frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 28 May 2021
doi: 10.3389/fams.2021.668082

Topology-Inspired Method Recovers Obfuscated Term Information From Induced Software Call-Stacks

Kelly Maggs[1] *and Vanessa Robins*[2]*

[1]Mathematical Sciences Institute, Australian National University, Canberra, ACT, Australia, [2]Research School of Physics, Australian National University, Canberra, ACT, Australia

Fuzzing is a systematic large-scale search for software vulnerabilities achieved by feeding a sequence of randomly mutated input files to the program of interest with the goal being to induce a crash. The information about inputs, software execution traces, and induced call stacks (crashes) can be used to pinpoint and fix errors in the code or exploited as a means to damage an adversary's computer software. In black box fuzzing, the primary unit of information is the call stack: a list of nested function calls and line numbers that report what the code was executing at the time it crashed. The source code is not always available in practice, and in some situations even the function names are deliberately obfuscated (i.e., removed or given generic names). We define a topological object called the call-stack topology to capture the relationships between module names, function names and line numbers in a set of call stacks obtained via black-box fuzzing. In a proof-of-concept study, we show that structural properties of this object in combination with two elementary heuristics allow us to build a logistic regression model to predict the locations of distinct function names over a set of call stacks. We show that this model can extract function name locations with around 80% precision in data obtained from fuzzing studies of various linux programs. This has the potential to benefit software vulnerability experts by increasing their ability to read and compare call stacks more efficiently.

Keywords: fuzzing, crash-triage, software vulnerability research, call-stack analysis, topology, TDA, specialization pre-order

OPEN ACCESS

Edited by:
Frédéric Chazal,
Inria Saclay-Île-de-France Research Center, France

Reviewed by:
André Lieutier,
Dassault Systèmes, France
Vincent Rouvreau,
Inria Saclay-Île-de-France Research Center, France

****Correspondence:***
Vanessa Robins
vanessa.robins@anu.edu.au

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *15 February 2021*
Accepted: *03 May 2021*
Published: *28 May 2021*

Citation:
Maggs K and Robins V (2021) Topology-Inspired Method Recovers Obfuscated Term Information From Induced Software Call-Stacks. Front. Appl. Math. Stat. 7:668082. doi: 10.3389/fams.2021.668082

1 INTRODUCTION

A black-box fuzzing campaign is one conducted without explicit knowledge of the source code or its intermediate representations. Generally, methods in this area require a brute-force generation of inputs. This can lead to masses of crashes where many are duplicates of one another. For practitioners, untangling the output of a black-box fuzzing campaign is a time-consuming task. The goal of this article is to investigate methods that alleviate the difficulty of comprehending such results.

1.1 Call Stacks

When a program crashes, the slew of error text it returns to the user is referred to as the call-stack (Example in **Figure 1**). The call-stack is a record of the nested functions traced out by the program in its final moments and is one of the few pieces of information available to us when analyzing black-box fuzzing. The lines in the call-stack are called frames, and while contingent on the operating system's debugging syntax, decompose roughly into three columns: 1) the module (or filename), 2)

Frame	Module	Function	Line Num.
0	parser.c	xmlParseCDSect	9750
1	parser.c	xmlParseContent	9806
2	parser.c	xmlParseElement	9995
3	parser.c	xmlParseContent	9822
4	parser.c	xmlParseElement	9995
5	parser.c	xmlParseDocument	10665
6	parser.c	xmlDoRead	15062
7	parser.c	xmlReadFile	15122
8	xmllimit.c	parseAnPrintFile	2382

Frame	Module	Function	Line Num.
0	parser.c	????	9750
1	parser.c	????	9806
2	parser.c	????	9995
3	parser.c	????	9822
4	parser.c	????	9995
5	parser.c	????	10665
6	parser.c	????	15062
7	parser.c	????	15122
8	xmllimit.c	????	2382

FIGURE 1 | A simplified call-stack in our data-set with and without function terms.

the function and 3) the line number. We will refer to the set of all constituent modules, functions and line numbers in a set of call-stacks $\mathcal{C}$ as the *terms* in $\mathcal{C}$.

Further complicating matters is that—depending on whether the source code is available—call-stacks may have partial information excluded. In particular, when fuzzing programs without possession of source code or full knowledge of terms, the partially obscured call-stack may be the only source of information available.

1.2 Goals

The ANU researchers [1] provided to us a data set of call stacks generated by fuzzing several Linux programs with the afl fuzzing algorithm (See [2]). They were interested in answering two key research questions:

1. Clustering and Deduplication: determining the extent to which there are discernible clusters in the set of call-stacks.
2. Term Removal: quantifying how much information about function terms can be recovered given they are obscured (for example, as in **Figure 1**).

While the first question has been studied in a number of contexts [3, 4], to the best of our knowledge no attempt has been made at the second. In this paper, we show that once the data has been suitably whitelisted, the set of crashes contain a high number of exact duplicate call-stacks. This observation highlights a fundamental lack of diversity in the data generated by fuzzing, and alone is enough to answer the first question to a large extent.

To address the question of function term removal, we introduce a model of call-stack information using finite topological spaces, posets and the theory of [5]. This not only helps to quantify the significance of removing function terms, but is a useful object to capture the dependencies between terms in the set of call-stacks.

2 DATA-SET OVERVIEW

Six common Linux programs were fuzzed using the program afl. Key aspects of the program: the binary name, file extension, and version are presented in **Table 1**. Call-stacks were generated within the framework of the GDB debugger using the AddressSanitizer (ASAN) [6] tool.

Upon recommendation by the ANU cyber-security researchers, we performed several pre-processing whitelisting steps to each call-stack text file. Firstly, frames appearing up to and including the ASAN error frame were considered superfluous and hence deleted. In crashes that did not call the ASAN module, frames up to assert_fail were deleted. For every file, the two final generic end-of-file frames were deleted. Finally, we extracted three salient features from each frame: the module, the function and intra-module line number, and discarded the other information in the call-stack file.

Unlike the afl protocol–where crashes are de-duplicated based on a hashing scheme–we labeled two call-stacks to be deplicates whenever their text files were identical after the pre-processing described above. A striking result of frequency analysis is that there are dramatically fewer distinct crashes relative to total crashes (see **Figure 1**). Further, the frequency of distinct crashes is unevenly distributed. Across programs, the call-stack data displayed largely the same pattern: most of the weight was distributed among a few crashes, with the rest rarely occurring (See **Figure 2**).

3 TOPOLOGICAL MODEL

In this section, we propose a model to frame the complex dependency relationships between the terms $\mathcal{T}$ appearing across a set of call-stacks $\mathcal{C}$. Our model is inspired by the work of [5] on finite topological spaces, where pre-orders, equivalence classes and posets capture certain topological interactions between points. Our applications will use primarily the poset representation of the data, but we have included the topological perspective which motivated the original idea with the hope that future work may be able to further incorpate the topological characteristics of the model.

Recall that in any topological space X, the open neighborhood $\mathcal{N}(x)$ of a point $x \in X$ is the set of open sets containing x. A rough intuition of point-set topologies over finite sets is that elements

TABLE 1 | The number of crashes generated by fuzzing each program, and the number of unique crashes after whitelisting and removed exact duplicates. Within the set of call-stacks, some files were either blank or unable to be opened. We discarded such files, as appears in the second column from the right.

Binary	Extension	Version	Call stacks	Discarded (per 1,000)	Distinct
SoX	mp3	14.4.2	40,017	1 (0.02)	12
Librsvg	Svg	2.40.20	6,276	94 (15)	68
Libtiff	Tiff	4.0.9	5,486	2 (0.36)	9
Freetype	Ttf	2.5.3	17,034	1 (0.06)	51
SoX	Wav	14.4.2	30,856	1 (0.03)	11
Libxml2	Xml	2.9.0	240,821	7,467 (31)	3,175

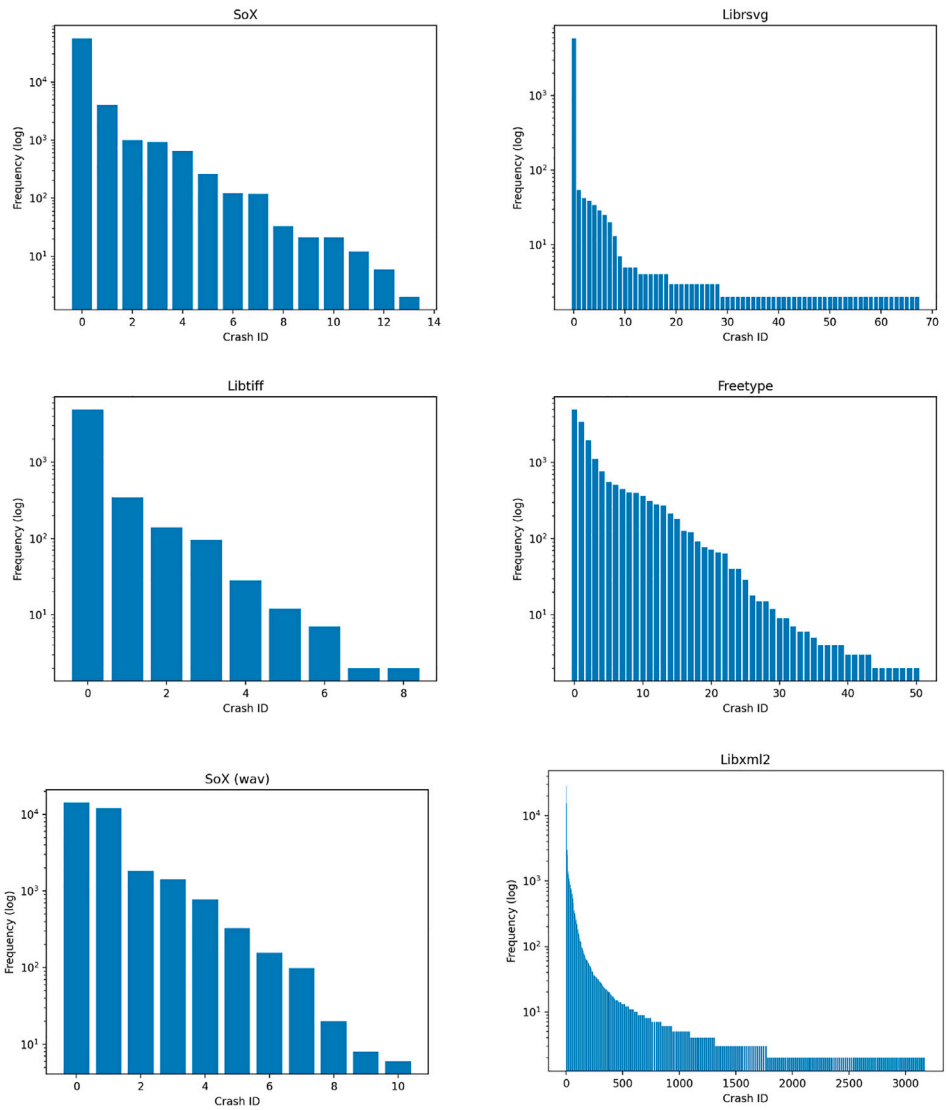

FIGURE 2 | Frequency (logarithmic scale) distribution of crashes in each binary, where the crash ID is with respect to distinct call-stacks.

are considered close when they have similar open neighbourhoods. Our goal is to create a topology over the set of terms $\mathcal{T}$ where those that occur in a similar set of call-stacks are close.

3.1 Call-Stack Topology

Given a set of call-stacks $\mathcal{C}$ comprised of terms $\mathcal{T}$, we define the call-stack topology $\mathcal{T}(\mathcal{C})$ on $\mathcal{T}$ to be that generated by treating each call-stack $c \in \mathcal{C}$ as an open set of terms. The complete

collection of open sets in $\mathcal{T}(\mathcal{C})$ is then formed by taking all possible intersections and unions of call-stacks from $\mathcal{C}$.

Unlike many topologies we are familiar with, e.g., topologies generated by open balls in a metric space, the call-stack topology is seldom Hausdorff. In our context, the looser criterion of a T_0 space is a more useful notion of point separation. Recall that a T_0 space $(X, \mathcal{N})$ is one where points may be distinguished by their open neighbourhoods; explicitly, for each pair of points $x, y \in X$ there exists either an open set in $\mathcal{N}$ containing x without y or y without x.

Since distinct terms can appear in the same subset of call-stacks, $\mathcal{T}(\mathcal{C})$ is not even T_0. However, we can transform $\mathcal{T}(\mathcal{C})$ into a T_0 space by taking the Kolmogorov quotient (see [7]). The Kolmogorov quotient $\tilde{X}$ is obtained from $(X, \mathcal{N})$ by the equivalence relation $x \sim y$ whenever they have the same open neighborhood $\mathcal{N}(x) = \mathcal{N}(y)$. It is known that $\tilde{X}$ and $(X, \mathcal{N})$ have the same homotopy type. By taking the Kolmogorov quotient of the call-stack topology, one reduces the object of study from a potentially large set of terms $\mathcal{T}$ into a more manageable set of equivalence classes of terms $\tilde{\mathcal{T}}$.

The following simple lemma shows that one may characterize equivalence classes in the Kolmogorov quotient of the call-stack topology by examining the set of call-stacks directly rather than the topology. For $t \in \mathcal{T}$, we refer to the set of call-stacks in $\mathcal{C}$ which contain t as the call-stack neighborhood, using the notation $\mathcal{C}(t)$.

Lemma 1. For a set of call-stacks $\mathcal{C}$ comprised of terms $\mathcal{T}$, two terms $t_1 \sim t_2$ are equivalent if and only if $\mathcal{C}(t_1) = \mathcal{C}(t_2)$.

Proof. The definition of the equivalence relation is $t_1 \sim t_2$ whenever $\mathcal{N}(x) = \mathcal{N}(y)$ in the call-stack topology. Hence, we need to show that open neighbourhoods $\mathcal{N}(x) = \mathcal{N}(y)$ are equal if and only if call-stack neighbourhoods $\mathcal{C}(t_1) = \mathcal{C}(t_2)$ are equal.

Suppose that $\mathcal{C}(t_1) \neq \mathcal{C}(t_2)$. Without loss of generality, suppose there exists $c \in \mathcal{C}$ such that $t_1 \in c$ and $t_2 \notin c$. By the definition of call-stack topology, c is open and hence a member of $\mathcal{N}(t_1)$. Since $t_2 \notin c$, it follows that $\mathcal{N}(t_1) \neq \mathcal{N}(t_2)$, proving one side of the statement.

Conversely, suppose that $\mathcal{C}(t_1) = \mathcal{C}(t_2)$, and further suppose that $U \in \mathcal{N}(t_1)$. All open sets in the topology generated by a set $\mathcal{C}$ may be expressed in the form

$$U = \bigcup_j \bigcap_i \mathcal{C}_{i,j} \tag{1}$$

where each $\mathcal{C}_{i,j} \in \mathcal{C}$ is a generating set. The assumption $U \in \mathcal{N}(t_1)$ implies that $t_1 \in U$ and further that there exists j such that $t_1 \in \mathcal{C}_{i,j}$ for all i. Since we have assumed that $\mathcal{C}(t_1) = \mathcal{C}(t_2)$, $t_1 \in \cap_i \mathcal{C}_{i,j}$ implies that $t_2 \in \cap_i \mathcal{C}_{i,j} \subseteq U$ as well. This implies that $\mathcal{N}(t_1) \subseteq \mathcal{N}(t_2)$. By the same argument $\mathcal{N}(t_2) \subseteq \mathcal{N}(t_1)$, thus $\mathcal{N}(t_1) = \mathcal{N}(t_2)$, and finishing the proof.

According to the above lemma, equivalence classes in the Kolmogorov quotient of the call-stack topology consist of terms that occur in the same set of call-stacks. The intuition is that by taking the Kolmogorov quotient, we only consider terms up to the information of which call-stacks they appear in. The composition of equivalence classes in such a quotient will be a key feature for analysis in our application.

In theory, calculating such equivalence classes requires knowledge of open neighbourhoods and, ergo, the entire gamut of open sets in the call-stack topology. Aside from providing useful intuition, the above lemma also ensures that we can avoid this computationally expensive task, attaining equivalence classes indirectly by comparing the call-stack neighbourhoods of pairs of terms.

Example 1. In **Figure 3**, we depict a set of three call-stacks. In the center of the Figure, the three circles each represent a generating set for the call-stack topology $\mathcal{T}(\mathcal{C})$ over the constituent terms $\mathcal{T}$ of $\mathcal{C}$. The coloring of the terms represents their partition into equivalence classes under the Kolmogorov quotient operation. Following Lemma 1, equivalence classes consist of terms sharing identical call-stack neighbourhoods. This example also highlights that both the ordering of terms in the call-stack and the frequency of each term within it are both disregarded by the model.

3.2 Call-Stack Partial Order

In this section we equip the set of call-stack terms with the additional structures of a pre-order and partial order. Our approach in later sections is to use this structure to examine relations between terms in different equivalence classes. For any topological space, one may use the structure of the open sets to define a pre-order over its points called the specialization pre-order. This may be defined in the following equivalent statements.

Definition 1. *For a topological space X, the* specialization pre-order (X,<=) over X is given by either

$$x \leq y \text{ whenever } \mathcal{N}(y) \subseteq \mathcal{N}(x)$$

or equivalently

$$x \leq y \text{ whenever } y \in \bigcap_{U \in \mathcal{N}(x)} U$$

The specialization pre-order forms a partial order over X precisely when X is a T_0 space, with the T_0 condition ensuring that the order relation satisfies the anti-symmetry condition: $x \leq y$ and $y \leq x$ implies $x = y$.

Definition 2. The call-stack pre-order on a set of call-stacks $\mathcal{T} \leq (\mathcal{C})$ is the specialization pre-order over the call-stack topology $\mathcal{T}(\mathcal{C})$.

definition 3. The call-stack poset on a set of call-stacks $\tilde{\mathcal{T}} \leq (\mathcal{C})$ is the specialization pre-order over the Kolmogorov quotient $\tilde{T}(\mathcal{C})$ of the call-stack topology.

Unlike the call-stack topology $\mathcal{T}(\mathcal{C})$ in general, the Kolmogorov quotient $\tilde{T}(\mathcal{C})$ of the call-stack topology is guaranteed to be T_0 space (see [7] for a full survey of Kolmogorov quotients). Note here that the call-stack poset is defined over equivalence classes of terms within the call-stacks, rather than the individual terms themselves. In moving to this construction, we reduce the space of information we are working with; order theoretic notions are considered between blocks of terms rather than individual ones.

As is the case for equivalence classes, the call-stack partial order can be computed without explicitly calculating the open sets in the call-stack topology.

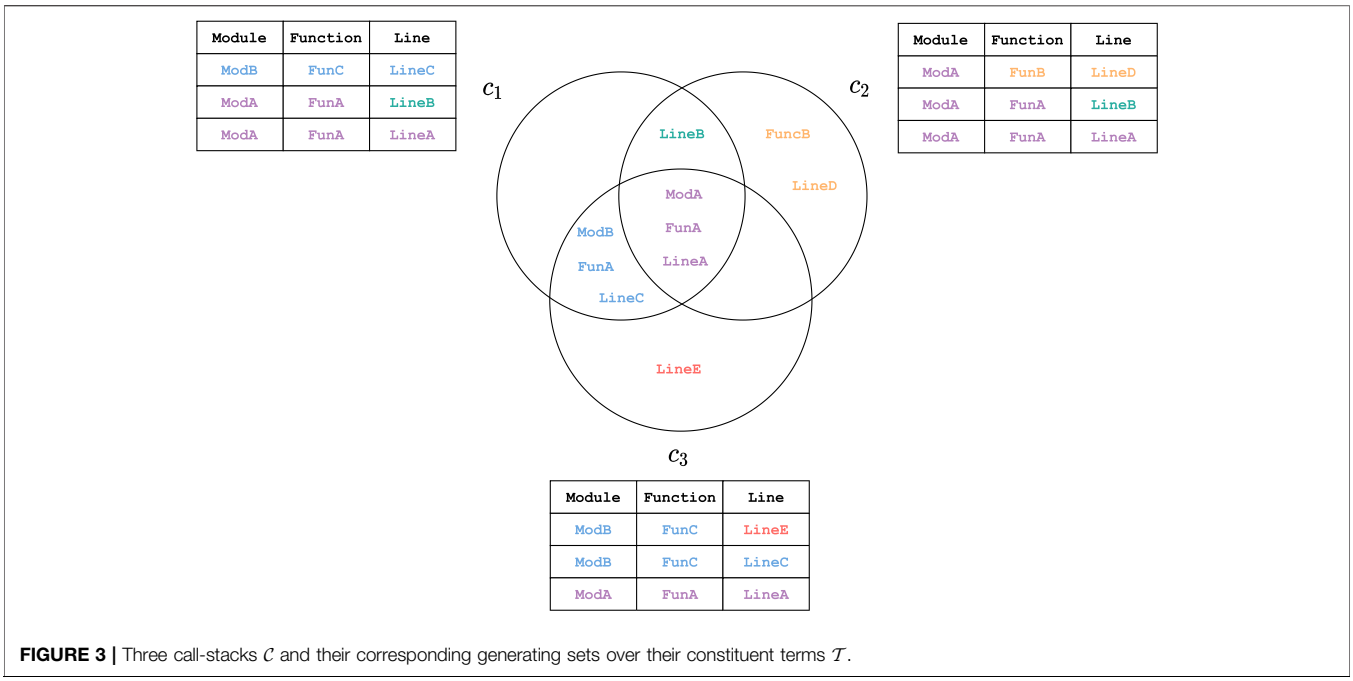

FIGURE 3 | Three call-stacks $\mathcal{C}$ and their corresponding generating sets over their constituent terms $\mathcal{T}$.

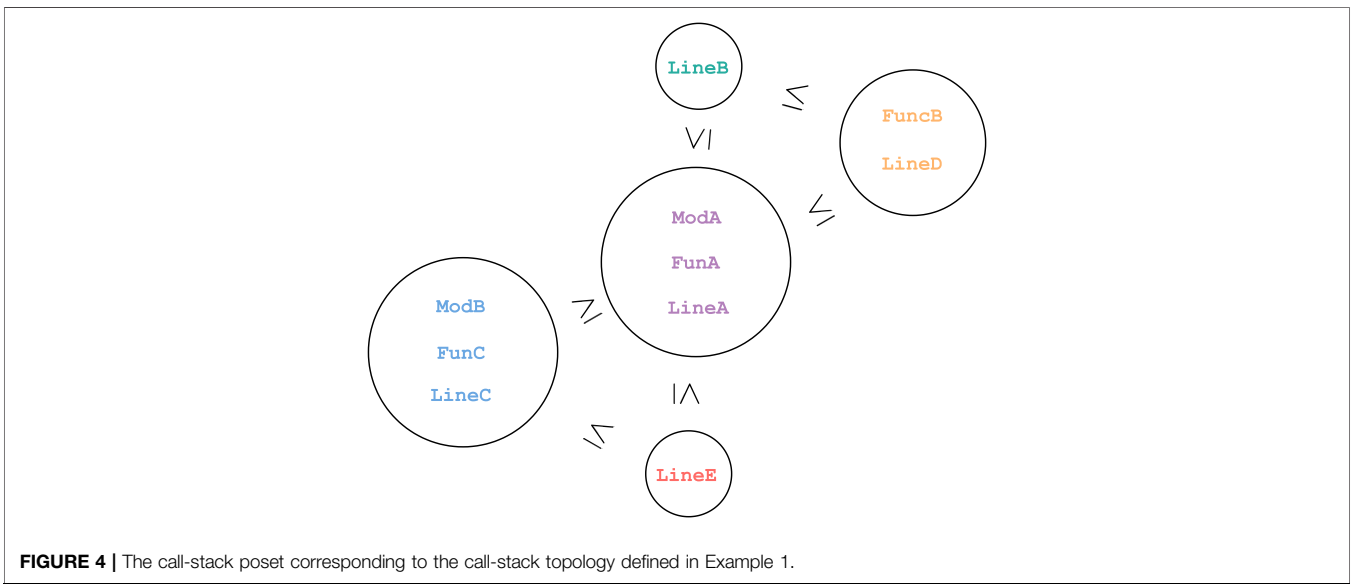

FIGURE 4 | The call-stack poset corresponding to the call-stack topology defined in Example 1.

LEMMA 2. *Two classes of terms* $[t_1], [t_2] \in \tilde{T}(\mathcal{C})$ *satisfy an order relation* $[t_1] \leq [t_2]$ *in the call-stack partial order if and only if* $\mathcal{C}(t_2) \subseteq \mathcal{C}(t_1)$.

PROOF. Suppose first that $\mathcal{C}(t_2) \subseteq \mathcal{C}(t_1)$. Let $U \in \mathcal{N}(t_2)$ be an open neighborhood of t_2 in the call-stack topology. As in Equation 1, U may be expressed in the form

$$U = \bigcup_j \bigcap_i \mathcal{C}_{i,j}$$

where there exists j such that $t_2 \in \cap_i \mathcal{C}_{i,j}$. Since $\mathcal{C}(t_2) \subseteq \mathcal{C}(t_1)$, it follows that $t_1 \in \cap_i \mathcal{C}_{i,j}$ also, implying that $U \in \mathcal{N}(t_1)$ and $\mathcal{N}(t_2) \subseteq \mathcal{N}(t_1)$. Thus, $[t_1] \leq [t_2]$ as required.

In the other direction, suppose that C_2 is not a subset of C_1. Then there exists $c \in \mathcal{C}(t_2)$ such that $c \notin \mathcal{C}(t_1)$. Since c is open in the call-stack topology, $c \in \mathcal{N}(t_2)$ and $c \notin \mathcal{N}(t_2)$, implying that $\mathcal{N}(t_2) \not\subseteq \mathrm{N}(t_1)$ and thus $[t_1] \leq [t_2]$.

The above lemma suggests how one should interpret the call-stack partial order: two sets of terms $[t_1], [t_2] \in [t_1], [t_2] \in \tilde{T}(\mathcal{C})$ satisfy an order relation $[t_1] \leq [t_2]$ when every call-stack containing the $[t_2]$ terms also contains the $[t_1]$ terms. In this sense, witnessing the terms in $[t_2]$ depends on witnessing the terms in $[t_2]$ across the call-stacks in $\mathcal{C}$.

Example 2. In **Figure 4**, we depict the call-stack poset corresponding to the example call-stack topology provided in Example 1. Each circle contains an equivalence class of terms that have identical call-stack neighbourhoods. Lemma 2 tells us that an order relation $[t_1] \leq [t_2]$ between classes occurs when $\mathcal{C}(t_2) \subseteq \mathcal{C}(t_1)$; namely, when all call-stacks containing t_2 also contain t_1.

3.3 Function Term Obfuscation

One of the key research questions is how much information can be extracted from the call-stack when terms are removed. Let $\mathcal{C}$ be a collection of call-stacks, tokenized into a set of terms $\mathcal{T}$. To consider the effect on the model of removing a single term $t \in \mathcal{T}$, let $\mathcal{T}' \triangleq \mathcal{T} \backslash \{t\}$ and

$$\mathcal{C}' \triangleq \{c \backslash (c \cap \{t\}) | c \in \mathcal{C}\}$$

be the set of call stacks with the term t removed. Note here that each $c \in \mathcal{C}$ is a set of terms $c \subseteq \mathcal{T}$, so the set notation $c \backslash (c \cap \{t\})$ makes sense. The following lemma describes the effect on the call-stack poset when t is removed.

Lemma 3. Suppose $t_1, t_2 \in \mathcal{T}$. Then

1. $[t_1] \neq [t_2]$ *in* $\tilde{\mathfrak{T}}(\mathfrak{C})$ if and only if $[t_1] \neq [t_2]$ *in* $\tilde{\mathcal{T}}'(\mathcal{C}')$.
2. $[t_1] \leq [t_2]$ *in* $\tilde{\mathcal{T}} \leq (\mathcal{C})$ if and only if $[t_1] \leq [t_2]$ *in* $\tilde{\mathcal{T}}'(\mathcal{C}')$.

Proof. For (1), $[t_1] \neq [t_2]$ in $\tilde{\mathfrak{T}}(\mathcal{C})$ if and only if there exists $c \in \mathcal{C}$ such that either $t_1 \in c$ and $t_2 \notin c$ or $t_1 \notin c$ and $t_2 \in c$. This occurs if and only if there exists $\mathcal{C}' \in \mathcal{C}$ such that either $t_1 \in \mathcal{C}'$ and $t_2 \notin \mathcal{C}'$ or $t_1 \notin \mathcal{C}'$ and $t_2 \in \mathcal{C}'$, which is equivalent to $[t_1] \neq [t_2]$ in $\tilde{\mathcal{T}}' \leq (\mathcal{C})$. For (2), $[t_1] \leq [t_2]$ in $\tilde{\mathcal{T}} \leq (\mathcal{C}) \Leftrightarrow \mathcal{C}(t_2) \subseteq \mathcal{C}(t_1) \Leftrightarrow \mathcal{C}'(t_2) \subseteq \mathcal{C}'(t_1) \Leftrightarrow [t_1] \leq [t_2]$ in $\tilde{\mathcal{T}}' \leq (\mathcal{C}')$.

One can summarize the above result as the fact that $\tilde{\mathcal{T}}'(\mathcal{C}')$ is isomorphic to $\tilde{\mathcal{T}}(\mathcal{C})$ whenever the singleton $\{t\}$ is not an equivalence class. When it is an equivalence class, it is the only difference between the two call-stack posets $\tilde{\mathcal{T}} \leq (\mathcal{C})$ and $\tilde{\mathcal{T}}' \leq (\mathcal{C}')$. By inductively removing all of the function terms, $\mathcal{F} \subset \mathcal{T}$, and applying the lemma at each step, we attain the following corollary.

Corollary 1. Let $\widetilde{\mathfrak{T}''}(\mathcal{C}'')$ be the call-stack poset over $\tilde{\mathcal{T}}'' \triangleq \mathcal{T} \backslash \mathcal{F}$ generated by

$$\mathcal{C}'' \triangleq \{c \backslash (c \cap \mathcal{F}) | c \in \mathcal{C}\}$$

Then $\tilde{\mathcal{T}}'' \leq (\mathcal{C}'')$ is the sub-poset of $\tilde{\mathfrak{T}}(\mathcal{C})$ spanned by equivalence classes

$$\left\{ [x] \in \tilde{\mathfrak{T}}(\mathcal{C}) \middle| [x] \not\subseteq \mathcal{F} \right\}$$

In other words, whenever we remove function terms from the model, the structure of the call-stack poset is unchanged away from classes comprised solely of function terms. When a function term t shares an equivalence class with non-function terms, these may be used to recover its structural dependency information even when t is removed. The point of the above theorems is to motivate the idea that many attributes of the call-stack poset are retained in the case where some terms are missing.

4 FUNCTION TERM RECONSTRUCTION

The goal of this paper is to reconstruct information about function terms from call-stacks in which they are obscured. In this section, we present a small-scale experiment on our linux data set using features extracted from the call-stack topology model.

Accordingly, we must first define what we mean by 'function information.' When the function names are missing, it is not possible to recover them explicitly from the call-stack data. The next best data, and what we choose to focus on in this paper, is to recover the set of positions within the call-stacks that share a common function name. This notion is captured in the following definition.

Definition 4. For a term $t \in \mathcal{T}$ within a set of call stacks $\mathcal{C}$, define the frame trace $\mathsf{FT}_{\mathcal{C}}(t)$ of t to be the set of pairs

$$\mathsf{FT}_{\mathcal{C}}(t) \triangleq \{(c, n) | t \in c[n]\}$$

where $c[n]$ is the nth frame of call stack c.

If t appears in multiple frames $c[n]$ and $c[n']$ within the same crash $c \in \mathcal{C}$, then both (c, n) and (c, n') are elements of $\mathsf{FT}_{\mathcal{C}}(t)$. For any pair of terms of the same type in a set of call stacks, their frame traces must be disjoint. It is impossible to guess the explicit names of obscured function terms. However, if one can recover the frame traces of every function, then one can generate call stacks that are equivalent up to re-naming function terms.

By performing logistic regression over features extracted from the call-stack model, we will show that a surprising number of function frame traces can be recovered without any explicit knowledge of function names. This is particularly striking given that the user also knows nothing about the internal structure of the program. Additionally, we provide an algorithm for generating fake function names based on the guessed frame traces, making sets of call stacks more human-readable in the setting where function names are missing.

4.1 Preliminary Analysis of Call-Stack Equivalence Classes and Poset Structure

To motivate the use of our novel tools in the task of recovering function frame traces, we first present a basic analysis of the data through the lens of the call-stack topology and poset. In particular, we study the characteristics of equivalence classes—their size and the types of terms of which they comprise—as well as the order relations and dependencies they exhibit on one another.

4.1.1 Basic Statistics

Recall that the equivalence classes in the call-stack topology consist of terms that occur in the same set of call stacks. **Table 2** shows the extent of reduction from the number of terms to their equivalence classes under the quotient operation.

Our primary interest is to understand the effect of obscuring function terms. Corollary 1 states that removing the function terms only alters the model's structure at equivalence classes consisting of function terms alone. Accordingly, we say a function term f is *retained* under the quotient operation when it is equivalent to a non-function term t. Notably, in the case $f \in \mathcal{F}$ is retained, there exists a term $t \in \mathcal{T} \backslash \mathcal{F}$ with call-stack set equivalent to f.

Table 2 shows that, on average, 86% of function terms are retained. Extensive term equivalences in the call-stack topology mean that a dramatic reduction in the available terms has little

TABLE 2 | Call-stack model and term statistics for each linux program.

	SoX (m)	Librsvg	Libtiff	Freetype	SoX (w)	Libxml	Mean
Modules	15	15	8	23	14	17	
Functions	36	92	17	48	34	151	
Line num	43	99	21	81	40	361	
Total terms	94	206	46	152	88	529	
Classes	33	113	14	66	27	343	
Reduction %	65%	45%	70%	56%	69%	35%	57%
Order relations	108	1,024	22	352	89	2,220	
F-loss	4	32	0	6	2	29	
F-retention %	89%	65%	100%	88%	94%	81%	86%

Frame	Module	Function	Line Num.
0	effects_i_dsp.c	lsx_make_lpf	367
1	effects_i_dsp.c	lsx_design_lpf	405
2	e.c	dft_stage_init	225
3	e.c	rate_init	443
4	rate.c	start	632
5	effects.c	sox_add_effect	157
6	sox.c	add_effect	708
7	sox.c	add_effects	1073
8	sox.c	process	1759

Frame	Module	Function	Line Num.
0	parser.c	xmlParseCDSect	9750
1	parser.c	xmlParseContent	9806
2	parser.c	xmlParseElement	9995
3	parser.c	xmlParseContent	9822
4	parser.c	xmlParseElement	9995
5	parser.c	xmlParseDocument	10665
6	parser.c	xmlDoRead	15062
7	parser.c	xmlReadFile	15122
8	xmllimit.c	parseAnPrintFile	2382

FIGURE 5 | Example call stacks from the Linux data exhibiting the behavior described by the two patterns. The left and right examples are taken from two separate libraries, where each of the seven colors represents an equivalence class in the call-stack topology; that is, a set of terms which have identical call-stack neighbourhoods.

effect on the call-stack poset structure. The main takeaway from this analysis is that function terms rarely occur in an equivalence class on their own.

4.1.2 Patterns Relating Line Numbers and Function Terms

When two terms are in the same equivalence class, they occur in the same set of call-stacks. However, our topological model encodes none of the information about which frame they occur in. Our toy example (Example 1) suggested that line numbers and functions in the same equivalence class tend to occupy similar frames in the call-stack. In a thorough examination of the data, we observed two patterns, demonstrating each with the example call stacks in **Figure 5**.

- Pattern 1: When multiple line numbers belong to an equivalence class, they are usually paired with functions in the same frame except for the line number in the bottom frame. The lowest line number appears to act as a switch point between blocks of terms, instigating a run of function calls that are either seen in only one call-stack or always together. In **Figure 5**, this occurs in the brown and green equivalence classes of the example on the left.
- Pattern 2: When a single line number is in an equivalence class with a function, it is likely paired with a function one frame above. In **Figure 5**, this occurs in the purple and orange boxes of the left call-stack, and the purple, orange and green boxes of the right.

It is important to state that neither pattern reflects an underlying mathematical truth. Rather, they seem to be a symptom of programming convention. Namely, as source code tends to decompose into many different simple functions nested within one another, runs of frames in the call-stack tend to cycle through distinct function names. Further, these patterns only apply in the case that a line number occurs in the same set of crashes as a function, in which case we assume that they describe how the frames of a function and line number are related.

4.2 Method

Our method for frame trace recovery is centered around leveraging structure of the call-stack topology and poset. To do so, we generate the call-stack equivalence classes $\widetilde{T''}(\mathcal{C})$ and poset $\widetilde{\mathfrak{T}''} \leq (\mathfrak{C})$ from the incomplete data $\mathcal{T}''$, i.e., the set of terms with function names omitted. The intuition of Corollary 1 — as well as the empirical observations of **Table 2** — suggest that such objects should be relatively similar to their counterparts $\tilde{\mathcal{T}}(\mathcal{C}), \tilde{\mathfrak{T}} \leq (\mathfrak{C})$ generated from the full data that we aim to partially reconstruct.

Once such objects are constructed, our approach consists of the following two steps.

1. Classifying Equivalence Classes: Within the incomplete data model, the terms of an equivalence class $[t] \in \widetilde{T''}(\mathcal{C})$ consist only of line numbers and module names. However, in the complete data model, there may exist function terms that are also in the corresponding class. The first step of

our method is to estimate the likelihood that an equivalence class $[t] \in \widetilde{T''}(\mathcal{C})$ in the incomplete data corresponds to an equivalence class containing a function term in the full data $\tilde{T}(\mathcal{C})$.

2. Pairing Frame Traces: The second step of our method is to apply our two observations above to obtain a heuristic for predicting frame trace locations of function terms. This is done by selecting line numbers within a given equivalence class whose frame traces are likely to be paired with a function term frame trace in the complete data. The frame traces of these line numbers serve as our set of predictions for function frame traces and enable us to partially reconstruct the data set.

4.2.1 Classifying Equivalence Classes Within Libxml

The first step in our method of frame trace recovery is learning to detect when an equivalence class contains a function term. Before tackling the task of classifying equivalence classes in the incomplete data, we restrict our focus to a small study of libxml, which offers the largest base of terms and equivalence classes from which to garner information. We outline our method here to examine the relationship between the structure of an equivalence class within the libxml call-stack poset and the types of terms that it contains.

Consider the following three binary classification problems over the equivalence classes $\tilde{\mathcal{T}}(\mathcal{C})$ in the call-stack topology.

1. Modules: each equivalence class $[t] \in \tilde{T}(\mathcal{C})$ is labeled 1 if there exists a module $m \in \mathcal{M}_{\cap}[t]$ and 0 otherwise.
2. Functions: each equivalence class $[t] \in \tilde{\mathcal{T}}(\mathcal{C})$ is labeled 1 if there exists a function $f \in \mathcal{F}_{\cap}[t]$ and 0 otherwise.
3. Line Numbers: each equivalence class $[t] \in \tilde{T}(\mathcal{C})$ is labeled 1 if there exists a line number $l \in \mathcal{L}_{\cap}[t]$ and 0 otherwise.

We address each of the above by performing a simple logistic regression based on four features in the call-stack model. For an equivalence class $[t] \in \tilde{T}(\mathcal{C})$, these are as follows.

1. The size $|\{t' \in [t]\}|$ of an equivalence class.
2. The frequency (number of call-stacks) of the class $|\{\mathcal{C} \in \mathcal{C} | [t] \subseteq \mathcal{C}\}|$.
3. The weighted in-degree

$$\sum_{[t']} \phi([t'] \leq [t])$$

of the class within the order graph of the call-graph poset $\tilde{\mathfrak{C}} \leq (\mathfrak{T})$.

4. The *weighted out-degree*

$$\sum_{[t']} \phi([t] \leq [t'])$$

of the class within the order graph of the call-graph poset $\tilde{\mathfrak{C}} \leq ([\mathfrak{T}])$.

The names given to features 3 and 4 reference the fact that a poset p can be represented as a graph whose nodes are elements of p and edges are order relations $p \leq q$. The in- and out-degree of $[t]$ are the number of equivalence classes that $[t]$ depends on and that depend on $[t]$ respectively. To incorporate the magnitude of such dependencies, the weight of an order relation is determined by the function

$$\phi([t] \leq [t']) = \frac{|\{\mathcal{C} \in \mathcal{C} | [t'] \subseteq \mathcal{C}\}|}{|\{\mathcal{C} \in \mathcal{C} | [t] \subseteq \mathcal{C}\}|}$$

Lastly, for normalization each of the four variables is scaled by minimum and maximum to lie within $[0, 1]$, making the logistic regression weights comparable across variables.

Since the classification labels are unbalanced, the classes were re-weighted according to the to sci-kit learn class re-weighting scheme. To prevent over-fitting, the data was randomly split into an 80% training set and 20% testing set. To measure results, we use the F1 score and Area Under (precision-recall) Curve, which is suggested to be the most sensible measurements when predicting heavily weighted classes in binary classification (See [8]).

As a baseline to compare the statistical significance of our method, we propose the following binary classification null-model. Firstly, we empirically derive three probabilities from the ratio of the number of equivalence classes containing each term over the total number of equivalence classes. For each type of term, the null-model randomly guesses whether each class contains that particular term type with the empirically derived probability

4.2.2 Classifying Incomplete-Data Equivalence Classes

Once we have attained logistic regression weights for the libxml data, we then apply them to other programs. An important point of this stage is that, unlike the libxml program experiment, we withhold the full-data with function names as a validation set. This means that the call-stack topology and poset are generated for each program from the call-stacks $\mathcal{C}''$ with function terms obscured $\mathcal{T}''$.

From each of these objects, we extract the same four features as above, normalizing in the same way to ensure that the learned libxml weights scale appropriately. The goal of this stage is to predict whether an equivalence class in the incomplete data is likely to contain a function in the full-data, thus predicting a set of call-stacks which share a common missing function term.

4.2.3 Pairing Line Numbers With Function Frame Traces

The outline of our approach to predicting function frame traces is to 1) guess when a line number in the incomplete-data model was likely to have been in an equivalence class with a function name and 2) generating predicted frame traces for functions from line number frame traces using our two heuristics. Algorithm LABEL: predict_FTs ties these two steps together, taking in the set of obscured call-stacks $\mathcal{T}''(\mathcal{C}'')$ and their frame traces $\mathsf{FT}_{\mathcal{C}''}$ then returning a set of predicted frame traces. The value p is a cut-off

TABLE 3 | Proportion of equivalence classes containing each term type, and logistic regression F1 score and AUC improvements on the null-model for the libxml call-stacks.

	Module	Function	Line number
% Equivalence Classes	4.76%	65.01%	88.06%
F1	0.40	0.89	0.92
Null model	0.00	0.40	0.90
AUC	0.20	0.94	0.97
Null model	0.04	0.43	0.93

likelihood for using logistic regression weights to decide when a pattern should be applied to predict a frame trace.
Algorithm 1PredictFTs$(\tilde{\mathcal{T}}''(\mathcal{C}''), \mathsf{FT}_{\mathcal{C}''}, p)$.

$\mathcal{CT}$
  predicted_fts = []
  $[t] \in \tilde{\mathcal{T}}''(\mathcal{C}'')$
    $\mathbb{P}(\exists f \in [t]) \geq p$
      $\triangleright$ $|[t] \cap \mathcal{L}| > 1$
        $\triangleright$ lines = $([t] \cap \mathcal{L})$.drop_bottom_linenum()
        $l \in$ lines
       predicted_fts = predicted_fts + $[\mathsf{FT}_{\mathcal{C}''}(l)$ for $l \in$ lines$]$
      $|[t] \cap \mathcal{L}| = 1$
        $\triangleright$ $l = [t] \cap \mathcal{L}$
     predicted_fts.append$((c, min(n-1, 0)) | (c, n) \in \mathsf{FT}_{\mathcal{C}''}(l))$
predicted_fts

In our algorithm, the logistic regression weights in Line 3 learned over libxml serve as the basis for detecting whether there exists a function in each line number's equivalence class. Using only the libxml weights ensures that when we predict function frame traces, we require no information about function names in other programs beforehand.

The logistic regression is learned over the full-term model of libxml then performed over the call-stack topology models generated without terms in other programs. There are two significant obstacles that the model must overcome to be successful. Firstly, the model must exhibit *transference* if the regression weights from libxml are to work for other programs. Secondly, the model must be *robust* to term removal given that it classifies over a model without function terms. On the second point, Corollary 1 states that the call-stack model retains much of its structure when function terms are removed, which suggests that the logistic regression weights have a chance of still being applicable.

The role of the logistic regression model is primarily to act as a gate-keeper, probabilistically determining when a given line number is *not* in the same class as any function. This prevents the model from over-predicting instances where a function's frame trace should be paired to that of a line number. The method drop bottom lineum in Line 5 removes the line number with the lowest average frame trace from the set, which is necessary to apply pattern 1.

4.3 Results

4.3.1 Learning Libxml Logistic Regression Weights

In **Table 3** we present the results of our binary classification experiment within the libxml data described in Subsection 4.2.1. The results show that the inclusion of call-stack topology features significantly improves the quality of prediction across terms when compared with the null model. To quantify the effect of each feature in classification, we plot the logistic regression weights in **Figure 6**. In all cases, the frequency of a term has little effect on classification. For individual term types, there are several observations about the model variables that detect its presence in an equivalence class.

- Modules are likely to be in smaller equivalence classes with lower weighted in-degree and higher weighted out-degree. This means more terms depend on them than they depend on.
- Functions are likely to be in large equivalence classes, with high out-degree. This means many terms are likely to depend on them.
- Line Numbers are likely to be in larger equivalence classes, with a low weighted out-degree. This means terms are unlikely to depend on line numbers.

Each observation agrees with the structure of library dependencies, where line numbers depend on functions, and functions depend on modules. The logistic regression model is notably adept at detecting the presence of function terms within a given equivalence class.

4.3.2 Frame Trace Recovery

The PREDICTFTS algorithm is run over each Linux program. Since the function term information in libxml was used to generate the logistic regression weights, we exclude it from the analysis. To measure the results, we compare the set of predicted function frame traces generated by the algorithm against the set of actual function frame traces in each set of call stacks.

Table 4 contains the results of each experiment with three different cut-off probabilities 0.4, 0.5 and 0.6. Despite the heavy reliance on fairly naïve heuristics, our model has a reasonable mean precision of above 0.75 is each case. Notably, both precision and recall of frame traces are relatively stable across each program. This suggests that the libxml logistic regression weights and the heuristics both exhibit some degree of transference across programs.

In **Figure 7**, we analyze the effect of the cut-off probability parameter in detail. When this parameter is high, the algorithm requires a large degree of confidence that an equivalence class contains a function term before predicting a frame trace. This is reflected in an increasing precision and decreasing recall as the cut-off probability increases.

The cut-off probability parameter indirectly allows the user to dictate the importance of precision at the expense of recall. Given that the purpose of our experiment is to reliably reconstruct what function names we can, the importance of precision outweighs that of recall. Indeed, there exist function names in the data that could not possibly be recalled from the module and line number information alone. For example, large swathes of function names are hidden behind the repeated line number 0 in the librsvg data (**Figure 8**), rendering their recall impossible by our method.

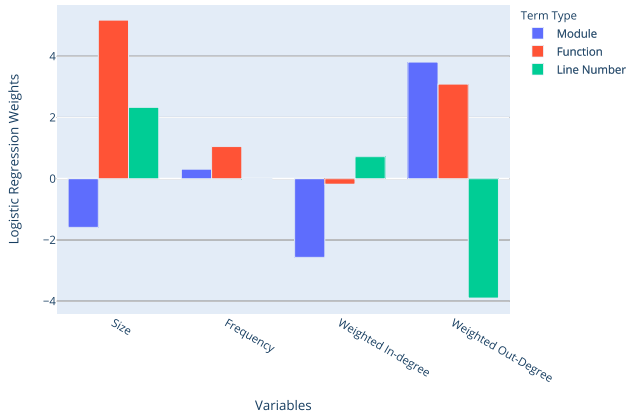

FIGURE 6 | Logistic regression weights for libxml fitted to each term type.

TABLE 4 | Precision and recall of PREDICTFTS algorithm at cut-off probabilities 0.4, 0.5 and 0.6.

	SoX (m)	Librsvg	Libtiff	Freetype	SoX (w)	Mean	Cut-off probability
Precision	0.71	0.57	1.0	0.78	0.67	0.75	
Recall	0.47	0.29	0.71	0.6	0.47	0.50	0.4
Precision	0.71	0.71	1.0	0.76	0.66	0.77	
Recall	0.47	0.22	0.71	0.52	0.47	0.48	0.5
Precision	0.73	0.89	1.0	0.86	0.74	0.84	
Recall	0.44	0.18	0.53	0.52	0.41	0.42	0.6

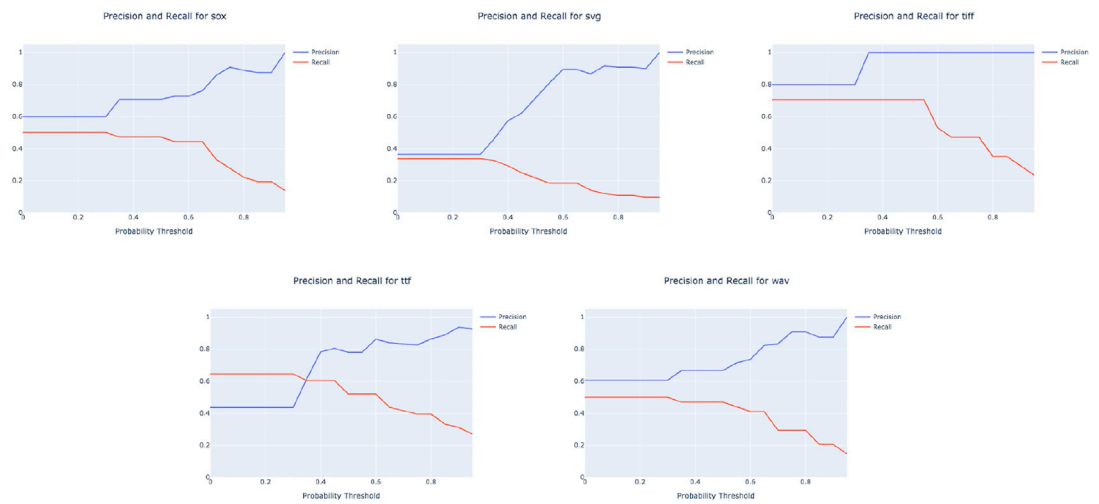

FIGURE 7 | Precision and recall for the PredictFTs algorithm on each program indexed by the cut-off probability parameter.

Example 3. To make call stacks without function terms more readable we insert random words into each predicted frame trace. Keeping with the afl theme, we sample random words from the surnames of champion players from the Richmond Tigers Australian Football League (AFL) team. These words are consistent across the set of call stacks, making it easier for the user to visually compare different call stacks.

Figure 9 demonstrates two reconstructed call stacks from the SoX program. In the original call-stacks, the coloring represents equivalence classes in the model. We color the reconstructed call-stacks the same, noting that when we attempt to reconstruct we do not know what equivalence classes will contain functions a priori.

As is evident, words representing functions are consistent across the set of call-stacks when frame traces are correctly predicted. The ???? terms in the reconstructed call-stacks represent function frame traces that the algorithm did not attempt to predict.

Frame	Module	Function	Line Num.
0	libc.so.6	raise	0
1	libc.so.6	abort	0
2	libglib-2.0.so.0	??	0
3	libglib-2.0.so.0	??	0
4	libglib-2.0.so.0	g_private_get	0
5	libglib-2.0.so.0	g_logv	0
6	libglib-2.0.so.0	g_log	0
7	libglib-2.0.so.0	g_malloc	0
8	libglib-2.0.so.0	g_strdup	0
...	...	...	...

FIGURE 8 | The top eight frames from an example librsvg call-stack.

Original Crashes

Frame	Module	Function	Line Num.
0	fft4g.c	bitrv2	721
1	fft4g.c	makewt	681
2	fft4g.c	____	350
3	effects_i_dsp.c	lsx_safe_rdft	219
4	e.c	dft_stage_init	239
5	e.c	rate_init	443
6	rate.c	start	632
7	effects.c	sox_add_effect	157
8	sox.c	add_effect	708
9	sox.c	add_effects	1073
10	sox.c	process	1759

Frame	Module	Function	Line Num.
0	fft4g.c	bitrv2	721
1	fft4g.c	lsx_rdft	681
2	effects_i_dsp.c	lsx_safe_rdft	219
3	e.c	dft_stage_init	239
4	e.c	rate_init	443
5	rate.c	start	632
6	effects.c	sox_add_effect	157
7	sox.c	add_effect	708
8	sox.c	add_effects	1073
9	sox.c	process	1759

Reconstructed Crashes

Frame	Module	Function Guesses	Line Num.
0	fft4g.c	COTCHIN	721
1	fft4g.c	MARTIN	681
2	fft4g.c	DELEDIO	350
3	effects_i_dsp.c	????	219
4	e.c	RIEWOLDT	239
5	e.c	RANCE	443
6	rate.c	????	632
7	effects.c	RICHARDSON	157
8	sox.c	HOULI	708
9	sox.c	EDWARDS	1073
8	sox.c	????	1759

Frame	Module	Function Guesses	Line Num.
0	fft4g.c	COTCHIN	721
1	fft4g.c	DELEDIO	681
2	effects_i_dsp.c	????	219
3	e.c	RIEWOLDT	239
4	e.c	RANCE	443
5	rate.c	????	632
6	effects.c	RICHARDSON	157
7	sox.c	HOULI	708
8	sox.c	EDWARDS	1073
9	sox.c	????	1759

FIGURE 9 | Two call stacks from the SoX data set, along with their reconstructions using the PredictFTs algorithm.

In **Figure 10**, we explain which fake function names were paired with which line numbers. We describe how each fake function name pairs with an original in the case that it was a correct prediction, as well as which of the two heuristics were used to pair it with the frame trace of a given line number. The only incorrect prediction was the fake function name DELEDIO, which erroneously predicted that the line number 219 was paired with a function in the original call-stack set via heuristic 2.

	Original Function	Function Guesses	Paired Line. Num	Heuristic Applied
Correct	bitrv2	COTCHIN	721	2
Correct	makewt	MARTIN	681	1
Incorrect	N/A	DELEDIO	219	2
Correct	dft_stage_init	RIEWOLDT	443	2
Correct	rate_init	RANCE	632	2
Correct	sox_add_effect	RICHARDSON	157	1
Correct	add_effect	HOULI	708	1
Correct	add_effects	EDWARDS	1073	1

FIGURE 10 | The list of functions whose frame trace was recovered, and the fake function names and line numbers that were paired.

5 RELATED WORK

There is a large collection of research centered on crash triage, in particular in crash de-duplication. The most common tasks are either 1) to automatically de-duplicate full bug-reports submitted to open-source software or 2) to bucket crashes by dissimilarity. In contrast to the setting considered in this paper, most research concerns full bug reports where call-stacks are only a subset of the entire information. In particular, the language in user-reported comments is incorporated, and in some cases the central object [9].

Some of the highest rates of an expert-validated crash duplicate recall are attributed when program execution traces are also recorded [10]. Including call stacks in bug report data has been shown to increase de-duplication recall of full bug reports significantly [11] validating that they are an important object of study in crash triage. We refrained from using the common methods outlined in the above research, showing that a reasonable whitelisting and de-duplication was enough to significantly reduce the number of call-stacks.

Other models of call-stacks exist, albeit with slightly different machinery. For example, the crash-graph defined in [12] serves as a way to graphically compare the similarity between call-stacks. The use of such a model for function frame trace recovery could be an avenue for future research.

6 LIMITATIONS

One of the main limitations of our work is that it is performed on a relatively small data-set. Indeed there are less than 100 distinct call-stacks in each Linux program we have tested, barring the libxml data used to generate the logisitic regression weights. When the set of call-stacks is not very diverse, there may be a tendency to only see function terms with a single line number, making them easier to recover using our method.

A second limitation of our model is that it relies on two fairly naïve patterns. It is not clear if 1) such patterns yield similar results on larger data-sets or 2) whether such patterns could be improved upon or replaced with a more scientific approach. At present, the heuristic method means that our model can only predict function terms that are consistently associated with a single line number across the call-stack set. Our hope is that more sophisticated pattern recognition techniques applied to our topological model could accommodate cases such as frames that pair a particular function with various line numbers. In particular, since the call-stack poset can be thought of as a graph, we expect that more sophisticated techniques from the graph-learning literature could be leveraged 1) in lieu of our logistic regression model and 2) to derive better heuristics and push recall beyond 40 – 50% while preserving precision.

7 CONCLUSION

In summary, our main contribution has been to present a novel topological model to address the problem of function term reconstruction in call-stack data. We performed a small-scale experiment, providing an algorithm to predict the frame-traces of function terms which have been obscured in the call-stack data. Despite the limitations, the performance of the model is relatively encouraging, showing that more information about obscured function terms can be recovered than one may initially suspect. In the future, we envision further research could be done within this framework to improve the recall of the PREDICTFTS algorithm.

We also showed that there is a fundamental lack of diversity in our call-stack data, and we hypothesize that the brute-force nature of fuzzing means that this will probably occur in most data-sets generated by a fuzzer. It is an open question whether our method will work on larger, more diverse call-stack data-sets. Given that some level of dependence between terms is required to form equivalence classes, there is no guarantee that similar results will be achieved.

Lastly, the topological model used here is an example of a larger framework defined in [13]. The extended model is used to tackle applications in gray-box fuzzing, with the goal being to help guide fuzzing campaigns to generate more diverse call-stack data. The use of these models of dependency relations may be applicable in broader contexts outside of fuzzing, such as analyzing dependencies between genes in medical data.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/**Supplementary Material**, further inquiries can be directed to the corresponding author.

AUTHOR CONTRIBUTIONS

VR conceived the project and helped source the data. KM conducted the research and wrote the manuscript with assistance and guidance from VR. Both authors discussed the results and analysis at length.

FUNDING

KM received funding from the Australian Commonwealth Department of Defense under the project title "Mathematical methods for analysis and classification of call-stack data sets". VR was supported by ARC Future Fellowship FT140100604 in the early stages of the project.

ACKNOWLEDGMENTS

The authors are indebted to W.P. Malcolm for suggesting that topological data analysis might be usefully applied to study call-stacks, for assistance in sourcing the data, and many fascinating conversations about probability theory. The paper would not exist without the data provided by Adrian Herrera. The authors gratefully acknowledge many helpful discussions with AH about the art of fuzzing, software compilers, and call-stack jargon.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fams.2021.668082/full#supplementary-material

REFERENCES

1. Gunadi H, and Herrera A. *Experiment Data for MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation* (2019). doi:10.1109/icicos48119.2019.8982513
2. Zalewski M. *Technical Whitepaper for AFL-Fuzz* (2014). doi:10.3726/978-3-653-03549-0 https://lcamtuf.coredump.cx/afl/technical_details.txt.
3. Bartz K, Stokes J, and Platt J. *Finding Similar Failures Using Callstack Similarity*. Redmond WA: Microsoft Corporation (2009).
4. Modani N, Gupta R, Lohman G, Syeda-Mahmood T, and Mignet L. Automatically Identifying Known Software Problems. In: Proceedings - International Conference on Data Engineering (2007). Istanbul, Turkey. doi:10.1109/ICDEW.2007.4401026
5. McCord MC. Singular Homology Groups and Homotopy Groups of Finite Topological Spaces. *Duke Math J* (1966). 33:465–74. doi:10.1215/S0012-7094-66-03352-7
6. Serebryany K, Bruening D, Potapenko A, and Vyukov D. AddressSanitizer: A Fast Address Sanity Checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. Boston, MA: USENIX Association, USENIX ATC'12 (2012). p. 28.
7. Pirttimäki T. *A Survey of Kolmogorov Quotients* (2019). https://arxiv.org/abs/1905.01157.
8. He H. *Imbalanced Learning*. John Wiley & Sons (2011). p. 44–107. doi:10.1002/9781118025604.ch3
9. Runeson P, Alexandersson M, and Nyholm O. Detection of Duplicate Defect Reports Using Natural Language Processing. In: Proceedings - International Conference on Software Engineering (2007). Barcelona, Spain. doi:10.1109/ICSE.2007.32
10. Xiaoyin W, Lu Z, Tao X, Anvik J, and Sun J. An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information. In: Proceedings - International Conference on Software Engineering (2008). Germany. doi:10.1145/1368088.1368151
11. Lerch J, and Mezini M. Finding Duplicates of Your yet Unwritten Bug Report. In: Proceedings of the European Conference on Software Maintenance and Reengineering. Geneva, Italy: CSMR (2013). doi:10.1109/CSMR.2013.17
12. Kim S, Zimmermann T, and Nagappan N. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks. Hong Kong: DSN (2011). p. 486–93. doi:10.1109/DSN.2011.5958261
13. Maggs K. *A Topological Model For Applications In Fuzzing*. Canberra: Mathematical Science Institute, ANU College of Science, The Australian National University (2021). Master's thesis.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.