A Factor Graph Model for Software Bug Finding

Ted Kremenek Andrew Y. Ng Dawson Engler
Computer Science Department
Stanford University
Stanford, CA 94305 USA

Abstract

Automatic tools for finding software errors require knowledge of the rules a program must obey, or "specifications," before they can identify bugs. We present a method that combines factor graphs and static program analysis to automatically infer specifications directly from programs. We illustrate the approach on inferring functions in C programs that allocate and release resources, and evaluate the approach on three codebases: SDL, OpenSSH, and the OS kernel for Mac OS X (XNU). The inferred specifications are highly accurate and with them we have discovered numerous bugs.

1 Introduction

Software bugs are as pervasive as software itself, with the rising cost of software errors recently estimated to cost the United States economy $59.5 billion/year [RTI, 2002]. Fortunately, there has been a recent surge of research into developing automated and practical bug-finding tools. While tools differ in many respects, they are identical in one: if they do not know what properties to check, they cannot find bugs. Fundamentally, all tools essentially employ *specifications* that encode what a program *should do* in order to distinguish good program behavior from bad.

A specification is a set of rules that outlines the acceptable behavior of a program. A "bug" is a violation of the rules. For example, one universal specification is that a program "should not crash." Because crashes are fail-stop errors (i.e., the program halts) they are easy to detect, but because many factors can lead to a crash they are equally difficult to diagnose.[1] Moreover, while software crashes are colorful symptoms of a program behaving badly, many bugs are not fail-stop. Memory leaks (or more generally leaks of application and system resources) lead through attrition to the gradual death of a program and often induce erratic behavior along the way. Data corruption errors can lead to unpleasant results such as loss of sensitive data. Further, most security-related bugs, such as those allowing a system to be compromised, are not fail-stop. Properly locating and fixing such bugs requires knowledge of the violated program invariants.

Traditionally, discovering such bugs was the responsibility of software developers and testers. Fortunately, automated bug-finding tools, such as those based on static program analysis, have become adroit at finding many such errors. Operationally, a static analysis tool analyzes a program without running it (similar to a compiler) and reasons about the possible paths of execution through the program. Conceptually, the tool checks that every analyzed path obeys the program invariants in a specification. When a rule can be violated, the tool flags a warning. Static tools have achieved significant success in recent years, with research tools finding thousands of bugs in widely used open-source software such as the Linux kernel [Engler *et al.*, 2000]. Unfortunately there is no free lunch. Like human testers, these tools require knowledge of a program's specification in order to find bugs.

The key issue is possessing adequate specifications. Unfortunately, many important program properties that we could check with automated tools are domain-specific and tied to a particular API or program. To make matters worse, the manual labor needed to specify high-level invariant properties can be overwhelming even for small programs [Flanagan *et al.*, 2002]. Further, in large evolving codebases the interfaces may quickly change, which further complicates the problem of keeping a specification current. Consequently, many bugs that could have been found with current tools are rendered invisible by ignorance of the necessary specifications.

This paper describes a technique that combines factor graphs with static program analysis to automatically infer specifications directly from programs. We illustrate the kind of specifications that can be inferred with an example specification inference task. This paper formalizes and extends the model informally introduced in our earlier work [Kremenek *et al.*, 2006]; we also describe algorithms for inference and parameter learning. These changes result in significantly improved performance of the model. We also apply these ideas to finding a number of bugs, many serious, in SDL, OpenSSH, PostgreSQL, Wine and Mac OS X (XNU).

1.1 Specifications of Resource Ownership

Almost all programs make use of dynamically allocated resources. Examples include memory allocated by functions like `malloc`, file handles opened by calls to `fopen`, sockets,

[1] This has led to significant work on post-mortem analysis of software crashes, including applying machine learning methods, to identify potential causes of a crash [Zheng *et al.*, 2003].

```
1.  FILE * fp1 = fopen( "myfile.txt", "r" );
2.  FILE * fp2 = fdopen( fd, "w" );
3.  fread( buffer, n, 1, fp1 );
4.  fwrite( buffer, n, 1, fp2 );
5.  fclose( fp1 );
6.  fclose( fp2 );
```

Figure 1: Example use of standard C I/O functions.

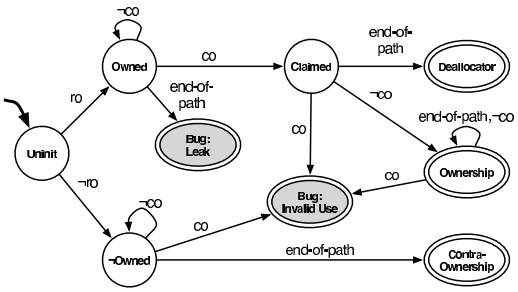


Figure 2: DFA for a static analysis checker to find resource errors. Shaded final states represent error states (bugs).

database connections, and so on. Functions that "allocate" resources, or *allocators*, typically have a matching *deallocator* function, such as `free` and `fclose`, that releases the resource. Even if a language supports garbage collection, programmers usually must enforce manual discipline in managing arbitrary allocated resources in order to avoid resource-related bugs such as leaks or "use-after-release" errors.

Numerous tools have been developed to find resource bugs, with the majority focusing on finding bugs for uses of well-known allocators such as `malloc` [Heine and Lam, 2003]. Many systems, however, define a host of allocators and deallocators to manage domain-specific resources. Because the program analysis required to find resource bugs is generally the same for all allocators and deallocators, current tools could readily be extended to find resource bugs for domain-specific allocators if they were made aware of such functions.

A more general concept, however, that subsumes knowing allocators and deallocators is knowing what functions *return* or *claim ownership* of resources. To manage resources, many programs employ the ownership idiom: a resource has at any time exactly one *owning* pointer (or handle) which must eventually release the resource. Ownership can be transferred from a pointer by storing it into a data structure or by passing it to a function that claims it (e.g., a deallocator). Although allocators and deallocators respectively return and claim ownership, many functions that return ownership have a contract similar to an allocator but do not directly allocate resources; e.g., a function that dequeues an object from a linked list and returns it to the caller. Once the object is removed from the list, the caller must ensure that the object is fully processed. A similar narrative applies to functions that claim ownership. By knowing all functions that return and claim ownership, we can detect a wider range of resource bugs.

This paper explores the problem of inferring domain-specific functions in C programs that return and claim ownership. Our formulation uses an encoding of this set of functions that is easily consumed by a simple static analysis tool, or *checker*, which we briefly describe. Figure 1 depicts a contrived code fragment illustrating the use of several standard I/O functions in C. For the return values of `fopen` and `fdopen`, we can associate the label ro (returns ownership) or $\neg ro$. For the input arguments (with a pointer type) of `fwrite`, `fread`, and `fclose` we can associate labels co (claims ownership) or $\neg co$. These labels can be used by a simple checker that operates by tracing the possible paths within the function where `fp1` and `fp2` are used, and, along those paths, simulate for each pointer the property DFA in Figure 2. Every time a pointer is passed as an argument to a function call or returned from a function the corresponding label is consulted and the appropriate transition is taken. An "end-of-path" indicates that the end of the function was reached. There are five final states. The states *Leak* and *Invalid Use* are error states (shaded) and indicate buggy behavior (*Invalid Use* captures both "use-after-release" errors as well as claiming a non-owned pointer). The other final states indicate a pointer was used correctly, and are discussed later in further detail. Further details regarding the implementation of the checker can be found in Kremenek et al. [2006].

1.2 Our Approach

While specifications conceivably come in arbitrary forms, we focus on inferring specifications where **(1)** the entire set of specifications is discrete and finite and **(2)** a given specification for a program can be decomposed into elements that describe the behavior of one aspect of the program. For example, in the ownership problem if there are m functions whose return value can be labeled ro and n function arguments that can be labeled co then there are $2^m 2^n$ possible combined labellings. In practice, there are many reasonable bug-finding problems whose specifications map to similar domains.

Our primary lever for inferring specifications is that programs contain latent information, often in the form of "behavioral signatures," that indirectly documents their high-level properties. Recall that the role of specifications is to outline acceptable program behavior. If we assume that programs for the most part do what their creators intended (or at least in a relative sense "bugs are rare") then a likely specification is one that closely matches the program's behavior. Thus, if such a specification was fed to a static analysis checker, the checker should flag only a few cases of errant behavior in the program. Finally, latent information may come in a myriad of other forms, such as naming conventions for functions (e.g. "alloc") that provide hints about the program's specification.

This motivates an approach based on probabilistic reasoning, which is capable of handling this myriad of information that is coupled with uncertainty. Our solution employs factor graphs [Yedidia *et al.*, 2003], where a set of random variables in a factor graph represent the specifications we desire to infer, and factors represent constraints implied by behavioral signatures. The factor graph is constructed by analyzing a program's source code, and represents a joint probability distribution over the space of possible specifications. Once the factor graph is constructed, we employ Gibbs sampling to infer the most likely specifications.

2 Factor Graph Model

We now present our factor graph model for inferring specifications. We illustrate it in terms of the ownership problem, and make general statements as key concepts are introduced.

We begin by mapping the space of possible specifications to random variables. For each element of the specification with discrete possible values we have a random variable A_i with the same domain. For example, in the ownership problem for each return value of a function "`foo`" in the codebase we have a random variable $A_{\mathsf{foo:ret}}$ with domain $\{ro, \neg ro\}$. Further, for the ith argument of a function "`baz`" we have a random variable $A_{\mathsf{baz}:i}$ with domain $\{co, \neg co\}$. We denote this collection of variables as $\mathbf{A}$, and a compound assignment $\mathbf{A} = \mathbf{a}$ represents one complete specification from the set of possible specifications.

2.1 Preliminaries

Our goal is to define a joint distribution for $\mathbf{A}$ with a factor graph. We now review key definitions pertaining to factor graphs [Yedidia *et al.*, 2003].

Definition 1 (Factor) *A factor f for a set of random variables $\mathbf{C}$ is a mapping from val$(\mathbf{C})$ to $\mathbb{R}^+$.*

Definition 2 (Gibbs Distribution) *A Gibbs distribution $\boldsymbol{P}$ over a set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ is defined in terms of a set of factors $\{f_j\}_{j=1}^J$ (with associated random variables $\{\mathbf{C}_j\}_{j=1}^J$, $\mathbf{C}_j \subseteq \mathbf{X}$) such that:*

$$\boldsymbol{P}(X_1, \ldots, X_n) = \frac{1}{Z} \prod_{j=1}^{J} f_j(\mathbf{C}_j). \quad (1)$$

The normalizing constant Z is the partition function.

Definition 3 (Factor Graph) *A* factor graph *is a bipartite graph that represents a Gibbs distribution. Nodes correspond to variables in $\mathbf{X}$ and to the factors $\{f_j\}_{j=1}^J$. Edges connect variables and factors, with an undirected edge between X_i and f_j if $X_i \in \mathbf{C}_j$.*

2.2 Overview of Model Components

We now define the factors in our model. Maintaining terminology consistent with our previous work, we call the factor graphs constructed for specification inference *Annotation Factor Graphs* (AFGs). The name follows from that the specifications we infer (e.g. ro and co) serve to "annotate" the behavior of program components. While often the only random variables in an AFG will be $\mathbf{A}$ (i.e., $\mathbf{X} = \mathbf{A}$), other variables, such as hidden variables, can be introduced as needed.

There are two categories of factors in an AFG that are used to capture different forms of information for specification inference. The first set of factors, called *check factors*, are used to extract information from observed program behavior. A given specification $\mathbf{A} = \mathbf{a}$ that we assign to the functions in a program will determine, for each tracked pointer, the outcome of the checker described in Section 1.1. These outcomes reflect behaviors the checker observed in the program given the provided specification (e.g., resource leaks, a pointer being properly claimed, and so on). Our insight is that **(1)** some behaviors are more likely than others (e.g., errors should occur rarely) and that **(2)** some behaviors are harder for a program to "coincidentally" exhibit; thus when we observe such behaviors in a program they may provide strong evidence that a given specification is likely to be true. Check factors incorporate into the AFG both our beliefs about such behaviors and the mechanism (the checker) used to determine what behaviors a program exhibits.

The second set of factors are used to model arbitrary domain-specific knowledge. This includes prior beliefs about the relative frequency of certain specifications, knowledge about suggestive naming conventions (e.g, the presence of "alloc" in a function's name implies it is an ro), and so on.

We now discuss both classes of factors in turn.

2.3 Check Factors

We now describe check factors, which incorporate our beliefs about the possible behaviors a program may exhibit and the specifications they imply.

Each final state in our checker corresponds to a separate behavioral signature observed in the program given a specification $\mathbf{A} = \mathbf{a}$. The checker observes five behavioral signatures, two of which indicate different kinds of bugs (leaks and everything else), and three which identify different kinds of "good" behavior. By distinguishing between different behaviors, we can elevate the probability of values of $\mathbf{A}$ that induce behaviors that are more consistent with our beliefs.

First, we observe that (in general) bugs occur rarely in programs. Although not a perfect oracle, the checker can be employed to define an error as a case where the DFA in Figure 2 ends in an error state. Thus an assignment to $\mathbf{A}$ that causes the checker to flag many errors is less likely than an assignment that leads to few flagged errors. Note that we should not treat errors as being *impossible* (i.e., only consider specifications that cause the checker to flag no errors) because **(1)** real programs contain bugs and **(2)** the checker may flag some false errors even for a bug-free program.

Further, not all kinds of errors occur with equal frequency. In practice *Invalid Use* errors occur far less frequently than *Leak*s. Thus, for two different specifications that induce the same number of errors, the one that induces more *Leak*s than *Invalid Use* errors is the more likely specification.

Finally, errors aside, we should not equally weight observations of different kinds of good program behavior. For example, the *Deallocator* signature recognizes the pattern that once an owned pointer is claimed it is never subsequently used, while *Ownership* matches behavior that allows a claimed pointer to be used after it is claimed. The former is a behavioral signature that is much harder for a set of functions to fit by chance. Consequently when we observe the *Deallocator* pattern we could potentially weight it as stronger evidence for a given specification than if a code fragment could only obey the *Ownership* pattern. Finally, the *Contra-Ownership* pattern, which recognizes all correct use of a non-owned pointer, is the easiest pattern to fit: all functions can be labeled $\neg ro$ and $\neg co$ and the checker will never flag an error. Such a specification is useless, however, because we wish to infer ro and co functions! Thus we should potentially "reward" observations of the *Ownership* or *Deallocator* signatures more than the *Contra-Ownership* pattern. In other words, we are willing to tolerate some errors if a set

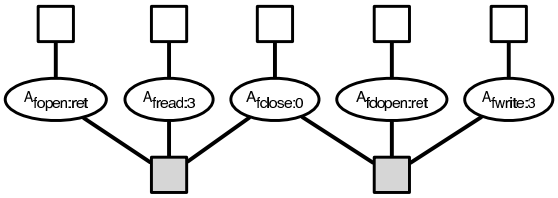


Figure 3: Factor graph model for the code in Figure 1. Circular nodes correspond to variables and square nodes to factors. The shaded factors indicate check factors, while the top row depicts factors modeling prior beliefs.

of functions appear to consistently fit either the *Deallocator* or *Ownership* signatures.

We now discuss how these ideas are modeled using factors. We first atomize the output of the checker into *checks*. A check is a distinct instance in the program where the specification could be obeyed or disobeyed. For the ownership problem, we have a check for every statement of the form "`p = foo()`" where a pointer value is returned from a called function. For the code in Figure 1 we have one check for `fp1` and another for `fp2`. In general, the actual definition of a check will depend on the specifications we are trying to infer, but essentially each check represents a distinct observation point of a program's behavior.

Once we define the set of checks for a codebase, for each check we create a corresponding *check factor*, denoted $f_{check_{(i)}}$, in the AFG. Check factors represent **(1)** the analysis result of the checker at each check when running the checker using a provided set of values for $\mathbf{A}$ and **(2)** our preferences over the possible outcomes of each check. The variables in $\mathbf{A}$ associated with a given $f_{check_{(i)}}$, denoted $\mathbf{A}_{check_{(i)}}$, are those whose values could be consulted by the checker to determine the check's outcome. For example, Figure 3 depicts the factor graph for the code example in Figure 1. We have two check factors (shaded), one for `fp1` and `fp2` respectively. Because for `fp1` the checker needs only consult the specifications represented by the variables $A_{\text{fopen:ret}}$, $A_{\text{fread:4}}$ and $A_{\text{fclose:1}}$, these variables are those associated with $f_{check_{(\texttt{fp1})}}$.

Check factors have a simple mathematical definition. If $\mathcal{C}_i(\mathbf{a}_{check_{(i)}})$ represents the output of the checker for check i when $\mathbf{A}_{check_{(i)}} = \mathbf{a}_{check_{(i)}}$, then $f_{check_{(i)}}$ is defined as:

$$f_{check_{(i)}}(\mathbf{a}_{check_{(i)}}) = \begin{cases} e^{\theta_c} & : \text{ if } \mathcal{C}_i(\mathbf{a}_{check_{(i)}}) = c \end{cases}$$

Thus a check factor is encoded with a set of real-valued parameters ($\theta_c \in \mathbb{R}$), one for each distinct behavior observed by the checker. These parameters are shared between all check factors that observe the same set of behaviors,[2] and are used to encode our intuitions about program behavior and the specifications they imply. For example, we expect that the parameters for error states, $\theta_{\textit{Leak}}$ and $\theta_{\textit{Invalid Use}}$, will have lower values than the remaining parameters (i.e., errors are rare). While parameters can be specified by hand [Kremenek *et al.*, 2006], in this paper we focus on learning them from partially known specifications and observing if the learned parameters both **(1)** match with our intuitions and **(2)** compare in quality to the specifications inferred using hand-tuned parameters.

[2]Multiple checkers with different check factors can conceptually be used to analyze the program for different behavioral profiles.

Multiple execution paths. Note that the value of the check is a summary of *all* the analyzed paths within the function for that pointer. Each analyzed path may end in a different state in the DFA. Instead of reporting results for all analyzed paths, we summarize them by reporting the final state from the analyzed paths that appears earliest in the following partial order:

$$\begin{aligned} \textit{Invalid Use} &\prec \textit{Leak} \prec \textit{Contra-Ownership} \\ &\prec \textit{Ownership} \prec \textit{Deallocator} \end{aligned}$$

For example, if on any path the analysis encounters an *Invalid Use* state, it reports *Invalid Use* for that check regardless of the final states on the other paths. The idea is to report bad behavior over good behavior.

2.4 Further Modeling: Domain Knowledge

Beyond exploiting the information provided by a checker, the factor graph allows us to incorporate useful domain knowledge. We discuss two examples for the ownership problem.

Prior beliefs. Often we have prior knowledge about the relative frequency of different specifications. For example, most functions do not claim ownership of their arguments and should be labeled $\neg co$. Such hints are easily modeled as a single factor attached to each A_i variable. We attach to each $A_{\text{foo:}i}$ a factor $f(A_{\text{foo:}i} = x) = e^{\theta_x}$. The two parameters, θ_{co} and $\theta_{\neg co}$, are shared between all factors created by this construction. Analogously we define similar factors for each $A_{\text{foo:ret}}$. These factors are depicted at the top of Figure 3.

Suggestive naming. Naming conventions for functions (e.g., a function name containing "alloc" implies the return value is ro) can be exploited in a similar fashion. We selected a small set of well-known keywords $\mathcal{K}$ ($|\mathcal{K}| = 10$) containing words such as "alloc", "free" and "new." To model keyword correlation with ro specifications, for each $A_{\text{foo:ret}}$ whose functions contains the keyword kw we construct a single factor associated with $A_{\text{foo:ret}}$:

$$f(A_{\text{foo:ret}} = x) = e^{\theta_{kw:x}} \tag{2}$$

Since $x \in \{ro, \neg ro\}$ this factor is represented by two parameters (per keyword). These parameters are shared between all factors created by this construction. Note that the factor is present only if the function has the keyword as a substring of it's name; while the presence of a keyword may be suggestive of a function's role, we have observed the absence of a keyword is usually uninformative.

Keyword correlation for co specifications is similarly modeled, except since a function may contain multiple arguments, each of which may be labeled co, we construct one "keyword factor" over all the $A_{\text{foo:}i}$ variables, denoted $\mathbf{A}_{\langle \texttt{foo:}parms \rangle}$, for a function `foo`:

$$f(\mathbf{a}_{\langle \texttt{foo:}parms \rangle}) = e^{\theta_{kw:co}\mathbb{I}\{\exists i | A_{\text{foo:}i} = co\} + \theta_{kw:\neg co}\mathbb{I}\{\forall i | A_{\text{foo:}i} = \neg co\}} \tag{3}$$

Thus, if *any* of `foo`'s arguments has the specification co then the factor has value $e^{\theta_{kw:co}}$ (and $e^{\theta_{kw:\neg co}}$ otherwise). For clarity, keyword factors have been omitted in Figure 3.

3 Inference

Once the factor graph is constructed, we employ Gibbs sampling to sample from the joint distribution. For each A_i

we estimate the probability it has a given specification (e.g., $P(A_i = ro)$) and rank inferred specifications by their probabilities. Analogously, we estimate for each check factor $f_{check_{(i)}}$ the probability that the values of $\mathbf{A}_{check_{(i)}}$ cause the checker to flag an error. This allows us to also rank possible errors by their probabilities.

When updating a value for a given $A_j \in \mathbf{A}$, we must recompute the value of each check factor $f_{check_{(i)}}$ where $A_j \in \mathbf{A}_{check_{(i)}}$. This requires actually running the checker. Because at any one time Gibbs sampling has a complete assignment to all random variables, the checker simply consults the current values of $\mathbf{A}$ to determine the outcome of the check. This clean interface with the checker is the primary reason we employed Gibbs sampling.

While our checker is relatively simple, the analysis is still very expensive when run repeatedly. To compensate, we cache analysis results by monitoring which values of $\mathbf{A}$ are consulted by the checker to determine the outcome of a check. This results in a speedup of two orders of magnitude.

We experienced serious issues with mixing. This is a by-product of the check factors, since values of several A_i variables may need to be flipped before the outcome of a check changes. We explored various strategies to improve mixing, and converged to a simple solution that provided consistently acceptable results. We run 100 chains for $N = 1000$ iterations and at the end of each chain record a *single* sample. Moreover, for each chain, we apply the following annealing schedule so that each factor f_i has the following definition on the kth Gibbs iteration:

$$f_i^{(k)}(\mathbf{A}_i) = f_i(\mathbf{A}_i)^{min(\frac{k}{0.75N},1)} \tag{4}$$

This simple strategy significantly improved the quality of our samples. While satisfied by the empirical results of this procedure, we continue to explore faster alternatives.

4 Learning

We now discuss our procedure for parameter learning. The factors we have discussed take the exponential form of $f(\mathbf{C}_j = \mathbf{c}_j) = e^{\theta_{\mathbf{c}_j}}$ (with $\theta_{\mathbf{c}_j} \in \mathbb{R}$). The set of parameters $\boldsymbol{\theta}$ for these factors can be learned from (partially) observed data, denoted $\mathbf{D} = \mathbf{d}$, by using gradient ascent to maximize the log-likelihood of $\mathbf{d}$. Generally $\mathbf{D} \subset \mathbf{A}$, representing partially known specifications. We omit the derivation of the gradient, as it is fairly standard. For the case where a single parameter $\theta_{\mathbf{c}_j}$ appears in a single factor f_j, the corresponding term of the gradient is:

$$\frac{\partial \log p(\mathbf{d}|\boldsymbol{\theta})}{\partial \theta_{\mathbf{c}_j}} = \mathbf{E}_{\boldsymbol{P}_{Clamped}}[\mathbb{I}_{\{\mathbf{C}_j=\mathbf{c}_j\}}] - \mathbf{E}_{\boldsymbol{P}_{Unclamped}}[\mathbb{I}_{\{\mathbf{C}_j=\mathbf{c}_j\}}] \tag{5}$$

Here $\boldsymbol{P}_{Clamped}$ represents the conditional distribution over all variables in the factor graph when $\mathbf{D}$ is observed, while $\boldsymbol{P}_{Unclamped}$ represents the distribution with no observed data. If a parameter appears in multiple factors, the gradient term for $\theta_{\mathbf{c}_j}$ is summed over all factors in which it appears.

4.1 Implementation: Heuristics and Optimizations

We now briefly describe a few key features of our implementation of gradient ascent for our domain.

		AFG Size		Manually Classified Specifications						
Codebase	Lines (10^3)	$\lvert\mathbf{A}\rvert$	# Checks	ro	$\neg ro$	$\frac{ro}{\neg ro}$	co	$\neg co$	$\frac{co}{\neg co}$	Total
SDL	51.5	843	**577**	35	25	1.4	16	31	0.51	**107**
OpenSSH	80.12	717	**3416**	45	28	1.6	10	108	0.09	**191**
XNU	1381.1	1936	**9169**	35	49	0.71	17	99	0.17	**200**

Table 1: Breakdown by project of codebase size, number of manually classified specifications, and AFG size.

Seeding parameters. All parameters, excluding θ_{Leak} and $\theta_{Invalid\ Use}$, were initialized to a value of 0 (i.e., no initial bias). θ_{Leak} and $\theta_{Invalid\ Use}$ were initialized to -1 to provide a slight bias against specifications that induce buggy behavior.

Estimating the gradient. For each step of gradient ascent, the expectations in Equation 5 are estimated using Gibbs sampling, but each with only *two* chains (thus relying on properties of stochastic optimization for convergence). Consequently, our estimate of the gradient may be highly noisy. To help mitigate such noise, samples are drawn from $\boldsymbol{P}_{Clamped}$ and $\boldsymbol{P}_{Unclamped}$ in a manner similar to *contrastive divergence* [Hinton, 2000]. First, each sample from $\boldsymbol{P}_{Clamped}$ is sampled as described in Section 3. To generate a sample from $\boldsymbol{P}_{Unclamped}$, we continue running the Markov chain that was used to sample from $\boldsymbol{P}_{Clamped}$ by **(1)** unclamping the observed variables $\mathbf{D}$ and then **(2)** running the chain for 400 more iterations. This noticeably reduces much of the variation between the samples generated from $\boldsymbol{P}_{Clamped}$ and $\boldsymbol{P}_{Unclamped}$.

Because the term for $\theta_{\mathbf{c}_j}$ in the gradient is additive in the number of factors that share $\theta_{\mathbf{c}_j}$, its value is in the range $[-NumFactors(\theta_{\mathbf{c}_j}), NumFactors(\theta_{\mathbf{c}_j})]$. This causes the magnitude of the gradient to grow with the size of the analyzed codebase. To compensate, we scale each $\theta_{\mathbf{c}_j}$ term of the gradient by $NumFactors(\theta_{\mathbf{c}_j})$, leaving each term of the modified gradient in the range $[-1, 1]$. This transformation, along with a modest learning rate, worked extremely well. We experimented with alternate means to specify learning rates for gradient ascent, and none met with the same empirical success.

Finally, since an AFG typically consists of multiple connected components, if a connected component contains no observed variables, then Equation 5 is trivially 0 for all factors in the component. We thus prune such components from the factor graph prior to learning.

5 Evaluation

We evaluate our model by inferring ro and co functions in three codebases: SDL, OpenSSH, and the OS kernel for Mac OS X (XNU). SDL is a cross-platform graphics library for game programming. OpenSSH consists of a network client and server for encrypted remote logins. Both manage many custom resources, and SDL uses infrequently called memory management routines from XLib. Like all OS kernels, XNU defines a host of domain-specific routines for managing resources. For each project we randomly selected and manually classified 100-200 specifications for the return values (ro or $\neg ro$) and arguments (co or $\neg co$) of functions. Table 1 shows the size of each codebase, the number of manual classifications, and AFG sizes.

5.1 Specification Accuracy

Our hypothesis is that many codebases will exhibit similarities in code structure and style, allowing a model trained on one codebase to be applied to another. We evaluate this hypothesis with two experiments.

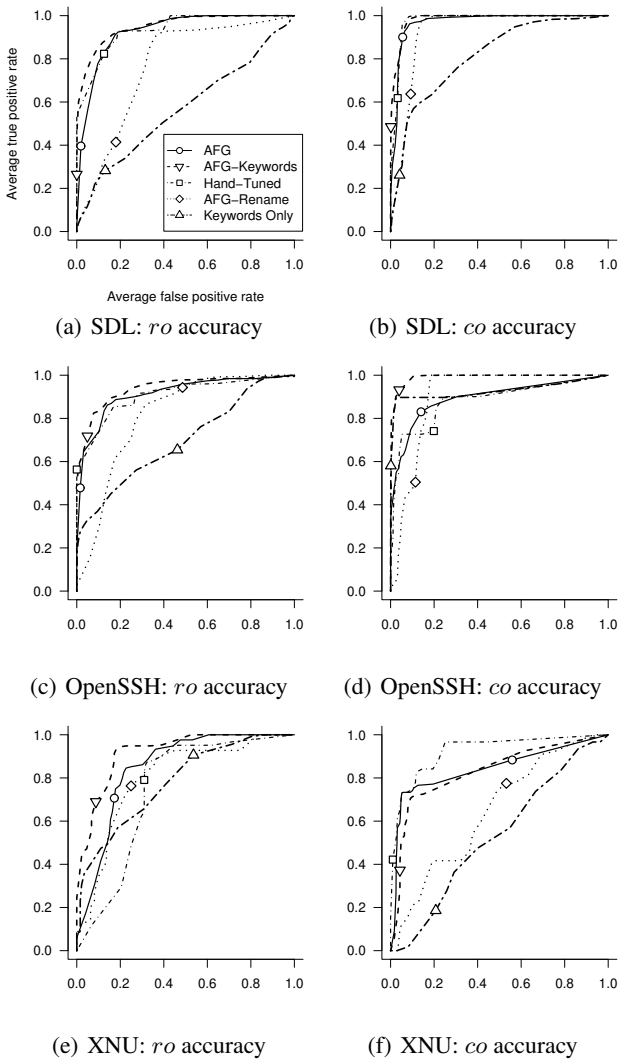
First, for each project we randomly divide our known specifications (Table 1) into training and test sets ($60/40\%$). We train the model on the training set and then use the trained model to infer the specifications in the test set. Because the strictest test of our model is to apply it to a codebase with no known specifications, when inferring specifications for the test set, *none* of the variables in $\mathbf{A}$ are observed (including those in the training set). This simulates applying a model to a codebase that has practically identical code characteristics to the codebase on which the model was trained. We repeat this experiment 10 times.

Figure 4 depicts averaged ROC curves for each project. Each figure depicts five lines. The base model, *AFG*, is an AFG that includes check factors and factors to model prior beliefs over ro/co labels. The second line, *AFG-Keywords*, is *AFG* augmented with keyword factors. *Hand-Tuned* is the *AFG* model using parameters that were tuned by hand over time by inspecting inference results on all codebases.

The remaining two lines represent an ablative analysis, where we test simpler systems that use only a subset of the features of the full system. One strength of the model is that it captures the inter-correlation between specifications across the codebase. *AFG-Rename* is constructed from *AFG* by weakening the correlation between variables: each variable $A_i \in \mathbf{A}$ is replicated for each associated check factor (this is equivalent to renaming each function call in the codebase to refer to a distinct function). For example, for the AFG in Figure 3, we split $A_{\mathsf{fopen:ret}}$ into two random variables, one for each of the two check factors for which $A_{\mathsf{fopen:ret}}$ shares an edge. These two variables then serve as substitutes to $A_{\mathsf{fopen:ret}}$ for the respective check factors. Specification probabilities are then estimated by averaging the probabilities of the replicated variables. The remaining model, *Keywords Only*, is an AFG that includes only keyword and prior belief factors. All models, with the exception of *Hand-Tuned*, had their parameters separately learned on the same data.

The ROC curves illustrate that our model generally performs very well. For SDL, *AFG*, *AFG-Keywords*, and *Hand-Tuned* achieve between 90-100% true positive rate (TPR) for both ro and co specifications with a 10% (or less) false positive rate. It is encouraging our trained models perform as well or better as *Hand-Tuned* (which essentially had access to both training and test set data for all codebases), with *AFG-Keywords* slightly edging out all other models. We observe similar results on OpenSSH and XNU. On XNU, both *AFG* and *AFG-Keywords* significantly outperforms *Hand-Tuned* for ro accuracy, with *Hand-Tuned* achieving higher co accuracy with the trade-off of lower ro accuracy.

Our ablated models perform significantly worse. For SDL and OpenSSH, *AFG-Rename* has noticeably degraded ro accuracy compared to *AFG*, but maintains decent co accuracy (the reverse being the case on XNU). We believe this is due to the richer models capturing relationships such as several



(a) SDL: ro accuracy (b) SDL: co accuracy

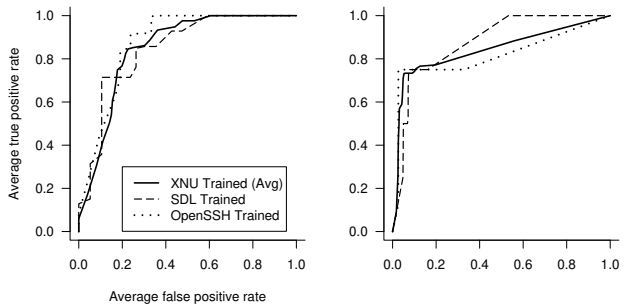
(c) OpenSSH: ro accuracy (d) OpenSSH: co accuracy

(e) XNU: ro accuracy (f) XNU: co accuracy

Figure 4: ROC curves depicting inferred specification accuracy.

allocators being paired with a common dealloactor function (thus information about one propagates to the others). Note that its performance is still significantly better than random guessing. This suggests that when the ownership idiom fits at a "local" level in the code it is still strongly suggestive of a program's specification. For *Keywords-Only*, we observe excellent co accuracy on OpenSSH because of the small number of co functions with very suggestive names, while for similar reasons it has decent co accuracy on SDL (up to the 50% TPR level, at which point accuracy falls off). On XNU, co accuracy is worse than random. On all codebases its ro accuracy is modest to poor; a more careful analysis suggests that some naming conventions are used inconsistently and that many ro functions do not have suggestive names.

Our second experiment directly evaluates training the model parameters on one codebase and applying them to inferring specifications on another. Figure 5 depicts specification inference results for XNU. The SDL and OpenSSH parameters are trained using our full set of known specifications for those projects and then are tested on our full set of known



(a) XNU: ro accuracy (b) XNU: co accuracy

Figure 5: Specification accuracy on XNU when using model parameters trained on SDL and OpenSSH.

specifications for XNU, while *XNU (Avg)* is the *AFG* line from the previous experiment. All models are *AFG* (without keywords). Graphs for the other codebases are similar. We observe in this figure that all the lines are very close to each other. We believe this strongly supports the generality of the model and its applicability across codebases.

Interpreting parameters. Upon inspection, in most case learned parameters matched well with our intuitions (Section 2.3). For all codebases, the parameters for error states, θ_{Leak} and $\theta_{Invalid\ Use}$, were less than the remaining parameters for check factors (non-errors). On some codebases, however, their relative values were higher to (we believe) compensate for increased codebase-specific noise from the checker. Consequently, our AFG model can compensate for some deficiencies in the checker as long as the checker can identify informative behavioral patterns. We also observed that $\theta_{Deallocator}$ was always greater than $\theta_{Ownership}$ and $\theta_{Contra\text{-}Ownership}$, which matches with our intuition that observations of the *Deallocator* pattern should be "rewarded" higher than other behaviors.

5.2 Software Bugs

As discussed in Section 3, we can use the results from Gibbs sampling to rank possible bugs by their probabilities before examining any inferred specifications. This enabled us to quickly find errors in each codebase that are based on the specifications inferred with the highest confidence. We observed about a 30-40% false positive rate for flagged errors (a rate consistent with current static checking tools). Most false positives were due to static analysis imprecision (a source of noise that our model appears to handle well when inferring specifications), with a few due to misclassified specifications.

In practice, we may feed the inferred specifications into a more precise (and expensive) static analysis (e.g., Xie et al. [2005]) when actually diagnosing bugs. Nevertheless, even with our simple checker we discovered 3 bugs in SDL and 10 bugs in OpenSSH. For XNU, many bugs were still pending confirmation, but 4 bugs were confirmed by developers, including one serious error (discussed below). We also casually applied our model to other projects including PostgreSQL (a relational database engine) and Wine (an implementation of the Win32 API for Linux) and quickly found several bugs in all of them. Most errors are leaks, and involve custom allocators and deallocators not checked by current tools.

```
int coredump(struct proc *p) {
 ...
 name = proc_core_name(...); /* allocates a string */
 ...
 /* "name" is ALWAYS leaked after calling vnode_open */
 if ((error = vnode_open(name, ... )))
```

Figure 6: BUG. Function `coredump` in XNU always leaks a string allocated by `proc_core_name`.

Figure 6 illustrates an example bug found in the XNU. The function `coredump` is invoked within the kernel to process a core dump of a user process. The function `proc_core_name` is called to construct a freshly allocated string that indicates the location of the core dump file. This string is always leaked after the call to `vnode_open`, which leads to the kernel leaking a small amount of memory every time a process core dumps. This is a serious error, as a renegade process can cause the kernel to leak an arbitrary amount of memory and eventually cripple the OS (this bug has been fixed for the next release of Mac OS X). The bug was found by inferring that `proc_core_name` is an ro because it calls a commonly invoked allocator function that was also inferred to be an ro.

6 Conclusion

We presented a general method that combines factor graphs and static program analysis to infer specifications directly from programs. We believe the technique shows significant promise for inferring a wide range of specifications using probabilistic analysis. This includes applications in computer security, where many security exploits could be fixed by correctly identifying "tainted" input data (such as from a web form) that is exploitable by an attacker, or by inferring possible bounds for arrays to detect buffer overruns when conventional analysis fails. These and other problems represent promising and exciting future directions for this work.

References

[Engler *et al.*, 2000] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 2000*, October 2000.

[Flanagan *et al.*, 2002] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, pages 234–245. ACM Press, 2002.

[Heine and Lam, 2003] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI 2003*, 2003.

[Hinton, 2000] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. Technical Report 2000-004, Gatsby Computational Neuroscience Unit, University College London, 2000.

[Kremenek *et al.*, 2006] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *"Proceedings of the Seventh Symposium on Operating Systems Design and Implemetation"*, 2006.

[RTI, 2002] RTI. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institution of Standards and Technology (NIST), United States Department of Commerce, May 2002.

[Xie and Aiken, 2005] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *FSE 2005*, New York, NY, USA, 2005. ACM Press.

[Yedidia *et al.*, 2003] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. In *Exploring Artificial Intelligence in the New Millennium*. Morgan Kaufmann Publishers Inc., 2003.

[Zheng *et al.*, 2003] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Seventeenth Annual Conference on Neural Information Processing Systems*, 2003.