Probabilistic Naming of Functions in Stripped Binaries

James Patrick-Evans
Royal Holloway, University of London
Egham, United Kingdom
james.patrick-evans.2015@rhul.ac.uk

Lorenzo Cavallaro
King's College London
London, United Kingdom
lorenzo.cavallaro@kcl.ac.uk

Johannes Kinder
Bundeswehr University Munich
Munich, Germany
johannes.kinder@unibw.de

ABSTRACT

Debugging symbols in binary executables carry the names of functions and global variables. When present, they greatly simplify the process of reverse engineering, but they are almost always removed (*stripped*) for deployment. We present the design and implementation of *punstrip*, a tool which combines a probabilistic fingerprint of binary code based on high-level features with a probabilistic graphical model to learn the relationship between function names and program structure. As there are many naming conventions and developer styles, functions from different applications do not necessarily have the exact same name, even if they implement the exact same functionality. We therefore evaluate *punstrip* across three levels of name matching: exact; an approach based on natural language processing of name components; and using *Symbol2Vec*, a new embedding of function names based on random walks of function call graphs. We show that our approach is able to recognize functions compiled across different compilers and optimization levels and then demonstrate that *punstrip* can predict semantically similar function names based on code structure. We evaluate our approach over open source C binaries from the Debian Linux distribution and compare against the state of the art.

CCS CONCEPTS

• Security and privacy → Software reverse engineering; **• Computing methodologies → Machine learning**.

KEYWORDS

binaries, function names, machine learning

ACM Reference Format:
James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Probabilistic Naming of Functions in Stripped Binaries. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427228.3427265

1 INTRODUCTION

Reverse engineering is a crucial step in security audits of commercial software, forensic analysis of malware, software debugging and exploit development. A main task in reverse engineering is to identify functional components in the software and discover the meaning behind different portions of binary code. When faced with a flat region of executable code, it is difficult and time consuming to gain a high-level understanding of what it does. During debugging, software developers can use *symbols* to relate binary code to source-level information such as file names, data structures and names of functions. When releasing software for distribution, however, symbols are routinely *stripped* from the binaries; this decreases the file size and impedes reverse engineering of proprietary code (commercial or malicious).

State-of-the-art reverse engineering tools, such as the IDA Pro disassembler, use databases of function signatures to reliably identify standard functions such as those included by a statically linked C runtime. This works well for systems where libraries are standardized and rarely recompiled. IDA Pro for example, maintains a directory of FLIRT signature files for the most common Windows libraries replicated for the most common compilers and instruction set architectures (ISAs). Reverse engineers also manually create custom databases of such signatures, because they can immediately identify many functions which otherwise would have to be rediscovered in new binaries through costly manual analysis. Such signature-based mechanisms can allow for some variation in the exact byte sequence matched, but they do not go further than relatively simple wildcard mechanisms.

The problem of matching sequences of binary code while allowing for variation presents itself in a number of domains. *Code clone detection* [9, 18, 24, 27, 41], *vulnerable code identification* [11], *code searching* [15], and *software plagiarism detection* [31] address the problem of finding exact matches between software components. They focus on finding a fixed set of previously seen functions with the main contributions drawn from methods of identifying semantically-equivalent code that have undergone various software transformations; these transformations are typical of source code compiled with different compilers or compilation optimizations. Techniques typically adopt static or dynamic approaches that build features of a functions interpreted execution or rely on fixed properties of compiler generated machine code. *Patch code analysis* [24, 52] borrows the same techniques from the problem domain for feature collection however it requires an existing executable with prior information to perform analysis on differential updates.

Common to these domains and approaches is that they aim to identify exact function matches in isolation in *previously seen* executables. They do not provide a way to derive names for functions that do not have an exact (semantic or syntactic) match in the set of known binaries. In contrast, our goal is to learn a general relationship between function names and their binary code.

We implement this approach with *punstrip*, our tool for reversing the stripping process and inferring symbol names in stripped binaries. *Punstrip* builds a probabilistic model that learns how developers use and name functions across a set of existing open source projects; using this model, we infer meaningful symbol information based on similarities in program structure and semantics in previously unseen, stripped binaries. It is not necessary to discover

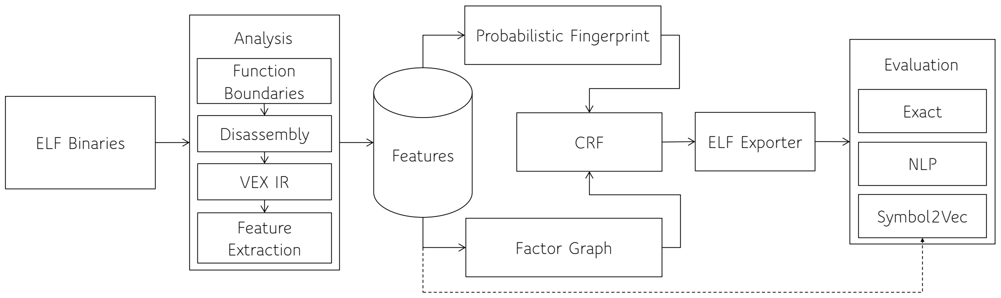


Figure 1: A block diagram overview of the components in *Punstrip*.

exactly the same identifiers that the developers used in the original program: for reverse engineering, we are interested in discovering symbol names that are helpful to an analyst. With *punstrip*, reverse engineers are able to pre-process an unknown binary to automatically annotate it with symbol information, saving them time and preventing mistakes in doing further manual analysis. We make *punstrip* available as open source[1].

We make the following contributions:

- We present a novel approach to function identification and signature recognition for stripped binaries that uses features in a higher-level intermediate representation. This approach can scale to real world software and seeks to be agnostic to both compiler architectures, binary formats, and optimizations.
- We introduce a probabilistic graphical model for inferring function names in stripped binaries that compares the joint probability of all unknown symbols simultaneously rather than treating each function in isolation. The model builds on our probabilistic fingerprint and analysis between symbols in binaries.
- We describe Symbol2Vec, a new high dimensional vector embedding for function symbols. We demonstrate that the embedding is meaningful by creating a set of relationships within the space of function names drawn from C binaries distributed as part of Debian GNU/Linux. We use Symbol2Vec as one metric in the evaluation of *punstrip* to capture relations between function names that do not share any lexical components. We release our vector embeddings[2] to the research community to provide a common method of evaluating semantically similar function names in compiled C binaries.

We evaluate *punstrip* against the current state of the art in function name detection against all C binaries in Debian with 10-fold cross validation. Furthermore, we evaluate our probabilistic fingerprint against leading tools using binaries with a large common code base that were compiled in different environments.

In the remainder of this paper, we give an overview of *punstrip*'s pipeline (§2), introduce our technical approach to the problem of function fingerprinting (§3), and describe the abstract graphical structure for learning (§4). We then present our method for relating function names, including Symbol2Vec (§5), before evaluating *punstrip* against previous work (§6). Finally, we discuss limitations of our approach (§7), contrast with related work (§8) and present our conclusions (§9).

2 OVERVIEW

Figure 1 shows an architectural overview of the *punstrip* pipeline. *Punstrip* takes as input a set of ELF binaries, which for training should be unstripped. In the initial analysis stage, *punstrip* extracts function symbols and their boundaries as defined in the symbol table, disassembles them, and lifts the instructions to the VEX intermediate representation. From this representation and interprocedural control flow information among functions, *punstrip* extracts a set of features that are stored in a database. Those features are used to build a per-function fingerprint, as well as a factor graph representing the relationships between functions and feature values for each executable. Our probabilistic fingerprint and factor graph are used to construct a Conditional Random Field (CRF) that learns how individual functions interact with other *code* and *data*.

After training a model on a large corpus of programs that include symbol information, we are able to use the learned parameters to infer the most likely function names in *stripped* binaries. The inferred function symbol names can then be added to the *stripped* binary and used for debugging and reverse engineering purposes. In this paper, we focus exclusively on the problem of naming functions; for detecting function boundaries in stripped binaries, we refer to recent approaches from the literature [2, 3].

We evaluate the accuracy of labeling with three metrics: (1) exact matches of function names, (2) normalized matches of function names, and (3) Symbol2Vec, a function name embedding based on random walks on callgraphs taken from a large dataset of binaries. In the remainder of this section we detail the individual stages of the pipeline referring to examples in Figure 2, in which we separate known from unknown functions, that we aim to label.

2.1 Probabilistic Fingerprint

Figure 2a shows the disassembly of a function from a stripped binary. Static analysis is able to detect that the unknown function is called from the unnamed function at address 0x0162d and calls the dynamically linked function errno_location (which gets the

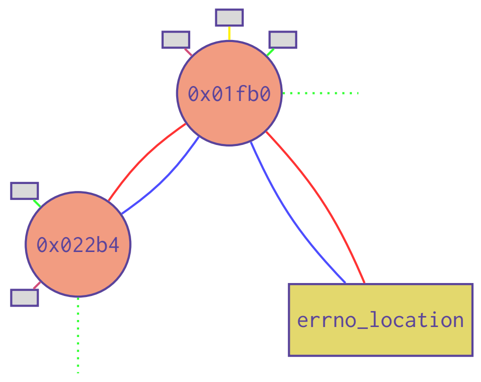
[1]https://github.com/punstrip/punstrip
[2]https://github.com/punstrip/Symbol2Vec

```
(fcn) 0x01fb0
; CALL XREF from 0x0000162d 0x0162d
0x00001fb0      test rdi, rdi
0x00001fb4      je 0x2027
0x00001fb6      mov esi, 0x2f
0x00001fbb      mov rbx, rdi
0x00001fbe      call imp.errno_location
0x00001fc3      test rax, rax
0x00001fc6      je 0x2017
0x00001fc8      lea rdx, [rax + 1]
0x00001fcc      mov rcx, rdx
0x00001fcf      sub rcx, rbx
0x00001fd2      cmp rcx, 6
0x00001fd6      jle 0x2017
0x00001fdf      mov ecx, 7
0x00001fe6      jne 0x2017
0x00001fed      call 0x022b4
...
```

(a) Disassembly of an unnamed function in a stripped dynamically linked ELF executable.

```
IRSB {
   t1:Ity_I64 t2:Ity_I64 t3:Ity_I64 t6:Ity_I64
        t7:Ity_I1
   00 | ------ IMark(0x1fb0, 3, 0) ------
   01 | t2 = GET:I64(rdi)
   02 | PUT(cc_op) = 0x0000000000000014
   03 | PUT(cc_dep1) = t2
   04 | PUT(cc_dep2) = 0x0000000000000000
   05 | PUT(pc) = 0x0000000000001fb3
   13 | ------ IMark(0x1fb4, 2, 0) ------
   14 | t15 = CmpEQ64(t2,0x0000000000000000)
   15 | t14 = 1Uto64(t15)
   16 | t12 = t14
   17 | t16 = 64to1(t12)
   18 | t7 = t16
   19 | if (t7) { PUT(pc) = 0x2027; Ijk_Boring }
   NEXT: PUT(rip) = 0x0000000000001fb6; Ijk_Boring
}
...
```

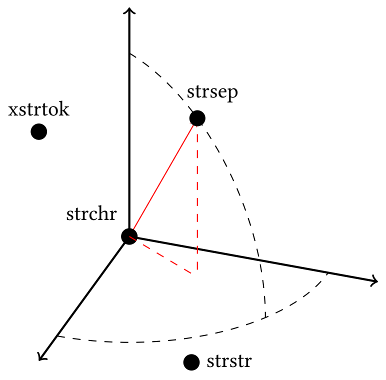
(b) VEX Intermediate Representation (IR) of the unknown function's first basic block.



(c) General graph based conditional random field based on relationships between features, knowns, and unknowns.

```
(fcn) set_program_name
; CALL XREF from 0x0000162d main
0x00001fb0      test rdi, rdi
0x00001fb4      je 0x2027
0x00001fb6      mov esi, 0x2f
0x00001fbb      mov rbx, rdi
0x00001fbe      call imp.errno_location
0x00001fc3      test rax, rax
0x00001fc6      je 0x2017
0x00001fc8      lea rdx, [rax + 1]
0x00001fcc      mov rcx, rdx
0x00001fcf      sub rcx, rbx
0x00001fd2      cmp rcx, 6
0x00001fd6      jle 0x2017
0x00001fdf      mov ecx, 7
0x00001fe6      jne 0x2017
0x00001fed      call strchr
...
```

(d) Disassembly of the same function with inferred function names added to the binary's symbol table.



(e) A projection of the most similar Symbol2Vec embeddings to strchr.

Figure 2: Stages in *punstrip*'s pipeline for learning and evaluating function name inference on stripped binaries.

error identifier from the last executed system call) before calling 0x022b4.

As the binary sequence of machine code can differ even for the same source code depending on compilation settings, we opt for using an intermediate representation that abstracts some implementation detail. We lift machine code to the VEX intermediate representation, as shown in Figure 2b, which offers an appropriate level of detail and abstraction for our analysis. From this representation, we extract a collection of features that help identify names of functions designed to be agnostic to changes in compilers and optimizations. While the binary code of a function may change due to compiler differences, and hence values of our features, our intuition is that even if a compiler modifies the order, number of, and type of instructions, changes in feature values based on optimized VEX IR will still be similar.

After extracting a set of features we first convert each feature into a vectorized form and use it as an input to a multiclass classifier. The output of multiclass classification gives a probability mass function over the set of all unknown functions in our training data. Thus for each function, we learn a probability distribution over all function names given input features derived from VEX.

2.2 Probabilistic Structural Inference

After extracting high-level features and relationships between functions, we build a probabilistic graphical model in the form of a Conditional Random Field (CRF) [29, 49] as depicted in Figure 2c. The CRF operates on a factor graph which factorizes inter-dependencies between functions and separates unknown and known information. Figure 2c shows that the unknown function 0x01fb0 has a factor based on calling the known function errno_location, factors based on statically derived known features such as its probabilistic fingerprint, and a factor based on its relationship with 0x022b4.

Our CRF models pairwise and generic factor-based relationships among *code* and *data* for all functions in an executable. We first train the graphical model and learn the weightings of relationships on a large set of prior unstripped binaries before building a new CRF for each stripped binary and applying the learned model parameters. Function names are then inferred by maximizing the conditional probability of unknown function names, given the known information and our models parameters. This enables *punstrip* to take into consideration known information from the whole binary simultaneously rather than considering each function in isolation; this may help identify functions that are weakly identifiable in and of themselves, but have strong connections to other more easily recognizable functions.

2.3 Function Name Matching and Evaluation

Finally, after we have inferred function names for the set of unknown symbols, we modify the ELF executable's symbol table and insert entries into the `strtab` and `symtab` sections. As a result, subsequent disassembly will yield the newly predicted function names (Figure 2d).

The names developers give to functions can vary wildly based on personal preferences and project styles. Therefore, relying on exact lexical matching only to evaluate the accuracy of name predictions would miss cases where predicted names are semantically correct but syntactically different. We therefore implement two additional metrics for evaluating name similarity. First, we propose a method based on natural language processing, which uses lexical techniques to determine the similarity of two names. The metric tries to mitigate grammatical differences in language used, expands common naming conventions used by programmers, and takes into account similar words, such as `start` and `begin`, within each name. Second, we introduce Symbol2Vec, a new embedding for function names based on the callgraph of programs. As it is only based on caller-callee relationships, it completely abstracts away the text of function names and provides a similarity metric based only on how software developers use and name functions. Figure 2e demonstrates how Symbol2Vec embeddings are used when evaluating the similarity of function names.

3 PROBABILISTIC FINGERPRINT

We now explain how we create a probabilistic fingerprint for each function. We give an overview of how we extract features using static (§3.1) and symbolic (§3.2) analysis, and we detail how extracted features are combined into a probabilistic identifier (§3.3).

3.1 Static Analysis

All features extracted from each function are listed in Table 1. We include two low-level features that help to find exact matches: a hash of the machine code and a hash of the opcodes in the disassembly. The opcode hash is included to recognize exact patterns of generated machine code with different parameters or relative offsets, which would not be matched by an exact binary hash. All other features are extracted from VEX IR. Our choice of VEX IR comes with the advantage of VEX providing a more abstract view than alternative representations and not requiring to deal with low-level details of the machine state, such as the EFLAGS register [28].

Symbols contained in an ELF binary's symbol table detail a component's address, size and string description in the target executable. This information is first extracted along with the raw bytes corresponding to the function. Using *capstone* [39], *pyvex*, and each function's boundaries as specified in the binary's symbol table [4], we lift each basic block into its optimized VEX IR and build a labeled Intraprocedural Control Flow Graph (ICFG) for each function. We then resolve dynamically-linked objects and build a callgraph for each statically-linked function in the binary.

We track all features given in Table 1 and convert this structure of features into a form that can be represented by a single stacked vector to be used as a fingerprint for each function. When labeling the ICFG, VEX basic blocks are distinguished by their terminators (jumps, calls, returns, and fall-throughs). We only store numeric integer constants, greater than 2^8, that are not operands of jump instructions to focus on infrequent and distinctive values.

After machine code is lifted into the VEX IR we categorize each instruction into a one-hot encoded vector according to the regular expressions defined in Table 2. The one-hot encoded vectors are then summed to produce an impression of functions operations.

To convert generic graphical structures to a vector representation we utilize the feature embedding technique *graph2vec* [40]. We compare the similarity of all ICFGs by using an implementation of the Weisfeiler-Lehman graph kernel [44]. By training a *graph2vec*

Table 1: Features extracted from functions and their representation in the probabilistic fingerprint.

Feature	Type	Description
Static features		
Size	Scalar	Size of the symbol in bytes.
Hash	Binary	SHA-256 hash of the binary data.
Opcode Hash	Binary	SHA-256 hash of the opcodes.
VEX instructions	Scalar	Number of VEX IR instructions.
VEX jumpkinds	Vector(8)	VEX IR jumps inside a function e.g. *fall-through*, *call*, *ret* and *jump*
VEX ordered jumpkinds	Vector(8)	A ordered list of VEX jumpkinds.
VEX temporary variables	Scalar	Number of temporary variables used in the VEX IR.
VEX IR Statements, Expressions and Operations	Vector(54)	Categorized VEX IR Statements, Expressions and Operations.
Callers	Vector(N)	Vector one-hot encoding representation of symbol callers.
Callees	Vector(N)	Vector one-hot encoding representation of symbol callees.
Transitive Closure	Vector(N)	Symbols reachable under this function.
Basic Block ICFG	Vector(300)	Graph2Vec vector representation of labeled ICFG.
VEX IR constants types and values	Dict	Number of type of VEX IR constants used.
Symbolic features		
Stack bytes	Scalar	Number of bytes referenced on the stack.
Heap bytes	Scalar	Number of bytes referenced on the heap.
Arguments	Scalar	Total number of function arguments.
Stack locals	Scalar	Number of bytes used for local variables on the stack.
Thread Local Storage (TLS) bytes	Scalar	Number of bytes referenced from TLS.
Tainted register classes	Vector(5)	One-hot encoded vector of tainted register types, e.g., stack pointer, floating point.
Tainted heap	Scalar	Number of tainted bytes of the heap.
Tainted stack	Scalar	Number of tainted bytes of the stack.
Tainted stack arguments	Scalar	Number of tainted bytes that are function arguments to other functions
Tainted jumps	Scalar	Number of conditional jumps that depend on a tainted variable.
Tainted flows	Vector(N)	Vector of tainted flows to known functions.

model, each ICFG is converted into a vector space in which similar graphical structures are numerically similar. We store each vector in an Annoy Database[3] that allows us to quickly find the nearest vectors for each graph based on the Euclidean distance from our model's embeddings (and hence the most similar graphs). Training the *graph2vec* model is computationally expensive; however, it allows us to avoid comparing pairs of graphs with a graph kernel over the testing set for every element in the training set. Using *graph2vec* we are able to compare the similarity of abstract graphical structures in $O(1)$ after training the model.

3.2 Symbolic Analysis

We extract additional semantic features using our own symbolic analysis built on top of the VEX IR. We write our own execution engine over the existing ANGR [48] implementation to provide a lightweight and more consistent analysis across multiple platforms. After the boundaries of each basic block are known from the initial static analysis, we are able to lift each block into VEX in Single Static Assignment (SSA) form. Within our model, reads from registers and memory locations with undefined contents return symbolic values for the size of data requested.

[3]Annoy: Approximate Nearest Neighbors Oh Yeah: https://github.com/spotify/annoy

Function Argument Extraction. To identify the number of function arguments we carry out live variable analysis on argument registers and memory references to pointers above the current stack pointer. As we need to track the value of the stack pointer, we perform a fixed point iteration algorithm to determine the base and stack pointer values on each use. Finally, we build a model to track memory references between basic blocks as the VEX IR's SSA form is only consistent per basic block.

Heap and Stack Analysis. We implement a stack of 2048 bytes starting at `0x7FFFFFFFFFFF0000` and model the stack registers, segment registers, and heap accordingly. For each function, we track the total number of bytes referenced on both the heap and stack, local variables and function arguments placed on the stack, thread local storage accesses, and perform taint analysis to calculate data flows from each input argument to arguments of other functions. Finally, we compute the transitive closure for each function under the binary's callgraph.

Symbolic Execution. After identifying the number of input arguments to a function we symbolically execute the function using our own execution engine that uses Claripy [10, 47] to formulate symbolic values and expressions. This allows us to create symbolic expressions for return values from each function, e.g.

Table 2: Rules for VEX IR categorization.

Regex	Description
VEX IR Operations	
`Iop_Add(.*)`	Addition
`Iop_Sub(.*)`	Subtraction
`Iop_Mul(.*)`	Multiplication
`Iop_Div(.*)`	Division
`Iop_S(h|a)(.*)`	Arithmetic and logical shifts
`Iop_Neg(.*)`	Negation
`Iop_Not(.*)`	Logical NOT
`Iop_And(.*)`	Logical AND
`Iop_Or(.*)`	Logical OR
`Iop_Xor(.*)`	Logical XOR
`Iop_Perm(.*)`	Permute bytes
`Iop_(.*)to(.*)`	Type conversion
`Iop_Reinterp(.*)as(.*)`	Reinterpretation
`Iop_(Cmp|CasCmp)(.*)`	String comparison
`Iop_Get(M|L)SB(.*)`	Get significant bit
`Iop_Interleave(.*)`	Bit interleaving
`Iop_(Min|Max)(.*)`	Min/max operations
Statements	
`Ist_Exit`	Exit
`Ist_IMark`	Instruction marker
`Ist_MBE`	Exit
`Ist_Put_(.*)`	Put
`Ist_(Store|WrTmp)`	Write

$ret \models SymbVec(ARG1) + BitVec(0x2)$. Where our symbolic execution engine cannot easily determine the result of an operation, e.g. the `x86_64` instruction `AESENC`, we inject symbolic values.

After identifying symbolic variables we are able to extract call sites to other functions that are control-dependent on a symbolic variable. We then track the number of call sites that is control-dependent on each input argument and use it as a feature in our fingerprint. We run our analysis for every recovered function argument to extract per input-dependent taints. Finally, we also include an analysis pass that taints all input arguments to mitigate against reordering of function arguments producing different results.

Register Classification. We classify registers referenced during execution into five generic classes: general purpose, floating point, stack and base pointer, segment register, and control register. For each input argument we produce a vector of tainted register classes from the set of tainted registers. This allows *punstrip* to capture the types of behavior performed by functions for individual arguments. Finally, we produce a final vector of tainted register classes irrespective of taint.

3.3 Probabilistic Classification

We aim to convert features extracted in Table 1 to a probability distribution over a corpus of symbol names $s \in \mathrm{S}$. For *binary* and *dictionary* typed features we emit a vector of size $|\mathrm{S}|$ if the feature has been seen in our training dataset for each function name. We

Table 3: Feature functions used in the CRF. *label-node* relationships relate known features $x \in \mathrm{x}$ to the current node y_u. *label-label* relationships relate unknown nodes $y_u \to y_v \in \mathrm{y}$.

Relationship	Description
label-node relationships	
Probab. fingerprint	The probability of function y_u given its extracted features in Table 1.
label-label relationships	
d^{th} pairwise callers	The probability of of function y_u calling y_v through $d-1$ other nodes.
d^{th} pairwise callees	The probability of of function y_u being called by y_v through $d-1$ other nodes.
Pairwise data xrefs	The probability of of function y_u referencing object x_v.
Generic factor callers	The probability of of function y_u calling the set of known functions x.
Generic factor callees	The probability of of function y_u being called by the set of known functions x.

stack the resultant scalars and vectors into a single feature vector to be used as the input to a machine learning classifier. As the feature vector is sparse we reduce its dimensionality by performing Principal Component Analysis (PCA) and scale the transformed principle components such that each column has 0 mean and a unit variance.

Finally, we train a Gaussian Naive Bayes[4] model to predict the probability of each input function belonging to $s \in \mathrm{S}$. Our model is implemented using ScikitLearn [40].

4 PROBABILISTIC STRUCTURAL INFERENCE

In this section we explain how we combine our probabilistic fingerprint with a third order general graph based CRF for symbol inference. First we explain how we generate the CRF (§4.1) using relationships between multiple symbols and features of individual functions. We then explain how *punstrip* performs parameter estimation (§4.2) and finally inference (§4.3).

4.1 CRF Generation

We refer to the process of symbol inference as predicting the most likely symbol names using a probabilistic graphical model that utilizes unary potentials from our probabilistic fingerprint and known nodes, pairwise potentials between unknown functions, and generic factor potentials between sets of unknowns and knowns.

In general, CRFs are used to predict an output vector $\mathrm{y} = \{y_0, y_1, \ldots, y_N\}$ of random variables that may have dependencies on each other given an observed input vector x. Our goal is that of structured prediction, or learning high-level relationships between symbols. Modeling the dependencies between all symbols in binary executables would most likely lead to an computationally intractable graphical model, therefore we use a discriminative

[4]We found Gaussian Naive Bayes out-performed Random Forests, Logistic Regression, and Neural Networks.

approach and model the conditional distribution $p(\mathrm{y} \mid \mathrm{x})$ directly without needing to model $p(\mathrm{x} \mid \mathrm{y})$.

As depicted in Figure 3, the CRF built is of the general graph form with relationships between known functions, known features (known features of unknown functions), and unknown functions. In our model, known vertices represent feature values or known function names, e.g., $Size = 5, name = read$; unknown vertices represent unknown symbol names. Edges between nodes represent relationships between feature values of which we define two types: label-observation and label-label. Label-observation edges represent relationships connecting known nodes in x to unknown nodes in y and label-label edges represent relationships between unknown nodes in y. Each feature function is replicated for each symbol name $s \in \mathrm{S}$. This is implemented as a vector N of size $|\mathrm{S}|$ with each element $n \in \mathrm{N} \rightarrow [0, 1]$. Our implementation exploits the sparsity between connected functions across millions of unique function names by storing each vector in a sparse matrix.

The feature functions used in building the CRF are listed in Table 3. For pairwise feature functions, we track dependencies to the d^{th} degree for $d \in \{1, 2, 3\}$. To clarify, under the $callee_d$ feature function, each edge potential is a probability distribution over all known symbol names S which describes the probability of the symbol name transition $s_u^d \rightarrow s_v^d$. This represents the probability of a symbol s_u being d calls away from s_v.

The CRF aims to predict the conditional probability over all unknown nodes y simultaneously given the set of known nodes x. Let G be a factor graph of relationships over all known symbols x and all unknown symbols y, then (x, y) is a conditional random field if for any value $x \in \mathrm{x}$, the distribution $p(\mathrm{y} \mid \mathrm{x})$ factorizes according to G. If we partition the graph G into maximal cliques $C = \{C_1, C_2, ..., C_P\}$ and into a set of factors $F = \{\Psi_c\}$, then the conditional distribution for the CRF is given by:

$$p(\mathrm{y} \mid \mathrm{x}) = \frac{1}{\mathcal{Z}(\mathrm{x})} \prod_{C_p \in C} \prod_{\Psi_c \in C_p} \Psi_c(\mathrm{y}_c, \mathrm{x}_c; \theta_p) \tag{1}$$

where $\mathcal{Z}(\mathrm{x})$ is a normalizing constant.

All of our label-label feature functions are discrete and return 0 or 1 for each function name depending on if the relationship exists in the training set. The weightings for pairwise feature functions are repeated for each clique and can be thought of as a global matrix between all function names $\mathrm{N} \times \mathrm{N}$. Then there exist $|\mathrm{N}|$ pairwise feature functions between a known and unknown node per relationship, e.g., for the first callee relationship, the probability of a known function being called by every other function in S. We set Ψ_c to be log linear for efficient inference and define it in the usual way as follows:

$$\Psi_c(\mathrm{y}_c, \mathrm{x}_c; \theta_p) = \exp\left(\sum_{k=1}^{K(p)} \theta_{pk} f_{pk}(\mathrm{y}_c, \mathrm{x}_c)\right) \tag{2}$$

whereby $K(p)$ returns the feature functions connected to vertex p. Both weightings θ_{pk} and feature functions f_{pk} are indexed by vertex k and factor p implying that each factor has its own set of weights. As the graphical structure of binaries is not fixed, and hence the structure of our CRF, our implementation replicates the weightings of each feature function globally. The normalization constant $\mathcal{Z}(\mathrm{x})$ is defined as

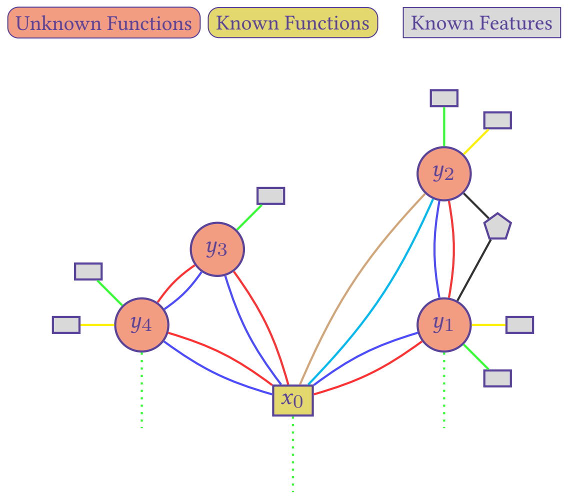


Figure 3: A visualisation of a snapshot of the general graph based condition random field showing known and unknown nodes and the relationships between them. Different types of relationships are represented by separate colors. Pairwise and generic factor based feature functions are represented by rectangles and polygons.

$$\mathcal{Z}(\mathrm{x}) = \sum_{\mathrm{y}} \prod_{c_p \in C} \prod_{\Psi_c \in C_p} \Psi_c(\mathrm{y}_c, \mathrm{x}_c; \theta_p) \tag{3}$$

4.2 Parameter Estimation

To estimate the weightings associated with the CRF we use a maximum likelihood approach, i.e., $\boldsymbol{\theta}$ is chosen such that the training data has the highest probability under the model. We achieve this by maximizing the pseudo log-likelihood given by Equation 4 over all of our training set graphs $g \in \mathcal{G}$.

$$\ell(\boldsymbol{\theta}) = \sum_{g}^{\mathcal{G}} \sum_{p}^{C} \sum_{k=1}^{K(p)} \theta_{pk} f_{pk}(\mathrm{y}_c, \mathrm{x}_c) - \frac{\theta_{pk}^2}{2\sigma^2} \tag{4}$$

As we aim to expect changes in structure and features we regularize the log likelihood with Tikhonov regularization [50] so that we do not overfit our model. We combine L-BFGS-B [8] on subgraphs in the training set with stochastic gradient descent to iteratively learn the optimal weightings for $\boldsymbol{\theta}$.

As we assume a large collection of independent and identically distributed samples in the training data, using a numerical approach to maximizing the likelihood in a batch setting is unwarranted and needlessly slow. We suspect that different items in the training data from disconnected graphs provide similar information about relationship parameters; therefore we opt to using a stochastic method for optimizing the likelihood. While such an approach is sub-optimal, we believe the trade off for training the CRF on big code is acceptable.

4.3 CRF Inference

Whilst exact inference algorithms exist for linear-chain and tree based models, in the general case the problem has been shown to be NP-hard. For inferring symbol names from the CRF we employ Maximum a Posteriori (MAP) estimation using the approximate inference method Loopy Belief Propagation [36] combined with an optimized greedy algorithm based on stochastic gradient descent. We use an approximate inference method in order to model large and complex graphical structures with the possibility for many loops whilst still being a tractable model for convergence. Our greedy algorithm works by making small changes to a subset of the most confident nodes in the model. In each run of Loopy Belief Propagation we use a random permutation of message updates to avoid local minima. By combining the two approaches it is hoped the model falls into a global minimum rather than a weak local minimum. The use of the CRF gives the possibility of inferring functions that have large machine code differences to previous instances based on the interactions with other known and unknown functions that are more easily recognizable.

5 MATCHING FUNCTION NAMES

We now turn to the issue of matching semantically similar function *names* using both existing Natural Language Processing (NLP) techniques (§5.1) and *Symbol2Vec* (§5.2).

5.1 Lexical Analysis

When inferring symbol names based on heuristics of the underlying code, it is difficult to know if the inferred name is correct. As previously mentioned in §3.3, multiple symbol names have exactly the same machine code, e.g., `xstrtol`, `strtol` and `__strtol` have the same byte sequence for the same compiler settings and come from different software packages. For this reason, we perform a series of measures adopted from NLP to compare the differences between the inferred symbol name and the ground truth.

We first pre-process all function names to remove common character sequences such as capital letters surrounded by underscores used to signify library versions, CPU extension named functions such as `function.avx512`, added compiler notation such as `function.constprop` and `function.part`, and ISA-specific naming of functions. This significantly reduces the number of unique symbol names stored in our database. Upon comparing the names of functions for a possible match, we first calculate the Levenshtein [30] distance between the symbol names to detect small changes similar to appending a suffix or prepending a prefix. Secondly we perform canonicalization and tokenization on both the inferred and ground truth before lemmatizing and word stemming [21] each token in order to match words of different tenses and cases. This enables us to match the symbol `wd_compare` and `wd_comparator` based on the stemmed word `compar`. In name canonicalization we maintain a list of common programming abbreviations such as `fd` for 'file descriptor' or `dir` for 'directory' and then use the dynamic programming rod cutting algorithm to match sub-sequences with a scoring function that prefers longer word lengths in order to produce a set of word descriptions for each function name. For example, the real function name `hexCharToInt` after symbol canonicalization is represented by the set $\{hexadecimal, character, to, integer\}$. We then use synonym sets from the Wordnet [35] lexical database for the English language to compare the synonyms of individual descriptions. A naming similarity score is produced based on the Jaccard distance between the matching canonicalization sets x_c and s_c as given by:

$$d_j(x_c, s_c) = \frac{|x_c \cup s_c| - |x_c \cap s_c|}{|x_c \cup s_c|} \tag{5}$$

The Jaccard distance gives a measure of the overlapping similarity in the synonyms of the canonical names for each function name. When the distance falls below a given threshold we deem the function descriptions to be similar.

This method aims to implement a subjective match on the similarity between function names but may introduce false positives into our results. However, our thresholds and techniques were derived from manual analysis in order to align function name similarity close to the decisions of a human analyst.

Table 4: Examples of five target words and their closest vector representations in Symbol2Vec using the cosine distance.

Target Symbol	Closest Vectors in Symbol2Vec
`grub_error`	`grub_video_capture_set_active_render_target`, `grub_crypto_gcry_error`, `grub_font_draw_glyph`, `grub_disk_filter_write`
`opendir`	`readdir`, `closedir`, `dirfw`, `rewinddir`, `readdir_r`, `fdopendir`
`tls_init`	`tls_context_new`, `mutt_ssl_starttls`, `tls_deinit`, `eap_peer_sm_init`, `initialize_ctx`
`tor_x509_cert_get_cert_digests`	`tor_tls_get_my_certs`, `should_make_new_ed_keys`, `router_get_consensus_status`, `we_want_to_fetch_unknown_auth_certs`
`clock_start`	`clock_stop`, `lindex_update_first`, `lindex_update`, `index_fsub`, `index_denial`

5.2 Symbol2Vec

Choosing an appropriate label for a function is a subjective goal in which different entities may choose different labels for the same function. The majority of the time we would hope that these labels are similar for functionally equivalent code and exhibit a subset of natural language features so that they can be compared similar in our NLP matching stage (§5.1). Unfortunately one programmer may choose a different name to that of another that does not match in our NLP comparison whilst still being functionally relevant. For example, consider the two real world functions that start a network connection to a remote server and return a file handler named `init_connection` and `get_resource_handler`. The two functions share no lexical similarities and would be matched as different function names under normal conditions. We create a numerical method to serve as a metric for name similarity which is able to alleviate this problem. We do this by projecting symbol names into a high dimensional vector space such that functions which are semantically similar appear close in the vector space and functions that differ are far apart.

Analogous to *Word2Vec* [33], we modify the Continuous Bag Of Words (CBOW) and Skip-Gram model in order to create *Symbol2Vec*;

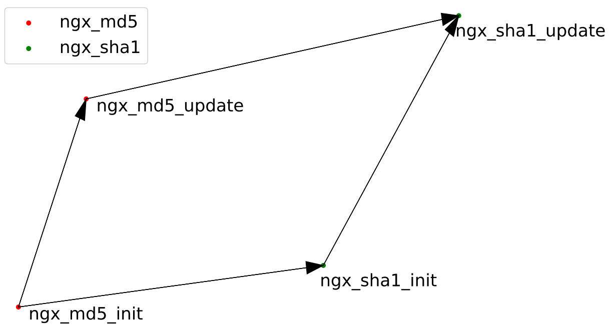


Figure 4: t-SNE plot of the closest vectors for the SHA-1 and MD5 hash algorithm relationship existing in Symbol2Vec.

a vector representation of function names. We replace the CBOW model with our own Continuous Bag Of Functions (CBOF) that uses the callgraph of binaries in order to predict the surrounding context functions given a pivot function. Using the CBOF and a context window of 1 we randomly sample a pivot function name with an associated target function name that is either a callee or caller of the pivot function.

For a large corpus of function names, very common functions provide less information than rare function names. Therefore we use a sub-sampling approach as per [34] to discard function names based on the probability defined by the following formula:

$$p(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{6}$$

where $f(w_i)$ is the frequency of the function name w_i in our corpus and t is an arbitrary threshold that we took to be 10^{-5}. This step reduced our set of unique function names removing the likes of `malloc`, `free`, and `csu_init`. We use TensorFlow [17] to create a autoencoder using 150 hidden nodes and 800,000 input and output nodes; one to represent each function name using one-hot encoding after the sub-sampling stage. We then train our neural network using Stochastic Gradient Descent (SGD) and Noise Contrastive Estimation (NCE) with negative sampling on a server with 256 GiB RAM and a Intel(R) Xeon(R) Gold 6142 CPU for three days to minimize the loss between predicting the pivot words from its context. The resulting weights of the hidden layer when activated by the input function name form the vector representation for each function name. We display function names and their nearest neighbors in our *Symbol2Vec* vector space for selected functions in Table 4, which demonstrates how semantically similar function names are grouped close together.

We use all function names found in or referenced by the code section of ELF binaries for C executables in Debian which resulted in 17,549 binaries, 5 million unique symbol names, and 1.1 million function names after our naming pre-processing step. Analogous to the classic *Word2Vec* analogy $King - Man + Woman = Queen$, we are able to reveal analogical relationships between function names. One such analogy is that between hash functions used in the *Nginx* software package, where the closest vector resulting from `ngx_md5_update` − `ngx_md5_init` + `ngx_sha1_init` is the function name `ngx_sha1_update` as shown in Figure 4.

Table 5: Lexical analogies in Symbol2Vec, where $\mathbf{a} - \mathbf{b} + \mathbf{c} \approx \mathbf{d}$.

a	b	c	d
sha1_init_ctx	md5_init_ctx	md5_update	sha1_update
realloc	malloc	xmalloc	xrealloc
fopen	open	close	fclose
icmp_open	open	close	icmp_close
hci_connect	connect	disconnect	hci_disconnect
close_log_file	fclose	fopen	open_log_file
sendmsg	write	recv	recvmsg
closepipe	close	open	openpipe
nxt_recv_file	recv	send	nxt_send_file
gethostent	get	set	sethostent
sha2_hmac_update	sha2_update	md5_update	md5_hmac_update
http_server_init	init	close	http_server_close
usb_bulk_write	write	read	usb_bulk_read
OPENSSL_CTX_init	init	free	OPENSSL_CTX_free
ssh_connect	connect	disconnect	ssh_disconnect
dhcpcd_config_get	dhcpcd_config_set	fopen	fclose
socket_accept	accept	close	socket_close
SQLConnect	connect	disconnect	SQLDisconnect
gethostname	get	set	sethostname
csu_fini	fini	csu_init	init
usb_bulk_send	send	recv	usb_bulk_recv
SHA384File	SH384_Init	SHA512_Init	SHA512File
Hread	read	write	Hwrite
SHA1File	SHA1_Init	MD5_Init	MD5File
btconnect	connect	disconnect	btdisconnected
EndDocFile	fclose	fopen	BeginDocFile
nxt_send_buf	send	recv	nxt_recv_buf
SHA256File	SH256_Init	SHA1_Init	SHA1File
cprng_deinit	free	malloc	cprng_init
unix_sock_open	open	close	unix_sock_close

The dense representation of function names by Symbol2Vec allows us to numerically express the similarity of function names. We evaluate the effectiveness of our dense representation in the standard way by comparing the correlation between the generated representation and a manually created list of lexical relationships. Our analogies are in the form $\mathbf{a} - \mathbf{b} + \mathbf{c} \approx \mathbf{d}$. The full list of analogies can be seen in Table 5.

We measure the distance between function name vectors in the ordinal sense as we are unable to produce an exact continuous measure for our manual analogies. For each of our analogies we evaluate the preceding formula on the vector representations of each word and then rank each word vector based on its cosine distance to the new point with the closest vector having rank, and hence distance, 0.

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{7}$$

The Spearman's rank correlation is then calculated as per Equation 7 with n being the number of observations and d_i is the ordinal distance $\mathbf{d}$ is away from $\mathbf{a} - \mathbf{b} + \mathbf{c}$ when compared to all function name vectors for analogy i.

We show that Symbol2Vec is a meaningful representation and the distance between vectors in this space strongly correlates to their semantic similarity. We achieve a Spearman's rank correlation coefficient of 0.97 between Symbol2Vec and our manually crafted analogies. Thus proving a strong correlation between our semantic analogies created by a human analyst and Symbol2Vec. In order for its use in the community and other similar applications, we release Symbol2Vec [26] open-source.

6 EVALUATION

We evaluate *punstrip* in two ways. First, we evaluate our probabilistic fingerprint (§6.1); for this we require a dataset that has source code compiled under different optimization levels from different compilers. Second, we evaluate the combination of our probabilistic fingerprint with *punstrip*'s probabilistic structural inference (§6.2) on a large scale.

6.1 Probabilistic Fingerprint

Dataset. To evaluate our approach to inferring function names in previously unseen binaries we constructed a dataset of programs that have moderate code reuse between them. We built a corpus of binaries from *coreutils*, *moreutils*, *findutils*, *x11-utils* and *x11-xserver-utils*. This resulted in 149 unique binaries and 1,362,379 symbols. Our criteria for choosing these binaries were open source software packages containing a large number of ELF executables. All of the binaries were then compiled under all combinations of $\{\texttt{og},\texttt{o1},\texttt{o2}\} \times \{\texttt{static},\texttt{dynamic}\}$ for both *clang* and *gcc* resulting in 2132 distinct binaries[5]. We randomly split the 149 programs into 134 in the training set and 15 in the test set. All binaries with debugging information included were replicated to a stripped set of binaries before running the `strip` utility on them; this removed all symbols where possible (dynamic binaries still have dynamic symbols for linking purposes). By having two copies of the binaries, one stripped, the other with debugging information, we are able to obtain the ground truth for the results of our experiments. Previous work [2, 3] has shown that function boundaries can be identified in stripped binaries with an average F1 score of 0.95 across compiler optimizations O0-O3. Our work focuses on the problem of inferring function names only; therefore, we assume function boundaries to be given and take them from the ground truth.

Throughout all of our experiments, the same program name, and hence exactly the same source code was never in both the training and testing set. Thus 100% accuracy may be impossible as there are many functions only contained within the testing dataset; however, common pieces of source code may exist between binaries from the same package. We perform this experiment to evaluate how our probabilistic fingerprint recognizes functions compiled into different binaries and under different settings. By training a model on binary names for a given configuration of compilation options we then inferred function names for a different configuration of compilation options in the testing set. Our results can be seen in Table 6 for which we compare our fingerprint against leading industry tools IDA FLIRT and Radare 2 Zignatures. A comparison against BinDiff[6] proved impossible since it aims to perform differential comparisons between similar binaries rather than inferring function names in completely new binaries. We were unable draw a comparison against existing state-of-the-art research projects that build searchable code fingerprints such as BinGold [1] and Genius [19] because they were not fully available. We provide a larger evaluation of the entirety of *Punstrip* against the leading state-of-the-art research tool Debin [23] in §6.2 that combines program features and structural inference.

All of our experiments were run under Debian Sid with dual Intel Xeon CPU E5-2640 and 128GB of RAM. On average the computation of feature functions after training was carried out in the order of seconds. We make our full dataset available online[7].

Explanation of Results. In evaluating all schemes, we calculate Precision (P), Recall (R) and F1 score for as per Equation 8, 9, and 10. The number of true positives TP, is given by the number of correctly named functions[8]. The number of false positives FP, is given by the number of functions that were named incorrectly. The number of false negatives FN, is given by the number of functions in which we did not predict a name, but valid names existed. We define the correctness of an inferred function name to be result of our NLP matching scheme (§5) between the inferred function and the ground truth.

$$P = \frac{TP}{TP + FP} \quad (8) \qquad R = \frac{TP}{TP + FN} \quad (9)$$

$$F_1 = \frac{2 \times P \times R}{P + R} \quad (10)$$

All approaches performed worst in cross compiler, cross optimization inference on dynamic binaries. From manual analysis, there are large differences in both the structure and interactions between functions and also in the number and name of functions. For example, *clang* always produces the symbol `c_isalnum` which is never present in binaries compiled by *gcc*. It's also worth noting that in general, the number of symbols in a binary decreased with higher levels of optimization, with the *coreutils* binary *who* ranging between 80–130 functions for the dynamically linked case across optimizations *og*–*o*3 for `x86_64`. The same program compiled under *clang* with *og* produced produced 129 symbols in its `.text` section whereas under *gcc* with *og* produced 106 symbols with 35 symbols that were not shared between the two binaries.

In the cases of very low recall, Zignatures's and FLIRT's precision rises. We attribute this to domain knowledge of ELF binaries with Radare2 always finding the symbol `__libc_csu_init`, a function with a size of 0 which without structure prediction, our fingerprint does not.

6.2 Probabilistic Structural Inference

Dataset. To test if we can learn abstract relationships between arbitrary functions in the general sense it is necessary to build a large corpus of binaries with debugging information from different software packages. We construct this dataset from thousands of open source software packages from the Debian repositories.

This produced 188, 253 binaries with debugging information from 14,000 different software packages resulting in 82GB of executables; we make the tools used to build this comprehensive dataset available[9]. Of the 188, 253 ELF binaries, 17, 549 binaries were compiled from the C language. We limit ourselves to C binaries

[5] *clang og* is equivalent to *clang o1*.

[6] https://zydnamics.com/software/bindiff.html

[7] https://github.com/punstrip/cross-compile-dataset

[8] For structural inference, *punstrip* makes the assumption that *libc* initialization (e.g. `libc_csu_init`) and deinitialization (e.g. `fini`) functions can be found based on static analysis and the ELF header. This assumption was also applied when evaluating Debin.

[9] https://github.com/punstrip/debian-unstripped

Table 6: Evaluation on the accuracy of symbol inference of different corpora and the different technologies used.

Experiment	IDA FLIRT			R2 Zignatures			*Punstrip*		
	P	R	F_1	P	R	F_1	P	R	F_1
`gcc,og,dynamic -> gcc,og,dynamic`	0.94	0.38	0.47	0.51	0.72	0.60	0.99	0.85	0.91
`gcc,og,dynamic -> gcc,o1,dynamic`	0.95	0.14	0.24	0.36	0.37	0.37	0.84	0.61	0.70
`gcc,og,dynamic -> gcc,o2,dynamic`	0.30	< 0.01	< 0.01	0.29	0.04	0.07	0.52	0.37	0.44
`clang,o1,dynamic -> clang,o1,dynamic`	0.78	0.38	0.51	0.40	0.49	0.43	0.97	0.87	0.92
`clang,og,static -> clang,og,static`	0.61	0.18	0.29	0.13	0.16	0.14	0.997	0.90	0.94
`clang,og,static -> clang,o2,static`	0.60	0.17	0.27	0.12	0.14	0.13	0.98	0.87	0.92
`clang,og,static -> gcc,og,static`	0.61	0.16	0.26	0.11	0.12	0.11	0.96	0.82	0.82
`clang,og,static -> gcc,o2,static`	0.61	0.16	0.26	0.11	0.12	0.11	0.98	0.83	0.84

Table 7: 10-fold cross validation against Debin and *Punstrip*.

Metric	Debin			*Punstrip*		
	P	R	F_1	P	R	F_1
Exact	0.63	0.66	0.51	0.65	0.92	0.73
NLP	0.66	0.67	0.55	0.68	0.92	0.75
Symbol2Vec	0.68	0.69	0.57	0.70	0.93	0.77

only as we expect different relationships between functions across different languages. Using the 17,549 C binaries, we randomly split the list of executables into 10 equally sized groups and perform 10-fold cross validation to evaluate our approach.

Explanation of Results. In evaluating the performance of our probabilistic graphical model we used the same metrics as in (§6.1), however we use three different measures of correctness. The first being an exact match between the canonicalized ground truth and our inferred function name, the second being our NLP matching scheme, and lastly we consider an inferred name to be correct if its Symbol2Vec representation is within the 5 closest vectors using the cosine distance. The reason for doing so is that the sparsity of function names across our corpus gives many names that are used in similar ways whilst still not evaluating as similar in our NLP matching scheme. Secondly, giving an analyst a list of the top 5 most likely symbol names and their corresponding probabilities allows them to make an informed decision that may take into account other information about the binary in question.

Table 7 displays the results of our large scale inference experiment using 10-fold cross validation. From a detailed analysis of the results our NLP and Symbol2Vec matching schemes correctly pick up meaningful inferred function names where the exact correct name is not present in the training set. Both tools perform worst on small dynamically linked binaries with little recognizable relationships. Furthermore it is evident that *punstrip* may infer symbol names that are structurally close on a micro-level to the correct names however they lie in a different orientation; for example symbol names with strong relationships between each other are often predicted locally correct as a group but not necessarily in the correct structural order which reduces our accuracy.

We build pairwise relationships up to the third degree and store factors involving up to three functions. To improve our our tool's accuracy, we could trivially increase the dimensions of relationships stored between functions however we were limited in our computational and storage resources for our large scale experiments.

7 LIMITATIONS

Throughout our approach, we rely on previous work [2] on function boundary detection to justify the conditions for function boundaries to be known. In real world environments, further errors may be introduced in the function boundary extraction stage which could have undesired effects in probabilistic inference due to the incorrectness of the recovered graphical structure. We believe that sufficient randomization of belief updates during loopy belief propagation would minimize the impact on inference as we aim to maximize the joint likelihood across all unknown nodes simultaneously.

Our probabilistic fingerprint may succeed when faced with small changes to machine code; however there are often large unknown functions in previously unseen or obfuscated binaries. It is highly likely that *punstrip* would perform poorly on executables that are highly obfuscated or contain hand written assembly code. *Punstrip* is limited by the correctness of binary analysis; we make use of program analysis to recover features and relationships between data and code. Techniques which aim to mislead or impede program analysis are out of scope, however trivial obfuscation techniques such as junk code insertion should overcome by the VEX IR optimization step. *Punstrip* may be combined with existing reverse engineering software suites or debuggers to analyze regions of memory containing unlabeled code; the prime example being recognizing functions during software *unpacking* at runtime.

8 RELATED WORK

We examine related work across probabilistic models for computer programs (§8.1) and function fingerprinting (§8.2). Finally we look at previous work using NLP (§8.3) for matching function names.

8.1 Probabilistic Models

The seminal work of Bichsel et al. [6] in building probabilistic models is closely related to this work. They describe the process of building linear chain condition random fields for sections of Java bytecode based on a program dependency graph and utilize high-level information such as types, method operations and class inheritance to build relationships for inference. When applying a similar technique to machine code, the problem is exacerbated by the lack of access to concrete information on which to build features or known relationships to describe the semantics of code. The work was built on the Javascript deobfuscation framework, JSNice [42], which infers local variable names for Javascript programs using CRFs. Other works utilize probabilistic graphical models to infer properties of programs, e.g. specification [5] [32], verification [22] and bug finding [22]. The closest work to ours that labels functions in stripped binaries is Debin [23] which infers names of DWARF debugging information and function names simultaneously.

Recent advances in function boundary detection in stripped binary executables form a foundation of this work. We utilize Nucleus [2] when inferring function names without known function boundaries; a tool which uses spectral clustering to group basic blocks into function boundaries and results are an improvement over work by Rosenblum et al. [43] and Shin et al. [45], the former uses a CRF and the later use neural networks to detect function boundaries in binaries; however neither performs the task of function naming.

8.2 Function Fingerprinting

Work in binary function identification predominantly focuses on the problems of clone or exact function detection. Code clone detection focuses on the recognition of previously seen functions [27]. *Punstrip* infers semantically similar names for previously unseen functions based on modified known examples.

Unstrip [25] aims to identify functions in stripped binaries and focuses on labeling wrapper functions around dynamic imports. BinSlayer [7], BinGold [1] and BinShape [46] identify and label functions in stripped binaries. They collect large numbers of features such as system calls, control flow graphs and statistical properties in order to fingerprint functions. Static approaches such as Genius [19] and discovRE [16] extract features from a binary's Control Flow Graph (CFG) and rank the similarity of functions based on the graph isomorphism problem. In contrast, *punstrip* utilizes a probabilistic graphical model that uses higher-level features to infer structure in stripped binaries; combined with *Symbol2Vec* and our NLP analysis we suggest semantically similar function names. Structural Comparison of Executable Objects [20] finds vulnerabilities through analyzing security patches. Gemini [51] creates a feature embedding based on Structure2vec [12] for code clone detection.

Dynamic approaches such as BLEX [15] and Exposé [38] use symbolic execution and a theorem prover to rank the similarity between *pairs* of individual functions. Egele et al. [15] employ symbolic execution and compare dynamic traces from functions to detect similar components. Others such as BinGo [9] and Multi-MH [24, 41] try to describe a functions behavior by sampling each function with random inputs to match known vulnerabilities across architectures and operating systems. Gupta et al. [37] use a dynamic matching algorithm for comparing control flow and callgraphs.

8.3 Symbol2Vec

Independently of us, Daniel De Freez et. al. [13] implemented path-based function embeddings in a similar manner to *Symbol2Vec*. Ding et. al. [14] built a vector embedding of features from a functions assembly code in Asm2Vec and use it to perform code clone detection.

9 CONCLUSION

We have presented *punstrip*, a novel approach for naming functions in stripped executables that combines program analysis and machine learning to infer symbol information. We have demonstrated that *punstrip* is a viable approach to learn a function fingerprint that is capable of inferring symbols between multiple compilers and optimization levels. Secondly we combine our fingerprint with structure learning to predict symbol information in binaries using all known relationships simultaneously rather than considering each function in isolation. We carry out an extensive 10–fold cross validated evaluation against C ELF binaries built from different environments and compilers in the Debian Sid repositories and make this dataset available.

We explore the subjective problem of evaluating the similarity of previously unseen symbol names from different software repositories and develop both an NLP pipeline and Symbol2Vec model to aid this comparison which we release to the community. We have shown that it is possible to learn intrinsic relationships between functions and transpose that information to other *previously unseen* stripped binaries and suggest meaningful names for functions in order to aid the reverse engineering process.

ACKNOWLEDGMENTS

James Patrick-Evans was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

REFERENCES

[1] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. 2016. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Digital Investigation* 18 (2016), S11–S22.

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries.. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017.* 177–189.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to Recognize Functions in Binary Code.. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* 845–860.

[4] Eli Bendersky. 2018. PyELFTools - A Python Library for parsing ELF files. https://github.com/eliben/pyelftools.

[5] Brian N. Bershad and Jeffrey C. Mogul (Eds.). 2006. *Operating Systems Design and Implementation (OSDI).* USENIX Association.

[6] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical Deobfuscation of Android Applications.. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 343–355.

[7] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: accurate comparison of binary executables.. In *Program Protection and Reverse Engineering Workshop.* ACM, 4.

[8] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM J. Scientific Computing* 16, 5 (1995), 1190–1208.

[9] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016.* 678–689.

[10] UC Santa Barbra Computer Security Lab and Arizona State University SEFCOM. 2020. Claripy: An abstraction layer for constraint solvers. (2020). https://github.com/angr/claripy

[11] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static Detection of Vulnerabilities in x86 Executables. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*. 269–278.

[12] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data.. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2702–2711.

[13] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 423–433.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 472–489.

[15] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 303–317.

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[17] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283.

[18] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, June 30 - July 2, 2014*. 78–87.

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 480–491.

[20] Halvar Flake. 2004. Structural Comparison of Executable Objects. In *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop*, Ulrich Flegel and Michael Meier (Eds.), Vol. P-46. 161–173.

[21] Teresa Gonçalves and Paulo Quaresma. 2004. The impact of nlp techniques in the multilabel text classification problem. In *Intelligent Information Processing and Web Mining*. Springer, 424–428.

[22] Sumit Gulwani and Nebojsa Jojic. 2007. Program verification as probabilistic inference.. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 277–289.

[23] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin T. Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*.

[24] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. 104–114.

[25] Emily R. Jacobson, Nathan E. Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries.. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11*. 1–8.

[26] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Feature Embeddings for Binary Symbols with Symbol2Vec. https://www.github.com/punstrip/symbol2vec.

[27] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28, 7 (2002), 654–670.

[28] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 353–364.

[29] Daphne Koller and Nir Friedman. 2010. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press.

[30] Yujian Li and Bi Liu. 2007. A Normalized Levenshtein Distance Metric. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 6 (2007), 1091–1095.

[31] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*. 872–881.

[32] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems.. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 75–86.

[33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality.. In *Advances in Neural Information Processing Systems*. 3111–3119.

[35] George A. Miller. 1995. WordNet: A Lexical Database for English.. In *Communications of the ACM*, Vol. Vol. 38, No. 11:. 39–41.

[36] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 2013. Loopy Belief Propagation for Approximate Inference: An Empirical Study. *CoRR* abs/1301.6725 (2013).

[37] Vijayanand Nagarajan, Rajiv Gupta, Matias Madou, Xiangyu Zhang, and Bjorn De Sutter. 2007. Matching Control Flow of Program Versions.. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. 84–93.

[38] Beng Heng Ng and Atul Prakash. 2013. Expose: Discovering Potential Binary Code Re-use. In *37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013*. 492–501.

[39] Coseinc Nguyen Anh Quynh. 2014. Capstone: Next-Gen Disassembly Framework.. In *BlackHat USA*.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[41] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 709–724.

[42] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Codea.". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[43] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code.. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. 798–804.

[44] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12 (2011), 2539–2561.

[45] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks.. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 611–626.

[46] Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2017. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape.. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 301–324.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2015).

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[49] Charles A. Sutton and Andrew McCallum. 2012. An Introduction to Conditional Random Fields. *Foundations and Trends in Machine Learning* 4, 4 (2012), 267–373.

[50] Andre Nikolaevich Tikhonov, A Goncharsky, VV Stepanov, and Anatoly G Yagola. 2013. *Numerical methods for the solution of ill-posed problems*. Vol. 328. Springer Science & Business Media.

[51] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection.. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 363–376.

[52] Zynamics. 2019. Using BinDiff v1.6 for Malware analysis. (2019). https://www.zynamics.com/downloads/bindiff_malware-1.pdf