. It lists the preferences that are going to be used later in the code. After being defined in the prefwindow, the preferences can be associated with form elements in the window, such as a text box. Figure 4 shows an example of preference handling in XUL. Lines #1-6 define the preferences, and lines #8-10 associate a preference to a text box. To extract the preferences from XUL files, our technique parses them and gathers the elements from the preference tags.

In XML documents, there is a section called CDATA, which is marked for the parser to interpret those elements as only character data, rather than markup. Preferences are handled as part of the CDATA section, in which the content is written in JavaScript. Figure 5 shows an example of preference handling in XML where, in the CDATA tag, a configuration API (*getIntPref*) is called. Our approach for extracting preferences from XML documents is to parse the corresponding file and retrieve the content from the CDATA tags. Then, we use the same strategy that we used to extract preferences from JavaScript.

5. IMPLEMENTATION

Our implementation of SCIC supports the family of Mozilla-based applications. We chose this set of software systems because they are implemented in multiple programming languages and use a preference system based on a hierarchical tree structure, neither of which has been studied before in the context of finding preference inconsistencies. The family of Mozilla-based applications is built using a set of core modules, which are used by all of the programs in the family. (Therefore, any problems within the core will affect all programs in the family.) Each individual application builds on this core and has its own modules to implement its specific unique

```
<field name="mMenuAccessKey"><![CDATA[
Components.classes
  ["@mozilla.org/preferences-service;1"]
  .getService(Components.interfaces.nsIPrefBranch)
  .getIntPref("ui.key.menuAccessKey");
]]></field>
```

Figure 5: Example of preference handling in XML.

functionality. These applications include the Firefox web browser, are actively developed and maintained, and are widely used. In 2014 alone, for instance, Firefox had eight major releases. Given the wide user base of these applications, it is clearly important to make sure that preferences are suitably maintained and consistent.

As we mentioned above, Mozilla applications are written in multiple programming languages that include C, C++, JavaScript, Java, Python, and some markup languages, such as *XML*, *XUL*, and *XHTML*. To expose its preference system, Mozilla uses *XPCOM* interfaces, where *XPCOM* is a cross platform component object model that has multiple language bindings. These language bindings act as a bridge between *XPCOM* and different languages by providing access to *XPCOM* objects in each language. In addition, languages for which there are *XPCOM* bindings can use modules that are written in other languages as *XPCOM* objects. Therefore, *XPCOM* components can be used and implemented in JavaScript, Java, and Python, in addition to C++. Interfaces in *XPCOM* are defined in *XPIDL*, which is an Interface Description Language [16].

Two commonly used interfaces in the Mozilla preference system are *nsIPrefService* and *nsIPrefBranch*. *nsIPrefService* is the preference system that is the main entry point to the management of preference files. It provides access to the preference branch object through methods such as *getBranch(prefroot)*, which allows the direct manipulation of preferences [12]. *nsIPrefBranch* can be obtained either directly or from the *nsIPrefService*. *nsIPrefBranch* is created with a *root* value, which describes the base point in the preference tree from which this *branch* stems. Preferences can be accessed starting from this root by simply using the final portion of the preference. If *nsIPrefBranch* is created with the root *browser.startup.*, for instance, preferences *browser.startup.page*, *browser.startup.homepage*, and *browser.startup.homepage_override* can be accessed by simply passing *page*, *homepage*, or *homepage_override* to configuration APIs. To enable manipulation of preference data, *nsIPrefBranch* provides two set of methods—one set for reading the preference value, such as *getTypePref* or *getComplexValue*, and another set for writing its value, such as *setTypePref* or *setComplexValue*. In both cases, *Type* should be replaced by one of the three types `Int`, `Char`, or `Bool` [11]. Each language implements its own interfaces to interact with these configuration APIs.

To implement the analysis for the C++ side of applications, we used the LLVM compiler infrastructure [8] and Clang [6]. We used them to generate the call graph and points-to information, as well as the data flow information in Algorithm 4. As mentioned earlier, C++ implements its own interface to interact with XPCOM objects in the preference system. Instead of *getTypePref* or *getComplexValue* configuration APIs, it uses *GetTypePref* or *GetComplexValue* with different arguments.

To implement the analysis for the JavaScript side of the applications, we used the WALA framework [15] to generate the call graph and points-to information. However, the recent versions of Mozilla follows the ECMAScript [7] 6 standard, which is not supported by most JavaScript analysis frameworks. Therefore, we used Traceur [14], a compiler that takes ECMAScript 6 and compiles it

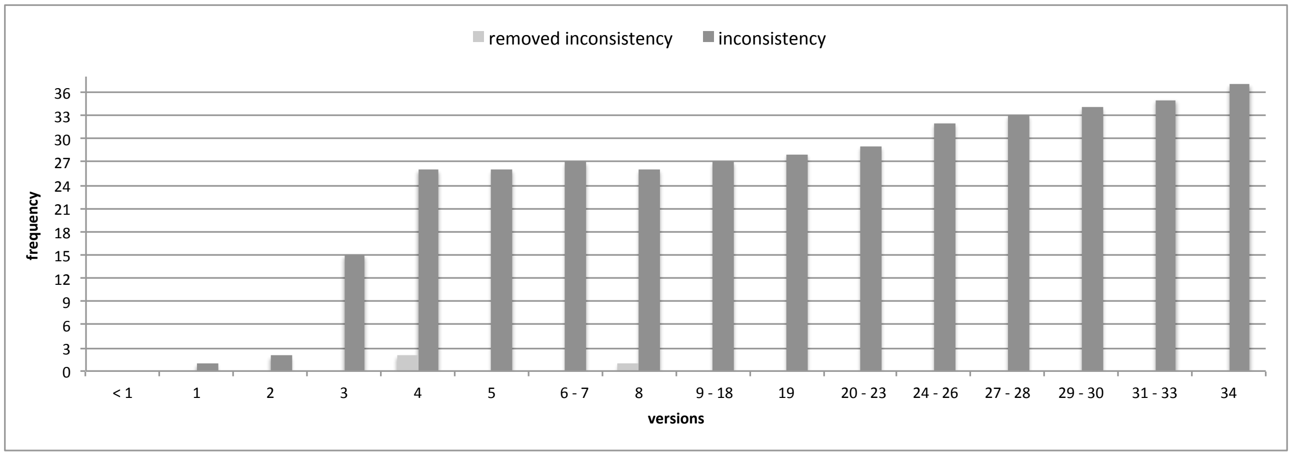

Figure 6: Number of source-UI inconsistencies appearing for the first time in each version.

down to regular ECMAScript 5 JavaScript, to generate source code that the current analysis framework is able to parse. Using this approach, we were able to analyze most of the files in the codebase we considered in our experiments. For about 15% of the code, however, we had to perform part of the analysis manually.

To analyze the XUL files, we used an XUL parser written in Python and provided by the Mozilla developers [17]. It parses XUL and builds a list of all the XUL elements and their attributes, from which we extracted preference elements. For XML, we used SAX (Simple API for XML [13]), a java-based parser. After parsing, we analyzed the content from the CDATA section (written in JavaScript).

6. EMPIRICAL EVALUATION

To evaluate how effective and useful is SCIC in detecting preference inconsistencies, we performed an empirical evaluation and investigated three research questions. The first question assesses whether SCIC can correctly identify inconsistencies that exist in an application. The second question focuses on the second part of our analysis, which is to understand how prevalent inconsistencies are, and how long they persist in the history of a system. The last question focuses on the categories of inconsistencies found by SCIC to understand if they are actually problematic and relevant for the users of the applications being analyzed. More specifically, we investigated the following three questions:

1. ***RQ1:*** *How accurate is SCIC in identifying inconsistencies?*
2. ***RQ2:*** *What is the lifespan of inconsistent configuration options?*
3. ***RQ3:*** *To what classes do the inconsistent preferences belong? Are these inconsistencies potentially harmful?*

6.1 Evaluation Method

We selected a 10-year period (from February 2004 to December 2014) of the Mozilla core modules, which are shared between most of the Mozilla applications, such as Firefox, Bugzilla, Thunderbird, and SeaMonkey. We also examined all Firefox versions (12 million lines of code on average) from the same time period, as this is one of the largest applications in the Mozilla family. This corresponds to 35 major releases of Firefox, from version 0.8 to version 34.0. Since the core modules are also used within Firefox, and the release schedules for these two may be different, we used the Firefox versions as our baseline for evolution and studied the versions of the core modules that corresponded to those versions of Firefox. This allowed us to examine and differentiate inconsistencies across the two sets of modules using a single timeline. We compared our analysis result against the configuration options that are exposed through `about:config`. If there was any preference in `about:config` in a particular version that did not exist in the source code of the same version, we marked that as an inconsistency. For each version of each application, we ran SCIC and identified both the new inconsistencies and those which persisted from earlier versions. For each of the inconsistencies found, we confirmed that the source code was missing through manual examination.

Table 1: Number of inconsistencies.

Modules	# source-UI inconsistency	Programming Languages		
		CPP	JS	Markup
Core	17	17	0	0
Firefox	23	11	8	4
Total	40	28	8	4
Percent	-	70%	20%	10%

6.2 RQ1: Finding Inconsistencies Accurately

Across the 35 major releases that we studied, we found 40 inconsistent preferences. Table 1 provides some statistics about these preferences. The first column lists the modules that we studied and shows the totals for both applications. The next column (#source-UI inconsistency) shows the number of inconsistencies found in each module between the source code and the `about:config` list. The next group of columns shows the distribution across different programming languages.

As Table 1 shows, of the 40 preferences, 17 belong to the core modules and 23 to the Firefox modules. Although Firefox also uses the Mozilla core modules, the 23 inconsistencies reported for Firefox are in addition to those found in the shared modules (which affect any Mozilla-based application, including Firefox, SeaMonkey, and Thunderbird). The table also shows that 70% of the inconsistencies exist in C++ code, most of which are in the core modules. The remaining 30% is shared between JavaScript (20%) and markup languages, such as XML and XUL (10%).

Our manual check for false positives found none, so to the best of our knowledge, SCIC correctly identified the inconsistent preferences. Although we do not have a complete list of preference inconsistencies in the analyzed code, and therefore we cannot as-

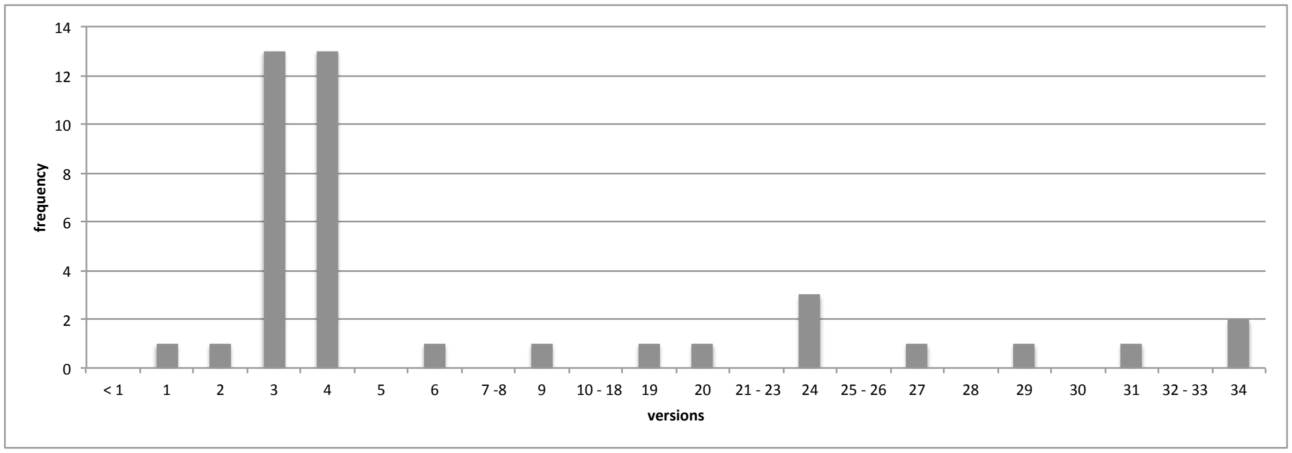

Figure 7: Number of source-UI inconsistencies in each version.

sess the recall of SCIC, we can nevertheless provide some initial, promising evidence: all the inconsistencies reported by users (*i.e.,* all the inconsistencies described in bug reports for the Mozilla core modules and Firefox) were detected by SCIC. We can therefore conclude that, at least for the known inconsistencies in the analyzed code, SCIC produced no false negatives.

Summary of RQ1. SCIC was able to accurately identify 40 inconsistencies distributed between the core modules and Firefox. The inconsistencies involve multiple programming languages, which indicates that single-language analyses would not be able to detect them.

6.3 RQ2: Life Span of Inconsistent Configurations

Since we studied the evolution of Mozilla-based applications for 35 major releases during 10 years, we also wanted to identify when the inconsistencies happened, how long they existed, and when they got removed (if ever). This data provides information on the potential impact and difficulty of detection of these inconsistencies. We therefore studied the lifespan of each inconsistency detected by SCIC.

Figure 6 shows how many inconsistencies appeared for the first time in each version. Versions without inconsistencies are not shown. Of the 35 versions that we studied, 22 did not have any new inconsistencies. 9 inconsistencies appeared for the first time in 9 versions, where they were individual instances. In one version (Version 24), 3 inconsistencies newly appeared. Interestingly, the largest portion of inconsistencies (65%) happened in two versions (Versions 3 and 4), each newly introducing about half of these inconsistencies.

The inconsistency that appeared in Version 27 for the first time has never been defined in the source code of major releases, neither in version 27 nor in the previous or later versions. (We only analyzed major releases, so it might exist in some Beta versions.) This was an interesting case, so we searched for any history related to this preference. We found a bug report from 2013 [5] where there was a discussion about introducing this preference. In the comments, the name *toolkit.asyncshutdown.timeout.crash* was initially chosen for the preference. However, someone later suggested to use *toolkit.asyncshutdown.crash_timeout* instead. Developers therefore changed the name in the source code, whereas the name in the intermediate UI was never changed. In fact, it still exists in the latest version we studied.

Figure 7 shows how many inconsistencies exist in each version, and how many of them were removed from each version. There are only two versions where the inconsistencies were removed. Two of the removals occur in Version 4, and one in Version 8 (which was our motivating example). The number of remaining inconsistencies in version 34 (the last version we studied) is 37, instead of 40, because of the removed inconsistencies.

Summary of RQ2. Our evaluation shows that the lifespan of inconsistencies can be long. Very few (only 3) of the inconsistencies that we found were removed.

6.4 RQ3: Type and Relevance of Inconsistent Preferences

We identified several classes of inconsistent preferences. Table 2 lists these classes, their description, and the number of (inconsistent) preferences that belong to each of the classes. In total, there are 10 classes and an additional class, *Others*, which contains those preferences that do not belong to any of the other classes.

Class *Performance* is the largest one, with six preferences. These are preferences that help users do tasks more efficiently. For example, preference *layout.frame_rate.precise* provides smoother scrolling for the users, and preference *application.use_ns_plugin_finder* enables the automatic plugin finder if a plugin is not found.

Class *Profile* is the next largest one, with five preferences. Browsers store user personal data, such as history, bookmarks, and extensions, in a profile, and a user can have multiple profiles. *Profile* related preferences allows customization of such profiles. For example, preference *profile.confirm_automigration* enables migration of other browser bookmarks when creating a new profile, and preference *profile.seconds_until_defunct* allows for specifying after how long an unused profile should be removed.

Classes *Accessibility*, *User interface*, and *Security* have four preferences each. *Accessibility* preferences provide services for people with disabilities. Preferences *accessibility.usebrailledisplay* and *accessibility.usetexttospeech*, for instance, are designed to accommodate alternate devices with enhanced accessibility features. *User interface* preferences enable customization of the user interface. Preference *browser.chrome.toolbar_style*, for instance, determines how the navigation toolbar buttons (*e.g.,* back, forward, reload) are displayed, either as text or icon. *Security* preferences are related to security features, and we discuss them in more detail later in this section.

Classes *DOM* and *Printer* contain three inconsistent preferences each. The former relates to the manipulation of the Document Object Model (*e.g., dom.workers.maxPerDomain*, which specifies the maximum number of workers), whereas the latter is used to customize printing settings (*e.g., print.print_extra_margin*, which al-

Table 2: Classification of inconsistent preferences.

Class name	Description	frequency
Accessibility	provide services for people with disabilities	4
DOM	related to Document Object Model (DOM) which is a programming interface	3
Encoding	allows different character encoding	2
History	related to browser history	2
Network	related to networking	1
Performance	enables users to perform tasks faster	6
Printer	enables users to configure printer	3
Profile	allows customization of browser profiles	5
Security	provides secure way of doing tasks	4
User Interface	related to users interaction with interface	4
Others	anything that does not fit into any of the above classes	6

lows for specifying an extra gap or margin around the content of the page). Classes *History* and *Encoding* contain two inconsistent preferences each. *History* related preferences are used to customize the browser history settings, whereas class *Encoding* refers to character encoding.

Finally, there is only one inconsistent preference of class *Network*, whereas we put all remaining inconsistent preferences in class *Others*. Some examples from this class are *browser.EULA.version*, which specifies the version of End User License Agreement that has been viewed and accepted by the user, and *capability.policy.default.SOAPCall.invokeVerifySourceHeader*, which permits users to make verified SOAP (Simple Object Access Protocol) calls by default.

Among the above classes, the security related preferences seem to be particularly important, as they might affect the overall security of the system. We therefore discuss the four inconsistent preferences in this class in greater detail. *Security.checkloaduri* forbids external sites to link to files that are stored locally, and *security.xpconnect.plugin.unrestricted* allows plugins to access XPCOM methods that would normally be unaccessible. When modifying these preferences in the UI, users would assume that they operate as expected, which they do not, and would therefore use the browser under wrong security assumptions. Preferences *security.ask_for_password* and *security.password_lifetime* define whether the browser should ask to remember passwords when logging into a web page and for how long. In this case, we found both a bug report [4] and a complaint [10] in which users were considerably annoyed by the fact that these preferences did not work as expected.

Other inconsistencies, although not related to security, may create issues too. Inconsistencies in the printer category may result in annoyances for the user, for instance, and those in the history category may impact privacy. To provide an example, consider preference *browser.history.grouping*, which defines how to group history URLs, which SCIC identified as inconsistent, and for which we found a bug report complaining about it [2].

Summary of RQ3. We found 10 distinct classes of inconsistencies in the applications studied. Although we do not have strong evidence that all classes are harmful, we found several examples of issues that are clearly problematic. In particular, security related preferences, when inconsistent, are likely to provide a false sense of security to users, who may use the browser in a way that makes them more vulnerable to attacks.

6.5 Lessons Learned

We now summarize the key lessons learned for this evaluation.

Inconsistencies do exist, and we should test for them. When we started this work, we were not sure of whether we could find real inconsistencies and whether they would be relevant. Based on our initial results, we believe that inconsistent preferences exist and are potentially problematic for the users.

Multi-language analysis is needed. We found inconsistencies that involved different parts of the system written in different languages. This means that an inconsistency analysis could not be performed without using a multi-language approach.

Doing individual analyses and merging their results seems to be a promising approach. An alternative approach to SCIC is to build a single analysis that works across multiple languages at once. The approach we have used, to merge individual analysis results is considerably simpler and seems to be effective in this context.

As a user, do not trust preferences. It appears that the layered approach for setting and using preferences can lead to inconsistencies, so users should not trust preference settings without making sure that they behave as expected (at least for the critical ones).

6.6 Reporting Our Findings to Developers

We submitted a bug report about the inconsistencies that we found to the Mozilla developers, who told us that some features were indeed left in the UI due to negligence, whereas others were left because in use in other applications of the Mozilla family (and having only one version of the UI was convenient). However, they recognized that some users found this to be annoying, and that such issues should be further investigated and fixed [1].

6.7 Threats to Validity

The primary threat to external validity of this work concerns whether or not our results will generalize. To mitigate this threat, we used the core modules from a large family of applications, as well as one of the most widely used applications in this family. Although we need to perform more studies on different systems before being able to claim that our approach works in general, considering the body of related work (*e.g.,* [23, 32]), we believe that our results can generalize to other families of application that have a tree-structured preference system and are multi-lingual (*e.g.,* the OpenOffice applications, industrial systems, and other applications that rely on the Windows registry). In future work, we plan to instantiate our approach for these systems, as we did for the applications that we targeted in this paper.

To address possible threats to internal validity, we manually evaluated the inconsistencies found by SCIC and did not find any problem with the results. Although this is not a guarantee of correctness, it excludes the presence of obvious issues with our implementation of SCIC.

Finally, with respect to construct validity, we might have used different metrics to answer our research questions, which is true for every empirical study. However, we believe that the ones we chose are appropriate for the questions that we were investigating.

7. RELATED WORK

There has been a large body of research on testing of highly configurable systems [21, 25, 29, 30, 33, 41], as well as on extracting feature models, configurations, and constraints from both source code [26, 27, 36] and intermediate representations (*e.g.,* KConfig for Linux [20,35]). This line of research uses only a single view of the configuration space and does not measure inconsistencies between code and other representations. In prior work, we used a model of the Firefox configuration space to perform failure avoidance [37]. However, we used only the `about:config` information to build the model. Interestingly, upon examining that model retrospectively, we have found multiple instances of preferences that SCIC identified as inconsistent. In fact, we discovered that in that work we might have been manipulating preferences that had no associated code and no behavioral impact. We focus the rest of the discussion on the work that is most closely related to SCIC; that is, work on identifying mismatches between source code and other components of the software and on diagnosis and debugging of configuration errors.

7.1 Identifying Mismatches

Most of the work on identifying mismatches between source code and other components of software focuses on documentation. Rubio-González and Liblit [34] examined mismatches between documented and actual error codes returned by system calls. Tan and colleagues [38] presented a technique that detects inconsistencies between comments and source code to identify both bugs and low-quality comments. The closest work to SCIC is that of Rabkin and Katz [32]. Their approach statically extracts the list of configuration options from source code and compares those options with those listed in the documentation. Although our analysis is inspired by their work, we are looking into a different type of preference systems, that is, those that have a hierarchical tree structure, which requires a more complex analysis. In addition, we focus on the user interface preference system, rather than on the documentation. Furthermore, the systems we are studying are written in multiple programming languages and are highly configurable. Their work, conversely, was limited to Java programs with a relatively low total number of configurations—less than 700 across seven programs, in their evaluation. (For comparison, each version of the Mozilla-based applications we studied have more than 2,000 configurations.) Finally, we studied the application along a long history or versions (10 years), rather than focusing on individual versions, which adds a longitudinal dimension to the analysis.

7.2 Error Diagnosis and Debugging

There has also been a good deal of research on configuration error diagnosis and debugging [18, 19, 23, 24, 31, 39, 40, 42–44], much of which has been empirical in nature. Jin and colleagues studied several highly configurable software systems to identify the challenges that configurability adds to testing and debugging [23]. In follow-up work, they built a system for helping identify preferences at the user interface level [22]. Yin and colleagues [42] conducted an empirical study on configuration errors in commercial and open source systems. Rabkin and Katz [31] defined a technique that precompute program points where there is a possibility that some configurations cause an error at those points to aid debugging. Zhang and Ernst [43] proposed a technique for identifying the root cause of crashing and non-crashing configuration errors. They also proposed a technique for debugging configuration errors that are caused by software evolution [44]. However these approaches require some form of user traces and are limited to a single programming language. Attariyan and Flinn [18, 19] proposed several techniques to diagnose and troubleshoot configuration faults, while Keller and colleagues [24] presented a tool for assessing resilience of software systems to human configuration errors. Finally, Wang and colleagues [39] presented a technique for diagnosing the root cause of misconfiguration.

SCIC differs from these techniques in its primary goal, which is to find inconsistencies between source code and high-level user interface preferences (with particular emphasis on those introduced during evolution). SCIC does not need to identify changed behavior ahead of time and can handle systems written in multiple programming languages and using complex, hierarchical preference structures. The inconsistencies that SCIC finds may be eventually classified as configuration errors that affect the usability of the software systems, but they may also point to preferences that simply need to be removed from the interface.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented SCIC, our static analysis based Software Configuration Inconsistency Checker. SCIC can identify preference inconsistencies on highly-configurable, complex applications. SCIC can handle applications that are written in multiple programming languages and that use complex (*e.g.,* tree-structured) configuration systems, which are both common characteristics in modern software (*e.g.,* Mozilla applications, open source office suites, and applications that use the Windows Registry).

We have empirically evaluated SCIC on 10 years of releases of the core modules from the Mozilla family, as well as on the 35 releases of the Firefox application that appeared during the same timeframe. SCIC was able to find 40 inconsistencies across these versions and both within the core modules and in application specific code. The inconsistencies found were not isolated to a single language, but rather involved code written in C++, JavaScript, and various markup languages (*e.g.,* XUL and XML). In addition, their lifetime spanned multiple versions.

We were able to classify the inconsistencies found by SCIC into ten categories, where each category contains from one to six preference inconsistencies, and whose potential severity varies from mild annoyances to more serious security or performance problems. We also found postings in various forums with user complaints and bug reports that are related to some of the inconsistencies that SCIC found. Finally, the Mozilla developers confirmed some of the issues we reported to them, which are currently being investigated. These reports, complaints, and confirmations provide clear evidence that at least some of these inconsistencies are perceived as problematic and harmful by the users and are thus worth discovering.

In future work, we plan to extend SCIC to additional applications, including the LibreOffice application family. We also intend to package and release SCIC as a tool that can be readily used and provides a way to plug-in new individual analyses into the framework (*e.g.,* to add analysis for new languages). Finally, we will investigate techniques for either automatically repairing the inconsistencies found by our approach or at least modify the codebase so that users are warned when trying to use an inconsistent preference.

9. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1161767 and CCF-1161821, and CCF-1320783, and by funding from Google, IBM Research, and Microsoft Research to Georgia Tech.

10. REFERENCES

[1] Bug 1176600 . https://bugzilla.mozilla.org/show_bug.cgi?id=1176600.

[2] Bug 518976 . https://bugzilla.mozilla.org/show_bug.cgi?id=518976.

[3] Bug 701139. https://bugzilla.mozilla.org/show_bug.cgi?id=701139.

[4] Bug 719705 . https://bugzilla.mozilla.org/show_bug.cgi?id=719705.

[5] Bug 917764 . https://bugzilla.mozilla.org/show_bug.cgi?id=917764.

[6] Clang. http://clang.llvm.org.

[7] ECMAScript. http://www.ecmascript.org.

[8] LLVM. http://llvm.org/.

[9] Mozilla. https://www.mozilla.org/en-US/.

[10] MozillaZine. http://forums.mozillazine.org/viewtopic.php?f=38&t=408689.

[11] nsIPrefBranch. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsIPrefBranch.

[12] nsIPrefService. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsIPrefService.

[13] SAX. http://sax.sourceforge.net.

[14] Traceur. https://github.com/google/traceur-compiler.

[15] Wala. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[16] XPCOM. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM.

[17] XUL Parser. https://developer.mozilla.org/en-US/docs/XUL_Parser_in_Python.

[18] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proceedings of the USENIX Conference on Annual Technical Conference (ATC)*, 2008.

[19] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[20] N. Dintzner, A. van Deursen, and M. Pinzger. Analysing the linux kernel feature model changes using fmdiff. *Software & Systems Modeling*, 2015.

[21] B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2011.

[22] D. Jin, M. B. Cohen, X. Qu, and B. Robinson. PrefFinder: getting the right preference in configurable software systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2014.

[23] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the International Conference on Software Engineering (ICSE Companion)*, 2014.

[24] L. Keller, P. Upadhyaya, and G. Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.

[25] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2013.

[26] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2014.

[27] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.

[28] OpenOffice.org 3.1 developers guide, 2009. https://wiki.openoffice.org/w/images/d/d9/DevelopersGuide_OOo3.1.0.pdf.

[29] X. Qu, M. Acharya, and B. Robinson. Configuration selection using code change impact analysis for regression testing. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2012.

[30] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the International Symposium On Software Testing and Analysis (ISSTA)*, 2008.

[31] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2011.

[32] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[33] B. Robinson and L. White. Testing of user-configurable software systems using firewalls. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[34] C. Rubio-González and B. Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

[35] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[36] C. Song, A. Porter, and J. S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[37] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2014.

[38] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[39] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[40] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In

Proceedings of the Conference on Symposium on Operating Systems Design and Implementation (OSDI), 2004.

[41] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering (TSE)*, 2006.

[42] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[43] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[44] S. Zhang and M. D. Ernst. Which configuration option should i change? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.