Static Extraction of Program Configuration Options

Ariel Rabkin and Randy Katz
EECS Department, University of California, Berkeley
{asrabkin,randy}@cs.berkeley.edu

ABSTRACT

Many programs use a key-value model for configuration options. We examined how this model is used in seven open source Java projects totaling over a million lines of code. We present a static analysis that extracts a list of configuration options for a program. Our analysis finds 95% of the options read by the programs in our sample, making it more complete than existing documentation.

Most configuration options we saw fall into a small number of types. A dozen types cover 90% of options. We present a second analysis that exploits this fact, inferring a type for most options. Together, these analyses enable more visibility into program configuration, helping reduce the burden of configuration documentation and configuration debugging.

Categories and Subject Descriptors

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

General Terms

Management, Measurement

Keywords

Configuration, documentation, experiences, static analysis

1. INTRODUCTION

Modern systems software often exposes a wide range of configuration options to users. By setting configuration options, users can control many aspects of execution. Configuration determines such aspects of execution as where in the local filesystem to store data, which ports a server should bind to, and even which algorithms should be used in various parts of a large system.

Program configuration is often stored as a map from option name to value. Some configuration mechanisms, such as the Unix system environment, use arbitrary strings as option names. Other mechanisms, such as the Windows Registry, store options hierarchically in a tree structure. Still other systems use XML formats for configuration data, where the path to the value represents its name. This key-value style of configuration is convenient for programmers, because it makes it easy to add new options incrementally. There is no schema or centralized list of supported options.

Several kinds of errors can arise from this style of configuration. User-written configuration files can assign values for options that a program never reads, either because of a typing error or because program evolution resulted in an option being removed or renamed. Documentation is often updated separately from code, and can fall out of step with an evolving program [28, 29]. As we show, documentation sometimes claims that an option has a particular effect, when in fact that option is never used at all. Conversely, a newly added configuration option does users little good if it is not documented. The open source systems software we study all have significant undocumented configurability, suggesting that this is a widespread problem.

Programmers often distrust documentation, preferring to read source code to understand program behavior [18, 29]. While this attitude may be appropriate for programmers, it works less well for users and administrators. Bad documentation is a significant bar to adoption of open-source software, as well as a considerable headache for users [19]. Excess configurability and poor documentation have been recognized as a problem for several years [22] but solutions have been slow to emerge.

This paper argues that static analysis can compensate for the weaknesses of this key-value style of configuration management. Static analysis can extract a schema for configuration, tying both user configuration files and program documentation back to the actual structure of the associated program. This can be done efficiently for substantial existing systems that would be expensive or difficult to rewrite.

We make three contributions. First, we document the way this key-value configuration pattern is used in a range of open source projects. We analyzed seven open-source programs, totaling well over a million lines of code and representing the work of many dozens of developers. We observed that configuration options tend to be used in standardized ways. There are modest number of use patterns that together account for more than 90% of options. We found pervasive documentation errors. Every program in our sample had documentation for options that do not actually exist as well as significant numbers of undocumented options.

These pervasive errors motivate our second contribution.

ICSE May 21-28 2011, Waikiki, Honolulu , HI, USA

We describe and evaluate a static analysis that outputs a list of configuration options potentially used by a program and the program points where each option is read. This analysis finds more than 95% of the options in the programs we inspected. This accuracy rate is higher than in the documentation of mot of the associated programs, making the analysis practical for finding documentation errors.

The analysis can also help catch mistakes made by users. In operational experience at Yahoo!, typographic errors in option names are a major cause of problems with Hadoop and related programs[1]. A Hadoop user can easily set the value of a non-existent option like `default.fs.name` when they meant to refer to a similarly-named real option, `fs.default.name`, instead. There is no central list of valid options that can be used as a dictionary for a "spell check for configuration"; references to configuration options are scattered throughout the code. As a result, this sort of error produces few overt symptoms and can be frustrating to track down. The developers could potentially maintain a list of valid options for use in configuration checking, but the many flaws we found in program documentation suggest that task is difficult and error-prone. Automated analysis, by reducing the burden, can help.

Last, we present and evaluate an additional analysis that can infer the domain of valid values for an option. Our approach works by recognizing the specific patterns by which developers use configuration. This approach finds types for most options in our sample, and has few false positives. This analysis builds on the previous one, consuming the set of options read by the program. It also builds on (and validates) the results of our empirical study: recognizing patterns in code makes sense because a small set of patterns covers a large fraction of options.

There are several applications for this analysis. It can be used to produce a first draft of program documentation for humans. It can also produce machine-readable specification for the permissible values of options. There has been work by the systems community on automatic performance tuning, for instance [8], and by the software engineering community on configuration-aware resolution of reflection in method calls [27]. Being able to automatically find tunable numeric parameters or class-name parameters would be helpful in these contexts.

Finding constraints on option values, even on a subset of options, would help catch additional user configuration errors. Our analysis can automatically determine not only that `hadoop.util.hash.type` is an option, but that the only possible values for it are "murmer" and "jenkins". Programs do not consistently report configuration errors [13]; often, they silently substitute a default value. Extracting constraints on option values enables static checking for these sorts of mistakes, potentially saving hours of user time if a bad configuration value causes a long batch job to go awry. This extends our "spell check for configuration" to validate values, not just names.

We opt for static, rather than dynamic analysis. Many options are used only in particular program modules or as a result of particular inputs. Hence, dynamic testing has poor prospects for finding all uses of configuration options. Static analysis can attain high coverage much more easily. Our analysis uses standard points-to and call graph construction algorithms. Our code bases of interest make heavy use of reflection and remote procedure calls. This requires some tailoring of the analysis implementation. Those aspects of our analysis are discussed in Section 4.2.

We begin in the next section with our empirical measurements. In Section 3 we show how static analysis can find the set of options used by a program. In Section 4, we describe and evaluate an analysis framework for determining the types of configuration options. Section 5 discusses how general the problem and solution presented in this paper are. Section 6 describes related work. We summarize our observations and conclusions in Section 7.

[1]Owen O'Malley, Yahoo!, Personal communication, January 2010.

2. QUANTIFYING PROGRAM CONFIGURATION OPTIONS

To better understand program configurability, we looked at the configuration mechanisms in a range of existing software projects. We looked for large highly configurable open-source software packages written in Java. We restricted ourselves to Java because our static analysis implementation targets Java and we use the results of the survey discussed here as ground truth in evaluating the analysis. Each of these projects includes several different programs, sharing much of their code.

The projects we analyze span a range of applications and have a wide variety of developers. Several were developed for industrial use by commercial developers. Hadoop is a distributed filesystem and MapReduce implementation [9] largely developed at Yahoo! and Facebook. HBase [2] is a re-implementation of Google's BigTable storage architecture developed by a loose collection of open-source developers spread across several companies. Derby is an open-source database originally developed as part of IBM's Cloudscape project [1]. Cassandra is a distributed storage service developed at Facebook [17]. Other projects were developed in an academic context. FreePastry is a peer-to-peer distributed hash table intended for the wide area, originally developed at Rice [25]. JChord is a program analysis engine developed at Stanford [23]. Nachos is a model operating system environment used for undergraduate education at Berkeley [11].

The code we analyze is the work of many programmers. We do not have precise counts for the number of contributors to each project but in aggregate the number is surely over 100. Derby and Hadoop together list 50 committers. Typically there are several times as many occasional contributors who are not individually credited. We therefore believe the programs in our study represent a range of programming styles and are unlikely to be unduly influenced by the idiosyncrasies of a few individuals.

For each program, we consulted the documentation and default configuration files to derive a list of options. We then manually classified each option.

2.1 A taxonomy of configuration options

Rather than impose a taxonomy *ex ante*, we built ours bottom-up, describing each option as we encountered it and then looking for patterns. We show statistics for each of the individual programs in our study in Table 1. We also present background statistics about each program. As can be seen, there is substantial variance across these programs: the absolute number of identifiers and the proportion of identifiers in each column varies widely. FreePastry, for instance, has

Application	LoC	KB compiled	Numeric	Identifier	Mode	Other	All options
Cassandra	36,823	1,961	17	14	3	2	36
Derby	1,136,718	7,903	16	17	23	13	69
FreePastry	175,085	6,073	154	8	39	10	211
Hadoop	167,653	5,440	89	70	31	16	206
HBase	104,781	3,149	38	20	3	3	64
JChord	35,761	1,209	5	26	14	12	57
Nachos	10,896	467	2	4	10	0	16
Total			321	159	123	56	659

Table 1: Numbers of options by application, with breakdown by type of option. KB compiled = size of compiled binary, in kilobytes.

Type	Category	Total
Time Interval	Numeric	118
Count	Numeric	115
Boolean	Mode	104
File	Identifier	60
Size	Numeric	53
Class Name	Identifier	38
Address	Identifier	28
Fraction	Numeric	28
Mode name	Mode	18
String	Other	17
Port number	Identifier	13
Internal ID	Other	9
Password	Other	8
URI	Identifier	7
User ID	Other	7
Network Interface	Identifier	5
Other	-	31
Total		659

Table 2: The most common configuration option types, with classification.

a very large number of controllable timers that boosts both the total and numeric columns. Table 2 shows our list of option types and how many instances we found of each type of option, summing across all the programs in our study.

Several categories need explanation: a *count* is an integer parameter describing how many of some entity should exist, such as threads in a pool, iterations of a loop, and so forth. A *size* is a quantity of memory or storage, measured in bytes. A *mode name* is a string, drawn from a small set, that selects how a program should behave from a small set of options. An *internal ID* identifies some program-defined entity, such as distributed files in the case of Hadoop. "String" and "Number" are catch-all types for string or numeric options that are not used in some other well-defined way by the program. For instance, JChord can run a target program using dynamic instrumentation; the labels used to indicate each test run are uninterpreted strings.

As can be seen, most options fell into a handful of types. The top three categories together account for slightly over half of all options seen. Numeric options are very common and are primarily used for a small number of purposes: to control timers, resource pool sizes, and memory allocation. Non-numeric options are overwhelmingly external identifiers, often designating files, network addresses, or Java classes.

Some options include complex structured data. Examples include regular expressions, date format strings, or command line arguments for a subprocess. These complex options are rare in the programs we have inspected: we found a total of four options controlling process arguments and three options with nontrivial internal semantics: one regular expression and two `strftime`-style date format strings.

The list of option types can be further summarized. Most options fall into one of three broad categories. The largest category of options is tunable *numeric parameters*, controlling buffer sizes, time-out periods, and so forth. (This includes both integer and floating point options.) Another large group of options select a *mode of operation* from a small set. This includes Boolean options, as well as mode names. The last group of options consists of *external identifiers*: strings or numbers that refer to some entity outside the program, such as file names or network addresses. Java class names are effectively also external identifiers, since the runtime maps them to the names of files in the Java class path. A handful of option types remain outside this tripartite classification. Internal identifiers, opaque strings or numbers, and a few miscellaneous types such as passwords do not fit neatly into any of the three categories mentioned above. These are tabulated as "other" above.

While our ad-hoc approach worked reasonably well, there were a few difficulties. Some options can take a list of values. We treated "Type" and "list of Type" as equivalent. When a string is used as a suffix to a file path, we count it as a string, not a file name. Several Hadoop options are path names to files in a distributed filesystem. We consider these to be internal identifiers, not file names.

2.2 Configuration APIs

We also looked at the structure of the code each program used for reading and processing options. Of the seven programs in our sample, six had a narrow and well-defined API for configuration. In each, there was exactly one class that exposed a key-value interface to the rest of the program, through which configuration options could be read. These classes offer a set of methods for retrieving configuration values of particular types: `getBoolean`, `getInt`, and so forth. Each method takes the name of an option as parameter and optionally a default value to be used if the option is not set. The Java System Properties API, part of the Java Platform standard, offers this interface as well. The Unix system envi-

ronment is also a key-value map. We therefore conclude that this style of interface represents a popular and widespread programming abstraction.

Derby was the one partial exception we saw to this pattern. In Derby, there are three levels of configuration, consulted in turn: global options specified in a file, per-database options, and options specified programmatically. Each option can be set in some subset of those tiers. A substantially more complex API is needed to manage configuration. There are configuration retrieval methods in several different classes, differing in which locations are searched for configuration. Each of these configuration retrieval methods, however, followed the standard pattern, taking a string argument for the option name plus an optional default value.

The concrete syntax for configuration varied significantly. Hadoop and HBase use an XML-based format containing key-value pairs. Cassandra uses a more complex XML structure, with nested elements; the program reads elements from this file using XPath queries in an essentially key-value style. The remaining programs use a flat ASCII file with a list of `name=value` pairs.

Some of the programs in our study also accept command-line arguments. In many cases, these arguments duplicate the functionality of configuration file options or are accessed via a similar key-value interface. Hadoop largely eschews command line options; most of Hadoop's component programs only accept options that set configuration values.

3. FINDING OPTIONS

In this section, we answer two research questions about configuration options: how good is existing configuration documentation and how well does static analysis do compared to this human standard. This is motivated by our desire to have automated tools check documentation, or even produce the authoritative version.

Today, developers looking to extract a list of configuration options from source code would have to resort to searching through the text for calls to configuration read methods. This approach cannot readily find all option names. In the code we examined, we saw many examples of application methods that take an option name as parameter. As a result, there can be several function calls between the string literal for an option's name and the point where that name is passed to a system or library-defined configuration method. All of the programs in our study define several utility methods that take an option name and return a new object corresponding to that name. For example, in Hadoop there is a method that take an option name as parameter, read the value of that option, and reflectively creates and object of the class named by that value. Hence, finding which strings are used as option names requires inter-procedural analysis to track these string constants through method calls.

Simple lexical approaches would yield less precise information than our analysis. All of the software packages we examined consist of multiple executables sharing large portions of code. It can be helpful to know which component programs will use which options, or even whether an option is only read in dead code. Our technique derives this information readily, but lexical techniques cannot.

3.1 Approach

Our approach is outlined in Figure 1. We break the overall problem into two major pieces: First, finding the program points where options are read or written; second, finding the possible option names at each of these points. These two stages are somewhat independent; different algorithms can be used in each without disrupting the overall structure of our approach. We output both the set of program points that read options and a regular expression for the options read at each of these points. This means that our analysis is independent of the syntactic details of configuration file format. Instead, we rely on the APIs, which tend to be more consistent across programs.

```
Construct points-to and call graphs.
Mark known configuration methods.
Mark option-name arguments to these methods.
while (not converged to fixpoint)
  for each method m:
    if an argument of m used as an option name
      Mark method as conf. read call.
      Mark argument as option name.
Find possible string params at call sites
Output option names and read points.
Output methods taking option names as arguments.
```

Figure 1: Pseudocode for analysis to find options.

The usual API for configuration consists of a set of related calls, each of which has a string-typed argument corresponding to the option name. We assume that we have either annotations on these API methods, or, equivalently, a list of methods returning or setting configuration. For this study, the annotations required were compiled into the code of the analyzer. For each program, these consisted of a few lines of the form "include all methods in class `Properties` whose name starts with get." These annotations also specify which parameter is the name of the option. (This is generally the first parameter of string type, in our experience.)

We expand this set of configuration-reading methods by finding all methods taking a string argument, where that argument is passed as an option name to an already-discovered configuration method. This accounts for the common programming pattern of having wrappers around other configuration calls to encapsulate type conversion. For example, FreePastry implements a method `getInetSocketAddress` that takes an option name as a parameter and returns a socket corresponding to the value of the option. Our analysis discovers that the method uses one of its arguments as an option name. We therefore infer that `getInetSocketAddress` is itself a configuration-reading method and that its return value corresponds to that option name. Finding these additional configuration read calls lets us find the earliest configuration read point in a call chain. This improves the precision of subsequent analyses, including those discussed in the next sections. Effectively, we are treating configuration reads context-sensitively, without the expense of a full-program context-sensitive analysis.

Once we have the set of option read points, we find the string parameters potentially passed to each read call. Most configuration options are named by compile-time constant strings. Sometimes, however, option names are constructed dynamically. A common pattern is to construct configuration option names out of several components, of which just one is variable. For example, in Hadoop, the implementation class used to access a filesystem of type t will be the

value of option `fs.`*t*`.impl`.

We implemented a string analysis to capture option names constructed as a fixed sequence of variable and constant fields, using the any-string regular expression $.*$ to handle dynamic inputs. For the example mentioned above, our code produces the regular expression `fs.\.*\.impl`. This analysis captured nearly all the dynamic option name creation we saw in our programs. This approach was scalable, easy to implement, and integrated cleanly with out points-to framework. More sophisticated string analyses (such as JSA [4]) could be used without changing our overall approach to finding configuration read points.

Configuration options are sometimes used by one part of a program to affect another part, rather than as a way for users to alter the program. In these cases, the lack of documentation for users is not a problem. Accounting for this, we do not report an option as undocumented if its value is set programatically. This requires that we find written options, as well as read ones. Our approach for finding writes is the same as for reads.

We have implemented the above analysis for Java bytecode. Our implementation relies on the JChord program analysis toolkit [23]. Our points-to and call graph construction is context insensitive, flow insensitive, and field sensitive. We use the SSA-representation of the program, as advocated by Hasti and Horwitz [10]. We resolve reflection using the cast-based technique described in [21].

3.2 Results

We evaluated the performance of our technique by running our prototype on each of the seven software packages listed in Section 2. Our research goal was to measure the completeness of the technique, as compared with the existing human-written documentation. When there were mismatches between our output and the documentation, we manually checked the code, searching for substrings of the option names in question. Table 3 represent our best effort at reaching "ground truth" on program configurability in terms of both undocumented and unused options. By "undocumented" options, we mean detected options not mentioned in any user-readable documentation associated with the package, including its website. By unused, we mean options that are never referenced anywhere in reachable code.

Table 4 shows how well our analysis does at matching this manually-generated ground truth. Our analysis found just over 96% of documented options that actually exist. Our tool failed to find uses for 61 documented options. For a third of these, we were able to manually find uses of these options. The remaining two-thirds appear truly unused. In some cases, we found commented-out code referencing these options, suggesting that they used to exist and have since been removed.

Put another way: When our automated analysis and the program's documentation disagree about whether an option exists, the automated approach is more likely to be correct. This also is true on a per-program basis. Our analysis is more accurate than human-written documentation on five of the seven projects.

Our technique will have false positives in cases where the reachability or string-flow analyses are imprecise. We cannot evaluate this directly because we have no ground truth for whether an undocumented option might potentially be read. Note that reachability is not well-defined for the programs in our study, since framework code can be invoked by user code not present at analysis time.

Project	Unused	Opts.	Undocumented	Opts.
Cassandra	2	6%	3	8%
Derby	2	3%	26	38%
FreePastry	24	11%	12	6%
HBase	1	2%	21	33%
Hadoop	6	3%	34	17%
JChord	3	5%	29	51%
Nachos	3	19%	3	19%

Table 3: Manually confirmed documentation errors. Percentages are of all documented options for each project.

Application	Found Unused	True Unused	False positives
Cassandra	3	2	1
Derby	9	2	7
FreePastry	24	24	0
Hadoop	10	6	4
HBase	9	1	8
JChord	3	3	0
Nachos	3	3	0
Total:	61	41	20

Table 4: Accuracy in finding unused options

Looking at the undocumented and unused options, we noticed a number of patterns by which these documentation errors arose. Many undocumented options appear to be new and specialized features, added for some specific purpose and not yet documented. Hadoop and HBase, which are production systems with many users, have disproportionately many of these specialized and undocumented features. Many unused options relate to specialized features, added in the past for exploratory purposes and since removed. As a result, FreePastry, which is an academic system used as a research testbed, has a large number of these unused options. In some cases, code to read the option is present but commented out. JChord is also a research testbed. Here, though, the consequence seems to be that researchers add additional options in their portions of the system without documenting them.[2]

Undocumented options tended to be tunable parameters or Boolean flags, not external identifiers. We conjecture that developers, trained to avoid hard-coding constants, introduce options as a way to specify a constant while retaining flexibility. Some options are described in the program source code as "deprecated;" unexpectedly, these do not appear to be a major source of undocumented options. These options were often still referenced in documentation, even if only to describe them as deprecated.

In Hadoop and Derby, we noticed an additional pattern. Undocumented options were being set in test code and read in the application code. (We filter out options that are set

[2]JChord does not have a formal release process; we are comparing in-development sources to in-development documentation.

in the main body of a program's code. The options we discuss here are set only in unit tests.) These options appear to have been added solely for the benefit of test writers. For instance, Hadoop has several timers controlling activity that normally happens hourly or daily. By setting an undocumented option, unit tests can make the behavior in question occur much more quickly. Test-only options accounted for half the undocumented options in Hadoop and somewhat over half for Derby.

We observed the following pattern by which unused but documented options arise. Sometimes, an option makes sense in the context of a particular implementation of a program feature. Sometimes, the feature is rewritten in such a way as to make the option unnecessary or meaningless. but the documentation is not updated.

3.3 Sources of Error

We now discuss the reasons why our analysis does not find all option uses. By far the largest problem, accounting for 10 of the 20 false positives, was code that performed significant string manipulation on option names. HBase and Derby break the key-value model slightly, iterating over a subset of options and renaming them before use. Our analysis is not sufficiently sensitive to determine the set of names being iterated over. Path sensitivity would be required to model code of this sort accurately. Some errors are caused by more traditional limitations of static analysis. Four options in Derby are read in code invoked indirectly via native code in the system library, where no single-language analysis can easily follow.

Some systems, including Hadoop, do a form of macro substitution for option values. In Hadoop, if the value of a configuration option includes the name of another option, wrapped in braces, the value of that included option is substituted. Nothing in the Hadoop code indicates that the substituted-in option would be read. This caused one false positive for our analysis. This could be handled as a special case in practice: any option lexically included in this way in a configuration file should be marked as used.

Turning from unused options to undocumented options, the biggest limitation we found was actually not a problem with our analysis at all, but with our notion of documentation. There are two significant categories of options picked up by our analysis that are irrelevant for documentation purposes. While programs are expected to document their own options, there is seldom cause to document the options used by included libraries. They are for client programs, not users. Second, not all system properties are configuration options. Java uses system properties both for configuration and also to expose information about the running machine, such as the operating system version. These properties have their values set by the run-time, not by users, so there is no reason for them to be documented.

These limitations are inconveniences, but are unlikely to pose significant problems in practice. Programs and libraries often have a shared prefix for their options (such as `hbase.*`), making it reasonably easy for humans to determine which options belong to which code base. The second problem, that not all system properties are configuration options, could be fixed by white-listing the standard JVM options.

4. CATEGORIZING OPTIONS

In Section 2, we noted that the large majority of the configuration options we encountered belong to one of a fairly small number of types. In the programs we inspected, these types often correspond to specific programming patterns. By finding the pattern, we can infer the type of the option.

We are not attempting to replace human-written documentation. We have three goals: helping developers write documentation, helping developers spot mistakes, and producing machine-usable annotations on options to help user troubleshooting. In all these cases, false negatives are fairly innocuous: an incomplete but accurate analysis is helpful, but false positives can be confusing. Consequently, our analysis is tuned to return "don't know" rather than to make wrong guesses.

Option names, while often descriptive, are sometimes misleading. Hadoop includes an option `mapred.skip.map.auto .incr.proc.count`. Despite the name, this option is actually a Boolean. Printing inferred types alongside the names of undocumented options can help remind programmers what the option does.

Inferring types helps developers check that options are being used in the ways they expect. We have seen options that are read, stored, and logged, but put to no substantive use. The analysis we present here can help developers spot these cases. If the analysis finds an unexpected type for an option, that is suggestive of an underlying bug or incorrect documentation. One striking example concerns the `ij.exceptionTrace` option in Derby. This option is documented as a Boolean, but our analysis was unable to determine a type. On inspecting the code, we discovered that the value was never used at all: the true states of the option were "set" and "unset". Setting any value, even "false" would enable the option; surely an unexpected behavior. We also spotted several options in FreePastry that were documented as being time intervals, but that were never used this way. They were read into the program and logged, but the code to use them was commented out.

This analysis also enables stronger automated checking of configuration files, effectively a "configuration spellcheck" for values as well as option names. This is only meaningful for some option types. For example, almost any string is potentially useable as a file name or password. Fortunately, our analysis works well on most types for which invalid values can be readily flagged.

The approach we take is to look for patterns in how programs use configuration values. This helps validate our taxonomy of configuration options, since a type that corresponds to a well-defined programming pattern is likely to be a useful abstraction. Just as FindBugs [12] and similar tools exploit a library of recognizers for various types of bugs, we envision a separate recognizer for each type of configuration option. Because the number of common option types is limited, the implementation effort required is reasonable.

We have implemented recognizers for most the option types listed above in Table 2: Booleans, class names, files, fractions, network addresses, network interface names, mode names, and port numbers. (We refer to these as "recognized" types). Our analysis splits numeric options into "time", "port number", and "other," rather than attempting to distinguish "counts" from "sizes. In our sample, there were 528 options found by the option-finding analysis and belonging to recognized types. This is the set against which we evaluate our analysis.

4.1 Approach

We exploit three basic techniques in recognizing option types: looking at the return types of configuration reads, looking at which library methods configuration data is passed to, and looking at which values are compared with one another. We assume the presence of a call graph, a points-to analysis, and the results of the above analysis specifying which configuration options are read at each point.

The simplest of our three approaches is to inspect the return type of the configuration call. Many configuration are read using typed methods, such as `getBoolean`, `getFloat`, and so forth. If an option is exclusively read via `getBoolean`, then we conclude that option can only hold Boolean values.

Some programs read options as untyped strings, and then convert them to the correct type. Our second technique handles this case. Instead of looking at how values enter the program, we look at how they leave the program: which library calls they are passed to. If a string is passed to the library `parseBoolean` method, we can infer that its value is expected to be a Boolean. This technique works well for identifiers. There are a small number of common system or library calls for opening files or resolving host names. We use approximately 30 rules to cover the option types in our classification. These rules are stored in a simple lookup table, mapping from called method and argument number to inferred option type. This technique requires a dataflow analysis to connect option reads with subsequent uses.

Mode options, with a handful of valid values, are an important special case. Here, we are often able to not only determine that an option represents a mode choice, but are also able to determine the set of valid values. We have seen only two patterns by which programs use mode variables. Often, programmers parse these options by comparing the returned value against a sequence of string constants. We are able to retrieve this set of valid option names by inspecting the strings a given configuration value is compared with. Alternatively, programmers can define an Enumeration class, and then use a standard library function to create objects of this class. Here, we can retrieve the set of valid values from the associated Enumeration class.

Similarly, we can often infer the parent type that a configurable class must have. When a configurable class name is used to create an object reflectively, and that object is cast to some type T, we infer that the permissible values for the options are subtypes of T.

To help distinguish time-valued options from other options, we employ our third and last technique. There are a handful of library calls for reading the value of the system clock. If an option's value and a known time value are used together in arithmetic, we assume that the option is likewise a time value. This works well in practice: we are able to find 90% of time options, with a 10% false positive rate. This approach is intrinsically imperfect: We have seen configuration options that are multipliers for time values. In these cases, the multiplier should not be marked as "time" – it is a dimensionless number, not a time period.

This analysis runs quickly enough to be used in the software release process. Analyzing Derby, the largest program in our set, took less than 30 minutes using a recent-model laptop with 4 GB of RAM and a dual-core 2.2 GHz processor. Most of the running time was used in the underlying points-to analysis, not in the type inference *per se*.

	Options	Fraction
Real documented opts. of recognized types	528	100%
Success	433	82%
All failures	95	18%
Out of scope	33	6%
Time/not-time miss	27	5%
Wrong guess	12	2%
No guess, other reason	23	4%

Table 5: Overall success rate for type inference and partial breakdown of errors. Percentages are of documented options of recognized types found automatically.

4.2 Implementation

As with the previous analysis, we used JChord to implement our option type determination. Unlike the previous analysis, finding option types requires a whole-program dataflow, tracking the values returned from configuration read points.

To cope with the complex, reflection-heavy programs in our study, we made several modifications to JChord's default algorithms. Most of the programs in our sample are networked services, making heavy use of remote procedure calls (RPCs). This means that much of the code in these programs is never invoked directly; rather, these methods are invoked via reflection. We handle this by specifying a list of extra entry points for each program. These lists had an average of three entries per program.

In Java, method dispatch depends on an object's dynamic type. Correctly modeling method calls on externally-supplied objects therefore required us to modify the underlying points-to analysis. We fall back on type-based alias analysis for these objects [5]. We add an abstract object for every type. When an entry point is invoked externally, we assume that reference arguments point to the appropriately-typed abstract objects. Reference-typed fields in abstract objects point, in turn, to other abstract objects of the appropriate types. In the programs we examined, RPCs are invoked on singleton "server" objects. As a result, this abstraction incurred no loss of precision.

In our experience, configuration options are seldom shared between the program in question and component libraries. To improve performance, we exclude library code from analysis. In most cases, once a value is passed into the library, it ceases to be tracked. In two cases, however, we explicitly model the behavior of library classes. Our points-to analysis is collection aware: if an object allocated at site h_1 is stored into a collection allocated at site h_2, then subsequent reads from that collection can return the object with site h_1. We also explicitly model the string-to-primitive conversion functions in the JVM, thus handling code that, for example, reads an option as a string, converts the string to an integer, and uses that integer as a time delay.

4.3 Results

We compared the results of our automated analysis to the manual labels we described in Section 2. Our results are summarized in Table 5. We look only at options found by the option-finding analysis discussed in Section 3; we want

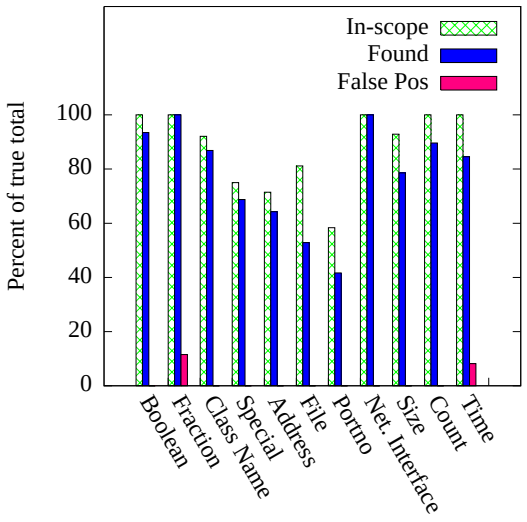

Figure 2: Accuracy in detecting option types. Only includes recognized types. In-scope means the option is used within the program being analyzed, not just in a library or external process.

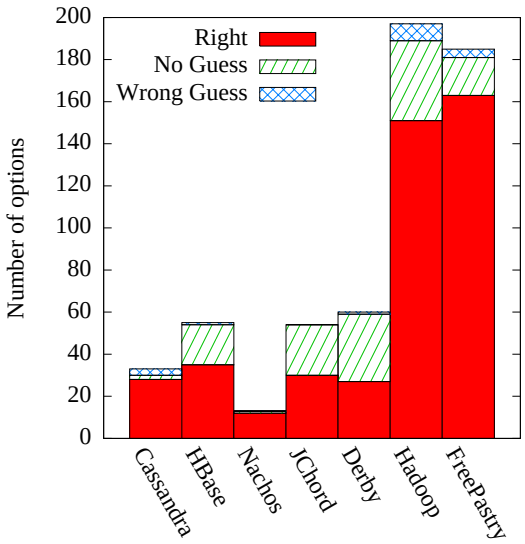

Figure 3: Success by application. Includes all options, not just those of recognized types.

to measure the effectiveness of type inference and option finding separately.

Overall, we succeed 80% of the time. There is substantial variation in success rate by type. Figure 2 displays this variance. False negatives are the gap between 100% and the "Found" bar. False positives (where the analysis finds a wrong type) are labelled explicitly.

The types are ordered based on the primary means we used to recognize them. The first two, Boolean and fractional, are recognized primarily based on the return type of the configuration read call; 24% of options of these types were recognized using called methods, our second technique. The next several columns are recognized entirely by this second technique. Time options, the last column, were found using our second and third techniques. Just over 80% of time options were detected by their use in arithmetic with known clock values; the remaining 20% were found based on their use as arguments to API calls such as `Sleep`.

Our analysis recognizes virtually all options whose values are Boolean, fractional or Java class names. It also does reasonably well for network addresses. Performance on files and port numbers is comparatively weak, for reasons discussed below. In distinguishing times from other numeric parameters, we succeed approximately 90% of the time; errors are roughly symmetric between false positives and false negatives. This imprecision accounted for 28% of misses.

We found few cases where documentation incorrectly described what an option did. We posit that developers add or remove options more frequently than they change the type of an existing option. Changing the meaning of an option would likely break existing configurations, a major enough change to trigger documentation changes. Mistakes in documenting an option's type do sometimes occur, however. The peculiar option in Derby mentioned above where "false" is treated as true is an example.

Despite its limitations, we believe this technique has practical uses. Ignoring the imprecision in detecting time options, incorrect type inferences happened on fewer than 3% of options. Our technique is accurate for several common types that have syntactic constraints. Our analysis is precise enough to accurately warn users when they put a string other than "true" or "false" for a Boolean option, or a similarly invalid value for numerical or class name options.

4.4 Sources of error

This analysis has several sources of error. Sometimes, options are read and then passed to an external process before being used. This was the biggest single source of imprecision for our analysis. These externally-used or "out of scope" options accounted for 30% of all failures to infer a type, and almost half of misses on non-numeric options. This problem showed up primarily for identifiers, particularly file and host names. Since these refer to system abstractions, they can be readily passed to a subprocess or a library. The meaning of a particular class name or Boolean option is more often confined to a single program and therefore the uses for options of these types are generally in-scope.

String operations were another major source of imprecision. We do not track the contents of every string variable. If a configuration value is stored inside a string and then later parsed out, we do not report a type for the option. This is a significant issue with file names, port numbers, and network addresses: there are standard programming idioms (such as constructing a host:port pair) that would defeat our analysis. This could potentially be fixed by a more sophisticated string analysis.

Some of the imprecision in finding time options is due to the presence of numeric utility functions. Since our analysis is context insensitive, we see time options "leak" through these functions. Context-sensitive analysis would help solve this problem. Otherwise, notably, points-to and dataflow imprecision did not appear to be a major problem.

5. DISCUSSION

In the previous two sections, we described how to statically analyze programs to find the set of option names in use and the types of these options. We now discuss how general the problem and approach are.

More careful programming with attention to configuration could reduce the mismatch between programs and documentation, and could catch more user configuration errors. Current versions of Derby and in-development versions of Hadoop attempt to confine option names to a small number of interfaces and classes. Our analysis could help enforce this property during development, since it is efficient enough to run every night alongside the regression test suite.

We focused on key-value style configuration, but there are at least two other common styles for configuration management where this same observation applies. Many programs have graphical configuration management interfaces. Our techniques are potentially applicable to these programs; often, the graphical interface masks an underlying key-value model and not all options are exposed via the graphical interface. Another common model for configuration is structured XML, where the program walks the DOM tree to retrieve values. An advantage of this style for programmers is that schema validation can catch a wide variety of user errors. The downside is that programming with this approach can be more cumbersome. Retrieving an option by name or an XPath query can be done a single line; walking the DOM tree cannot.

The techniques presented in this paper could be generalized to this alternate style. The limitation in generalizing our analysis is matching program points to the DOM nodes they retrieve. Other work has shown that static analysis can model the output (including output XML) from a program fragment [6, 14]; similar techniques may be applicable here.

We focused primarily on large highly configurable systems, with dozens or hundreds of options. Many programs, however, are small and have just a few named configuration options, commonly environment variables. The APIs to retrieve environment variables fall into the key-value pattern we have shown is easy to analyze. Hence, our technique would be useful to extract and model environment dependencies in smaller programs.

Our prototype implementation of our analysis was confined to Java. However, the key-value configuration model we focused on is also used in other contexts. While Java is a comparatively easy language to analyze, our results used fairly simple points-to algorithms, suggesting that optimal points-to accuracy is not required for this domain.

6. RELATED WORK

We describe three areas of related work. We automated efforts to document program behavior, configuration-aware program analysis, and configuration debugging techniques.

The general topic of automatic documentation of program behavior has been addressed previously. Rubio-González and Liblit show that static analysis can catch incorrectly documented error code return values in the Linux kernel [26]. Kremenek *et al.* show that static analysis can find resource allocation and deallocation sites in real-world systems programs, without the benefit of annotations [16]. Buse and Weimer show that the exceptions thrown by Java functions can be inferred more accurately than they are currently documented [3]. Static analysis approaches can also extract higher-level properties of program behavior, such as file or network packet formats [20]. Wang *et al.* use dynamic taint tracking to find security-related options. [32].

Our approach to configuration type inference relies on the fact that many options are used in similar ways and that programmer- and user-oriented descriptions of options are a close match for program structures. Prior work has made similar observations about object oriented design patterns. Reverse-engineering design patterns from program code has been an active area of research since at least 1996. This work sought to report pattern use as a form of design documentation [15, 7].

Like this prior work, we are trying to use program analysis to remedy deficiencies in documentation. Unlike this prior work, we are deriving a comparatively high-level and informal property. Return codes and exceptions are aspects of program behavior that can be directly expressed in the semantics of the associated programming language, as can many object-oriented design patterns. Configuration is a library-defined abstraction; configuration option types are an ad-hoc human concept.

Another branch of related work concerns configuration-aware program analysis. Reisner *et al.* have used symbolic execution to model configurable programs [24]. They show that configuration options often localized effects: most options only affect a small portion of program state. This result is in accord with our findings. We observed that option use falls into patterns; most of the configuration patterns we saw are likely to have localized effects.

Some prior work would benefit from the results of our analysis. The ever-increasing complexity of software has made configuration debugging an important topic. The AutoBash and Triage systems attempt to diagnose failures by repeatedly applying potential fixes and testing them using speculative execution [30, 31]. Knowing which options a program can read and how those values are used may help determine which potential fixes are most promising, thus reducing the time taken to diagnose an error. More precise information about program option use (including option types) may enable automatic generation of potential fixes.

7. CONCLUSION

All of the programs we examined had documentation for options that were not actually present. This appeared to be the result of error, not design. The programs we looked at also contained undocumented options. Some undocumented options appear to be intended only for unit test writers. Others might be useful to users, particularly users with atypical needs. As a result, the lack of documentation is a problem worth correcting.

Several conclusions emerge from our work.

Open-source programs are rife with stale documentation for configuration. Many real options are undocumented, and not all documented options exist.

Configuration option names are available statically. Our static analysis was able to find the overwhelming majority (more than 95%) of options in the programs we looked at. This is more complete than existing manually-produced documentation.

String analysis and external dependencies pose the biggest challenges to static analysis of configurability. Points-to and call graph imprecision was not the biggest

challenge for our analysis. The chief limitations we found were two-fold. Values are sometimes stored inside strings, frustrating simple data-flow analysis. Some options are passed to external programs before being used. This frustrates single-program analysis.

Option behavior can often be described and documented automatically. For some types of options, such as class names and Booleans, analysis can find a high proportion of options, without any observed no false positives. For options drawn from small sets of strings, analysis can often also give the set of legal values. Beyond documentation, this offers the possibility of catching a range of user configuration mistakes, via a "spell check for configuration".

Acknowledgements

We thank Koushik Sen and David Wagner for offering advice and encouragement and thank Mayur Naik for extensive support in using JChord. We thank the anonymous reviewers for their input. This research was supported by California MICRO, California Discovery and the following Berkeley RAD Lab sponsors: Sun Microsystems, Google, Microsoft, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

8. REFERENCES

[1] Apache Derby. `http://db.apache.org/derby/`.

[2] HBase. `http://hbase.apache.org/`.

[3] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *ISSTA*, New York, NY, USA, 2008.

[4] A. Christensen, A. Møller, and M. Schwartzbach. Precise Analysis of String Expressions. In *Symposium on Static Analysis*, 2003.

[5] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. *SIGPLAN Notices*, 33(5), 1998.

[6] K.-G. Doh, H. Kim, and D. A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Symposium on Static Analysis*, 2009.

[7] J. Dong, Y. Zhao, and T. Peng. Architecture and design pattern discovery techniques-a review. In *International Conference on Software Engineering Research and Practice (SERP)*, 2007.

[8] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2009.

[9] Hadoop. `http://hadoop.apache.org/`.

[10] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI*, 1998.

[11] D. Hettena and R. Cox. A guide to nachos 5.0j. `http://inst.eecs.berkeley.edu/~cs162/sp07/Nachos/walk/walk.html`.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):106, 2004.

[13] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.

[14] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Symposium on Static Analysis*, 2006.

[15] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, 1996.

[16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI*, 2006.

[17] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. In *SIGMOD*, 2008.

[18] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE software*, pages 35–39, 2003.

[19] M. Levesque. Fundamental issues with open source software development. *First Monday:*, Special Issue #2: Open Source, October 2005.

[20] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Working Conference on Reverse Engineering*, 2006.

[21] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Third Asian Symposium on Programming Languages and Systems*, 2005.

[22] M. Michlmayr, F. Hunt, and D. Probert. Quality practices and problems in free software projects. In *Proceedings of the First International Conference on Open Source Systems*, 2005.

[23] M. Naik. JChord. `http://jchord.googlecode.com`.

[24] E. Reisner, C. Song, K. Ma, J. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[26] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *PASTE*, 2010.

[27] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. In *Automated Software Engineering*, 2009.

[28] D. Schreck, V. Dallmeier, and T. Zimmermann. How documentation evolves over time. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, 2007.

[29] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[30] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[31] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.

[32] R. Wang, X. Wang, K. Zhang, and Z. Li. Towards automatic reverse engineering of software security configurations. In *CCS*, 2008.