FunProbe: Probing Functions from Binary Code through Probabilistic Analysis

Anonymous Author(s)

ABSTRACT

Current function identification techniques have been mostly focused on a specific set of binaries compiled for a specific CPU architecture. While recent deep-learning-based approaches theoretically can handle binaries from different architectures, they require significant computation resources for training and inference, making their use less practical. Furthermore, due to the lack of interpretability of such models, it is fundamentally difficult to gain insight from them. Hence, in this paper, we propose FunProbe, an efficient system for identifying functions from binaries without relying on deep learning. In particular, we identify 16 architecture-neutral hints for function identification, and devise an effective method to combine them in a probabilistic framework. We evaluate our tool on a large dataset consisting of 19,872 real-world binaries compiled for six major CPU architectures. The results are promising. FunProbe shows the best accuracy compared to five state-of-the-art tools we tested, while it takes only 6 seconds on average to analyze a single binary. Notably, FunProbe is 6× faster on average in identifying functions than XDA, a state-of-the-art deep-learning tool that leverages GPU in its inference phase.

ACM Reference Format:
Anonymous Author(s). 2023. FunProbe: Probing Functions from Binary Code through Probabilistic Analysis. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Function identification is a pivotal task in binary analysis. Major decompilation techniques, such as variable recovery [5], type recovery [33], and high-level control-flow restructuring [15, 48, 49, 58], operate at a function level, assuming Control-Flow Graph (CFG) of a function is given. Binary-level control-flow integrity approaches, such as [61], adopt a function-level protection mechanism, too.

Conventional approaches to binary-level function identification employ various target-specific *heuristics*, which are often specifically devised to handle specific types of binaries, to overcome the intrinsic difficulty of binary analysis. For example, Ghidra [38] and FETCH [41] identify functions by leveraging C++ exception handling information, which does not necessarily exist in all binaries. Similarly, DDisasm [13] and FunSeeker [25] leverage target-specific syntactic patterns, and Jima [2] harvests function pointers from analyzing specific data sections. While such approaches are effective in identifying functions in certain binaries, they are not generally applicable to all binaries.

Learning-based approaches [6, 30, 56, 59] have been proposed to overcome the limitations of conventional approaches. Particularly, deep-learning-based approaches [44, 50, 60] recently demonstrate high accuracy in identifying functions. While these approaches do not require target-specific heuristics, they still suffer from three critical limitations. First, their detection performance is highly dependent on the training dataset used. Second, they require significant computation resources for both training and inference. Thus, applying them to large-scale binary analyses is not trivial in practice. Furthermore, those models are *not* interpretable, making it difficult to gain useful insight into the learned models. As such, one cannot easily understand why a particular function is misclassified, hence further improving the model is challenging.

Therefore, in this paper, we seek to develop a function identification algorithm that is *general*, *efficient*, and *interpretable*. Our approach should work for a wide range of binaries compiled for various CPU architectures; should run fast on a regular desktop machine; and should not rely on a deep-learning model that is difficult to understand.

To this end, we identify 16 architecture-neutral *hints*, and combine them to form a probabilistic model for function identification. The key intuition of our approach is that every function identification heuristic has a certain level of uncertainty, which can naturally be represented as a probability. For example, many tools heuristically regard a target of a `call` instruction as a function entry point, but call targets are not always a function entry point in reality. Such a heuristic is not always correct, but does provide a probabilistic *hint* for function identification. Hence, we combine those hints to form a Bayesian Network (BN) representing causal relationships between them.

There are mainly two challenges in designing our algorithm. First, our approach should be generally applicable to any type of binaries built with varying compilers, compiler options, and target architectures. Second, our model, i.e., a BN, naturally includes cyclic dependencies, which is expensive to handle, if not impossible [28, 29, 47, 52]. Hence, we should devise an efficient way to perform probabilistic inference on our BN.

We address the first challenge by carefully selecting architecture- and compiler-independent heuristics, such as those found in control flows between functions and basic blocks [24]. FunProbe employs 16 intuitive function identification hints obtained either from existing work or from our own observations. We also tackle the second challenge by exploiting the fact that our BN includes many unnecessary edges. Specifically, we devise a novel approach, named bogus dependency pruning, which heuristically removes edges as

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

well as loops from a BN. Our technique enables 12.6× faster function identification with negligible accuracy drop as shown in our experiments.

To realize these ideas, we implement FunProbe, a novel function identification tool that runs on raw (i.e., stripped) binaries. It leverages our bogus dependency pruning technique to quickly decide whether every instruction in the target binary should be considered to be a function entry point or not. We evaluate FunProbe on a large-scale benchmark that includes 19,872 real-world binaries compiled for six major CPU architectures. Our evaluation confirms that FunProbe shows the best accuracy compared to the five state-of-the-art binary analysis tools, including IDA Pro, Ghidra, and XDA, while it takes only 6 seconds on average to analyze a single binary. Notably, FunProbe is 6× faster than XDA, a state-of-the-art deep-learning tool, without the need for GPU resources. We also demonstrate that FunProbe is complementary to XDA. By simply feeding the output of XDA into FunProbe, we were able to achieve 99.9% of F1-score on our benchmark.

Overall, this paper makes the following contributions.

- We present a novel function identification algorithm that leverages a BN.
- We propose bogus dependency pruning to efficiently derive solutions from a BN.
- We design and implement FunProbe that incorporates our novel function identification algorithm.
- We publicize our tool to support open science: https://github.com/anonymized.

2 BACKGROUND AND MOTIVATION

This section describes the basic concept of Bayesian Network (BN) and belief propagation, and motivates our research.

Notation. In this paper, $P(X)$ denotes the probability distribution of a Boolean random variable X. For simplicity, we let $P(x)$ be the probability of X being true, i.e., $P(x) = P(X = 1)$.

2.1 Bayesian Network & Belief Propagation

Bayesian Network (BN) is a set of Directed Acyclic Graphs (DAGs), each of which represents causal dependencies between random variables [42]. Each node in a BN is a random variable, and each directed edge represents a dependency between two random variables. For example, an edge $X \longrightarrow Y$ means that X causes Y (or Y is dependent on X).

We use a BN to model the relationships between probabilistic statements, such as "the probability of an address being a function entry point". Consider a basic block located at address α. Let F_α be a Boolean random variable indicating whether or not α is a function entry point, and let C_α be a Boolean random variable denoting whether or not α is a target of a `call` instruction. We can then represent a causal relationship between F_α and C_α with an edge $C_\alpha \longrightarrow F_\alpha$ in a BN: If C_α is true (there is a `call` instruction whose target is α), then F_α is likely to be true (α is a function entry point). Such a relationship can be denoted by a conditional probability $P(f_\alpha \mid c_\alpha)$, and we call it a **hint** throughout this paper because it provides a *hint* for identifying functions.

Notice not every random variable in a BN is observable. In the above example, F_α is a *hidden* random variable, which cannot be

```
0x3bb0 <main>:
  0x3bb0:   push  r15
  0x3bb2:   push  r14
  0x3bb4:   mov   r14d, edi
  0x3bb7:   push  r13
  0x3bb9:   push  r12
...
  0x4944:   mov   edi,  0xd
  0x4949:   call  58b0  <is_colored>
...
0x5830 <dev_ino_free>:
  0x5830:   jmp   1b770 <rpl_free>
...
0x58b0 <is_colored>:
  0x58b0:   mov   edi,  edi
  0x58b2:   lea   rax,  [rip+0x2217a7]
  0x58b9:   shl   rdi,  0x4
  0x58bd:   add   rdi,  rax
  0x58c0:   xor   eax,  eax
  0x58c2:   mov   rdx,  QWORD PTR [rdi]
  0x58c5:   test  rdx,  rdx
  0x58c8:   je    58df
...
```

(a) Disassembled x86-64 binary code.

```
0000f4 000010 0000c8 FDE cie=000030 pc=000058b0..00005909
  DW_CFA_nop
...
```

(b) Frame Description Entry (FDE) for `is_colored`.

Figure 1: Our motivating example taken from `dir`, a GNU Coreutils binary compiled with GCC.

directly observed from analyzing a binary. On the other hand, C_α can be easily observed: We can disassemble a binary and see if there is a `call` instruction whose target is α. Therefore, our goal in this paper is to compute the marginal probabilities of hidden random variables, e.g., $P(f_\alpha)$, based on the probability distributions of the observed variables. This process is often referred to as *probabilistic inference* [10], which does *not* scale well with the number of random variables—the number of terms to consider for marginalization grows exponentially to the number of hidden variables.

Belief propagation [42] is a technique that addresses the scalability challenge with a dynamic programming method. When a BN contains a loop(s), however, belief propagation cannot compute the exact solution [37]. Thus, *loopy belief propagation* is used as an alternative [21, 28, 37, 47], which iteratively runs belief propagation until it converges or reaches a fixed time limit. However, when a graph is large and highly connected, (loopy) belief propagation is still computationally expensive [12, 21]. In this paper, we propose an efficient way to perform belief propagation on a large-size cyclic BN by reducing the graph size (see §4.3).

2.2 Motivation

While *hints* are useful to identify functions, they do *not* provide a definite answer. In this section, we motivate the need for a probabilistic method by illustrating how one can combine various hints collected from a binary to make a precise decision. To support our claim, we ran Nucleus on `dir`, a binary taken from GNU Coreutils, to identify functions from it. Figure 1a shows the disassembled code snippets of the binary. To ease the explanation, we explicitly mark function entry points in Figure 1a with symbols, although we used a stripped binary when we ran Nucleus.

In this example, Nucleus identifies functions using two hints: (1) the target of a `call` instruction is likely to be a function entry

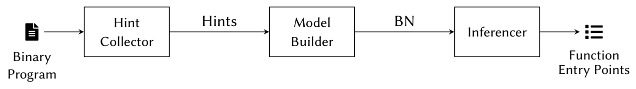


Figure 2: FunProbe architecture.

point, and (2) a no-op-like instruction is unlikely to be a function entry point (as it is often a padding sequence). Unfortunately, both hints suffer from a certain level of uncertainty. The first hint seems legitimate as `call` instructions are designed to call a subroutine in a program. However, there are cases where a `call` instruction is used to retrieve the current Program Counter (PC). For example, a code pattern “`call +5; pop ebx`”, where the target of `call +5` is the `pop` instruction, is often used to obtain the current PC. Similarly, the latter hint is not always correct: Compilers sometimes emit a no-op-like instruction at a function entry point.

The primary challenge here is that there are cases where two hints are in conflict. Conventional approaches, such as Nucleus, would simply follow one of them. In our example, the function at `0x58b0` is a target of the `call` instruction at `0x4949`. Thus, Nucleus can correctly identify the function with the first hint. However, the function begins with a no-op instruction, which is deemed as an invalid instruction by Nucleus based upon the second hint. Note that `mov edi, edi` does not change the CPU state except for the program counter, and is often used as a padding byte. The second hint makes Nucleus disregard the function, thereby causing a false negative error.

FunProbe handles such an intrinsic challenge with a probabilistic framework. Intuitively, each different *hint* provides a different *clue* about identifying function entry points, and we can represent their relationships with a BN. If we collectively consider the observed hints in the BN, we can make a decision in a more *systematic* and *holistic* way.

For example, let us consider an additional heuristic developed by FETCH [41], which leverages exception handling information stored in the `.eh_frame` section of ELF binaries. The section stores a sequence of Frame Description Entries (FDEs), each of which corresponds to a consecutive code chunk in the binary. Typically, each code chunk represents a function (although there are exceptional cases). Therefore, in our example, the FDE shown in Figure 1b indicates that there is a code chunk located at `0x58b0`, and it is likely that the address indicates a function entry point. With such an additional hint, we can say that `0x58b0` is more likely to be a function entry point (two positives vs. one negative). As we will discuss, FunProbe provides a way to systematically make an informed decision by gathering all the observed hints.

3 OVERVIEW

In this section, we first present the overall architecture of FunProbe, and describe its workflow with a running example shown in Figure 1. We then discuss several technical challenges in designing FunProbe.

3.1 FunProbe Architecture

Figure 2 illustrates the overall architecture of FunProbe, which takes in a binary as input, and produces a set of function entry points as output. FunProbe consists of three components: (1) Hint Collector, (2) Model Builder, and (3) Inferencer. Hint Collector harvests hints by analyzing the given binary. Model Builder constructs a BN using the collected hints. Inferencer runs belief propagation on the BN to infer the marginal probabilities for each address in the binary can be a function entry point.

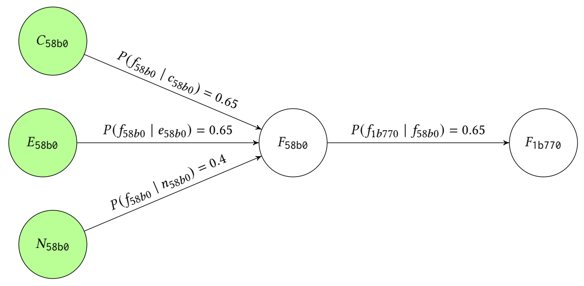

C_{58b0}
E_{58b0}
N_{58b0}
F_{58b0}
F_{1b770}
$P(f_{58b0} \mid c_{58b0}) = 0.65$
$P(f_{58b0} \mid e_{58b0}) = 0.65$
$P(f_{58b0} \mid n_{58b0}) = 0.4$
$P(f_{1b770} \mid f_{58b0}) = 0.65$

Figure 3: Illustration of the BN generated from our example.

We now demonstrate the component-wise workflow of FunProbe using the example binary shown in Figure 1. Note our system runs on a stripped version of the binary.

3.2 Hint Collector

First, Hint Collector takes in a binary as input, and outputs a set of hints taken from the binary. Formally, a *hint* is a conditional probability for an address being a function entry point. Let F_α be a hidden Boolean random variable indicating whether or not α is a function entry point. We can then define a *hint* as a conditional probability of the form $P(f_\alpha \mid x)$ where X is a Boolean random variable and x is a shortcut for $X = 1$ as we defined in §2. Hint Collector analyzes the given binary to construct an inter-procedural CFG (§4.1), and, it collects 16 different types of hints from the CFG as well as the metadata stored in the binary (§4.2).

From our example binary shown in Figure 1, Hint Collector outputs a variety of hints including the following four: $P(f_{58b0} \mid c_{58b0})$, $P(f_{58b0} \mid e_{58b0})$, $P(f_{58b0} \mid n_{58b0})$, and $P(f_{1b770} \mid f_{58b0})$, where C_{58b0} indicates that `0x58b0` is a call target, E_{58b0} means that there is an FDE for the instruction located at `0x58b0`, and N_{58b0} shows that the instruction located at `0x58b0` is semantically a no-op instruction; It does not change any CPU state except for the PC.

Note that two *hidden* random variables, e.g., F_{1b770} and F_{58b0}, can also have a causal relationship. Let us consider the tail-call heuristic used by many tools [11, 38, 41, 45], which regards a jump as a tail-call when it crosses over a function entry point. This strategy is based on the observation that compilers usually emit a function body in a consecutive region. Therefore, if a jump instruction passes over another function entry point, it means the jump target is likely to belong to another function. In our example binary, there is a jump instruction located at `0x5830`, and there is another function entry point at `0x58b0` in between the jump instruction and its target (`0x1b770`). Therefore, if there is a function entry point between `0x5830` and `0x1b770`, then `0x1b770` is also likely to be a function entry point.

This observation provides at most 89,919 distinct hints ($P(f_{1b770} \mid f_{5831}), P(f_{1b770} \mid f_{5832}), \cdots, P(f_{1b770} \mid f_{1b76f})$), which can easily bloat up the BN, rendering our probabilistic inference inefficient.

Thus, we reduce the number with a novel technique, named bogus dependency pruning (§4.3).

3.3 Model Builder

Model Builder first builds a BN from the observed hints. Specifically, each hint forms an edge in the resulting BN. Figure 3 illustrates the BN obtained from our example binary. The BN only shows four hints (i.e., edges) for simplicity.

Next, Model Builder assigns probabilities for each edge based on the characteristic of each corresponding hint. Specifically, we use two user-configurable probability values: one for positive hints and another for negative hints. We say a hint is positive if it increases the likelihood of the corresponding address being a function entry point, and negative if otherwise. For example, $P(f_{58b0} \mid c_{58b0})$ increases the likelihood of `0x58b0` being a function entry point because it is used as a call target, thus, it is a positive hint. On the contrary, $P(f_{58b0} \mid n_{58b0})$ decreases the likelihood of `0x58b0` being a function entry point because there is a no-op instruction at `0x58b0`, thus, it is a negative hint.

Obviously, positive hints, such as $P(f_{58b0} \mid c_{58b0})$, should have a high value (close to 1), and negative hints, such as $P(f_{58b0} \mid n_{58b0})$, should have a low value (close to 0) in order for them to respectively give a positive and negative influence to the probability of `0x58b0` being a function entry point. We introduce two user-configurable parameters $\mathcal{P}^+$ and $\mathcal{P}^-$ in order to assign the probability values for positive and negative hints. In our current implementation, we use $\mathcal{P}^+ = 0.65$ and $\mathcal{P}^- = 0.4$ by default, which are empirically chosen best parameter values (see §5.2). Thus, in the previous example, we assign 0.4 to $P(f_{58b0} \mid n_{58b0})$, and 0.65 to the other edges. The marginal probabilities, i.e., $P(f_{58b0})$ and $P(f_{1b770})$, are then computed in the next step based on the assigned probabilities.

3.4 Inferencer

Inferencer takes in as input the initialized BN, and performs belief propagation on the BN to compute the marginal probabilities. That is, our goal here is to obtain $P(F_{58b0})$ and $P(F_{1b770})$ shown in Figure 3. Once we have the marginal probabilities, we can decide whether each address is a function entry point or not. In our current implementation, we say an address α is a function entry point if $P(f_\alpha) > 0.5$. In our example, using belief propagation, Inferencer returns $P(f_{58b0}) = 0.9216$ and $P(f_{1b770}) = 0.9999$. Since these probabilities are higher than our threshold 0.5, we regard both of them as a function entry point.

Recall from §3.2 that a tail-call detection strategy can easily produce thousands of hints, which makes the resulting BN complex and highly connected with loops. Therefore, Inferencer does not scale well with large binaries with many functions. We overcome this challenge by introducing a novel technique, named *bogus dependency pruning*, as we detail in §4.3.

4 FUNPROBE DESIGN

This section details the design of FunProbe. We first show how FunProbe prepares an inter-procedural CFG to extract hints from the binary (§4.1). Next, we present all the function identification hints that FunProbe utilizes and discuss how we make them general (§4.2). Finally, we present bogus dependency pruning, a technique to simplify BNs to make probabilistic inference efficient (§4.3).

4.1 Control Flow Graph Construction

To collect hints from binaries in architecture- and compiler-independent manner, we mainly focus on structural features that can be obtained from a CFG instead of observing instruction patterns. One can leverage existing binary analysis frameworks, such as Ghidra and angr, to obtain CFGs from a binary, but those tools typically involve heavy-cost analyses. Thus, we perform our own lightweight analysis to quickly recover CFGs so that we can apply our function identification strategies to them. Note that our CFG recovery is a preprocessing step for function identification, and our goal is far from constructing precise CFGs. Our analysis runs in the following four steps.

(1) Linearly disassemble the given binary (§4.1.1).
(2) Detect non-returning function calls (§4.1.2).
(3) Resolve indirect branch targets (§4.1.3).
(4) Build inter-procedural CFG (§4.1.4).

4.1.1 Linear-Sweep Disassembly. Hint Collector first linearly disassembles instructions from a given binary. One could use superset disassembly [7] to completely recover all possible instructions in the binary, but linear-sweep disassembly is widely known to be efficient in achieving high instruction coverage without many false positives [3]. Therefore, we choose a simpler technique for our implementation.

One challenge here is to avoid disassembling embedded data in code sections, which are commonly found in `ARMv7` binaries. To distinguish code and data, we examine every memory load from the disassembled instructions, and filter out instructions that are referenced from another instruction.

4.1.2 Non-Returning Function Call Detection. Not every function call returns. Thus, we cannot simply connect a CFG edge from a `call` instruction to its fall-through instruction. Having a bogus edge can be problematic especially when a new function starts immediately after a non-returning function call. For example, a function may start right after a `call` to `exit`, because `exit` will never return. Thus, it is imperative to identify non-returning function calls to obtain precise CFGs.

Hint Collector employs a widely used mechanism for detecting non-returning functions, which simply finds calls to a known non-returning function name [34]. This is possible because even a stripped binary maintains the symbols of imported functions. In our implementation, we use the same list of non-returning function names used by Ghidra [38].

4.1.3 Indirect Branch Resolution. It is crucial to resolve indirect branch targets to improve the coverage of a CFG. For example, when an edge from an indirect branch to a basic block is missing, one may falsely identify the missing block as a function. To recover the targets of indirect jumps, we leverage a pattern-based heuristic used by Ghidra [38]. Specifically, we find the corresponding jump table based on the instruction patterns near the indirect jump instruction, and parse the jump table to recover the indirect jump targets.

4.1.4 Inter-procedural CFG Building. Given a list of disassembled instructions (§4.1.1), a set of non-returning function call sites (§4.1.2), and a set of jump targets for each indirect branch (§4.1.3), Hint Collector finally builds an inter-procedural CFG. Additionally, we parse exception handling information and connect edges from a `try` block to its corresponding `catch` block(s).

4.2 Function Identification Hints

FunProbe gathers 16 kinds of hints listed in Table 1. The first column shows the information source. The second column describes each hint. The third column shows which random variable is used to represent each hint. For example, a random variable X_α for an address α can represent a hint $P(f_\alpha \mid x_\alpha)$. The fourth column indicates how each hint influences function identification: whether it is a positive hint (➕) or a negative hint (➖). And the last column specifies which tool is using the corresponding heuristic. We mark with "All" when it is used by every tool that we studied, and "New" when we are unaware of a tool that employs a similar heuristic.

We note that all these hints rely on architecture-neutral metadata and CFG-structural features. Such a design choice makes FunProbe perform well across various binaries obtained from different architectures and compilers as we will show through our experiments.

4.2.1 Hints from Data. Hints ① and ② are derived from ELF metadata. First, ELF binaries have special sections that contain function pointers, such as `.init_array` and `.fini_array`. ① states that those values are always function addresses, i.e., $P(f_\alpha \mid p_\alpha) = 1.0$. Second, there are function symbols that remain intact even after stripping, e.g., GOT-based indirect jump targets in `MIPS`. ② suggests collecting such function addresses by analyzing symbols. Both ① and ② provide definite evidences for a function located at α, so we assign the probability 1.0, i.e., $P(f_\alpha \mid \cdot) = 1.0$. For other hints, we assign probability values based on the parameters $\mathcal{P}^+$ and $\mathcal{P}^-$.

Hint ③ provides a positive prediction for a code address α if it is pointed to by a relocation entry: $P(f_\alpha \mid r_\alpha) = \mathcal{P}^+$. Relocation entries often contain function pointers. For example, when a PIE (Position Independent Executable) has a global function pointer, it should be relocated at runtime by the loader. Hence, a relocation section, e.g., `.rela.dyn`, of the binary should store a relocation entry for the function pointer.

Hint ④ states that addresses found in `.eh_frame` are likely to be a function address. GCC has recently started to provide exception handling information for every C function in order to support C++ interoperability [19, 20]. Particularly, the `.eh_frame` section contains a list of Frame Description Entries (FDEs) to support stack unwinding when an exception occurs, and such information allows us to infer function addresses [41].

Hint ⑤ suggests that a data value is likely to be a function address if it is within a valid code address range. This heuristic is employed by several reassemblers, such as Ramblr [55] and DDisasm [13]. The intuition is that a constant value in a data section is likely to be a function pointer if it is within a valid address range.

4.2.2 Hints from Code. Hint ⑥ says that a call target is likely to be a function entry point. Hint ⑦ states that a jump target is likely to be a tail-call, i.e., the target is a function entry point, if the jump crosses over a function entry point. Both hints are discussed in §3.2.

Hint ⑧ handles a special case where a function body solely consists of a single (unconditional) jump instruction (see Line 11–12 of Figure 1a). Such functions usually act as a trampoline, passing arguments to another function.

Hint ⑨ helps discover functions from unreachable parts of our CFG. Since our CFG reconstruction (§4.1) is incomplete, it may leave several basic blocks within the range of a function unreachable. Such a code chunk is often referred to as a gap [31], and this hint says that every basic block found in a gap is likely to be a function entry point.

However, not every unreachable basic block is a function entry point, and ⑨ alone can produce many false positives. Thus, we have to employ several negative hints to filter out false cases. Hint ⑩ says that gaps surrounded by connected basic blocks in our CFG are unlikely to be a function entry point. This is because a function body is usually a consecutive chunk of code. Thus, such a gap is unlikely to be another function. Compiler-generated padding bytes may form a gap, too. Thus, hint ⑪ helps disregard padding bytes from being considered a function entry point. Compilers can also use jump instructions to fill a gap. For example, consider a code pattern "`ret; jmp LBL; nop; LBL: push ebp`". In this example, the `ret` instruction is the end of a function, and `push ebp` is the start of the next function. The `jmp` instruction between these two functions is unreachable, and it merely acts as a dummy padding. ⑫ helps detect such a pattern to disregard the gap from being considered as a function entry point.

Recall from §2.2, there are cases where a `call` instruction is used to retrieve the current PC. Hint ⑬ helps prevent a call target of a PC-getter from being considered as a function entry point.

Hint ⑭ states that jump targets of a conditional jump are unlikely to be a function entry point because a conditional jump is rarely a tail-call [40].

Hint ⑮ suggests that basic blocks before the main entry point, i.e., `_start`, are unlikely to be a function entry point. This is because GCC often puts rarely executed parts of a function, named with the suffix `.cold`, before the main entry point.

Hint ⑯ says that while a leader of a basic block is a potential function entry point, most leaders are not. This is the nature of a CFG; only the root node is a function entry, and the rest are not.

4.2.3 Generality of Function Identification Hints. We claim that our function identification hints in Table 1 are general enough to analyze various kinds of binaries. Hint ①–④ use metadata commonly found in ELF binaries. Although our current implementation handles only ELF binaries, other file formats provide similar information, and supporting them should be straightforward. Hint ⑤ exploits a general characteristic of function pointers found in binaries. Hint ⑥–⑯, except ⑪ and ⑫, are based on structural properties found in CFGs. ⑪ and ⑫ are based on semantic properties of instructions, which can be captured by lifted Intermediate Represent (IR) [26]. Therefore, our function identification hints are generally applicable to various different binaries.

4.3 Bogus Dependency Pruning

Although FunProbe employs only 16 hints, they can produce an extremely large BN with many loops, which makes traditional belief propagation significantly slow, if not impossible. In our dataset,

Table 1: Function Identifcation Hints that FunProbe used.

	Function Identification Hints	R. V.	Influence	Used By
Data	① Pointers in known pointer-array sections (e.g., `.init_array`) refer to a function entry point.	P_α	+	[13]
	② Non-strippable function symbols point to a function entry point.	D_α	+	[38, 51]
	③ Relocation entries can specify a function entry point.	R_α	+	[9, 13]
	④ An FDE (Frame Description Entry) can point to a function entry point.	E_α	+	[2, 13, 25, 38, 41, 45]
	⑤ Pointer-like values in data sections (e.g., `.data`) can refer to a function entry point.	V_α	+	[13, 55]
Code	⑥ Call targets can be a function entry point.	C_α	+	All
	⑦ When a jump edge crosses a function entry point, then the jump is likely to be a tail-call.	F_α	+	[11, 38, 41, 45]
	⑧ If a call target is a jump instruction, then the jump target is likely to be a function entry point.	W_α	+	New
	⑨ Unreachable basic blocks are likely to be a function.	U_α	+	[2, 4, 13, 45, 51]
	⑩ An unreachable basic block surrounded by two connected basic blocks is unlikely to be a function.	S_α	−	New
	⑪ Padding sequences, such as no-op, are unlikely to be a function entry point.	N_α	−	[4, 13, 45]
	⑫ If there are only no-op instructions between a jump instruction and its target then the jump is likely to be a no-op.	I_α	−	New
	⑬ Inlined PC-getters are unlikely to be a function.	G_α	−	[2, 13, 45]
	⑭ Conditional jump targets are unlikely to be a function entry point.	J_α	−	[38, 51]
	⑮ Basic blocks before the main entry point of a binary are unlikely to be a function.	A_α	−	New
	⑯ Basic block leaders are unlikely to be a function entry point.	B_α	−	All

`R. V.` stands for Random Variable.

Algorithm 1: Bogus Dependency Pruning Algorithm.

```
1 function BogusDenpendencyPruning($\mathcal{G}$)
2     $\mathcal{G}', \mathcal{V}, \mathcal{E}$ ← RemoveObservedNodes($\mathcal{G}$)
3     $\mathcal{G}_t$ ← ComputePolytree($\mathcal{G}'$)
4     $\mathcal{G}'_t$ ← RestoreObservedNodes($\mathcal{G}_t, \mathcal{V}, \mathcal{E}$)
5     return $\mathcal{G}'_t$
```

FunProbe builds a BN of 46K nodes and 255K edges per binary on average. Therefore, we devise a novel approach to reduce the size of a BN, while not sacrificing much the accuracy.

We note that hint ⑦ is the major source of complexity. First, it produces too many bogus dependencies as we discussed in §3.2. Formally, we say a hint $P(f_\alpha \mid f_\beta)$ is *bogus* if α or β is not a function entry point. From our study, we found that about 92% of dependencies introduced by ⑦ were indeed bogus dependencies. Second, ⑦ can create cyclic dependencies, i.e., loops, in the resulting BN. For example, three hints $F_\alpha \longrightarrow F_\beta$, $F_\beta \longrightarrow F_\gamma$, and $F_\gamma \longrightarrow F_\alpha$ form a loop, which prevents us from using the traditional belief propagation.

Therefore, we propose a novel method, named bogus dependency pruning, to find out and exclude such bogus dependencies as well as loops, so that the resulting graph becomes simple and loop-free, allowing us to use the traditional belief propagation. Algorithm 1 describes the overall process of bogus dependency pruning. It takes in as input a BN ($\mathcal{G}$), which is a set of DAGs, and returns a modified BN ($\mathcal{G}'_t$), which is a set of polytrees, as output. In Lines 2 and 4, observed nodes in the given BN are temporarily removed ($\mathcal{G}'$) and restored ($\mathcal{G}'_t$) in order to make the polytree calculation efficient (§4.3.1). In Line 3, we reduce each DAG in the modified BN to a polytree (§4.3.2).

4.3.1 Removing and Restoring Nodes. `RemoveObservedNodes` and `RestoreObservedNodes` are respectively preprocessing and post-processing steps of our dependency pruning process (§4.3.2). When `RemoveObservedNodes` temporarily removes all the observed nodes and their outgoing edges, removed nodes ($\mathcal{V}$) and edges ($\mathcal{E}$) are returned and they are restored in `RestoreObservedNodes`. Note that our polytree computation (§4.3.2) is not affected by removing observed nodes because each of those nodes has a degree of one. Therefore, the removed nodes and edges will always be included in the resulting polytree anyways. By temporarily removing all the nodes of degree one, we can make our polytree computation fast.

4.3.2 Computing Polytree. Given the modified BN $\mathcal{G}'$, a function `ComputePolytree` transforms each DAG in $\mathcal{G}'$ into a polytree. The transformation runs in three steps. First, we convert a DAG into an undirected graph. Second, we run Kruskal's minimum spanning tree algorithm [32] using our custom weight that prefers edges with more positive hints. Formally, we define a weight $w(\alpha, \beta)$ for an edge $F_\alpha \longrightarrow F_\beta$ as

$$w(\alpha, \beta) = -\left(\sum_{i=1}^{m} \mathtt{W}(X^i_\alpha) + \sum_{i=1}^{n} \mathtt{W}(X^i_\beta)\right),$$

where X^i_α means the i-th observed Boolean random variable with regard to the address α, and m and n are the numbers of observed Boolean random variables for α and β, respectively. The function `W` outputs 1 when the given random variable represents a positive hint (+) and -1 for a negative hint (−), respectively.

For example, consider the edge $F_{58b0} \longrightarrow F_{1b770}$ shown in Figure 3. We can compute the weight of the edge as below.

$$\begin{aligned} w(58b0, 1b770) &= -\left(\mathtt{W}(C_{58b0}) + \mathtt{W}(E_{58b0}) + \mathtt{W}(N_{58b0})\right) \\ &= -\left(1 + 1 - 1\right) \\ &= -1. \end{aligned}$$

This way, we can prefer edges that provide more positive influence in detecting functions. Notice that we negatize the sum as Kruskal's algorithm prefers a smaller weight.

4.4 Implementation

FunProbe is written in 5.5K SLoC of F#. To parse ELF file headers and disassemble instructions for various architectures, we used B2R2 [22], which provides an efficient binary analysis front-end.

5 EVALUATION

In this section, we evaluate FunProbe to answer the following research questions.

RQ1. How do the probability parameters affect the effectiveness of function identification? (§5.2)

RQ2. How much performance gain does bogus dependency pruning provide? (§5.3)

RQ3. How well does FunProbe perform against the conventional function identification tools? (§5.4)

RQ4. How well does FunProbe perform against the learning-based function identification tools? (§5.5)

RQ5. Can FunProbe and learning-based approaches be complementary to each other? (§5.6)

5.1 Evaluation Setup

5.1.1 Benchmark. We build our benchmark by compiling three popular packages, GNU Coreutils (v9.0, 107 programs), GNU Binutils (v2.37, 15 programs), and SPEC CPU2017 (16 programs) written in C and C++ languages. We consider the following compiler configurations to build our benchmark: (1) target architectures, (2) position independence, and (3) code optimization levels. These configurations can largely affect the shape of resulting binaries, thereby impacting the function identification results.

Specifically, we chose 6 popular CPU architectures (`x86`, `x86-64`, `ARMv7`, `AArch64`, `MIPS`, and `MIPS64`), 2 major compilers (GCC v8.4.0 and Clang v13.0.1), both PIE and non-PIE options, and six different compiler optimization levels (O0, O1, O2, O3, Os, and Ofast). This gives us a total of 144 (= $6 \times 2 \times 2 \times 6$) different configurations. As a result, our benchmark consists of **19,872** binaries. To the best of our knowledge, this is the *largest benchmark* used to evaluate function identification algorithms [30].

5.1.2 Ground Truth. To evaluate function identification tools, we need to obtain the ground truth of our benchmark, i.e., a set of function entry points for each binary. Specifically, our ground truth data deal with functions located in the `.text` section. We mainly leveraged debugging information to gather ground truth data. Also, we manually filtered out several function symbols with `.cold` or `.part` suffixes from GCC-compiled binaries [25]. Additionally, we manually added the addresses of compiler intrinsic functions to our ground truth data as they do not have debugging information.

5.1.3 Comparison Target. We selected five state-of-the-art tools for comparison. Four of them use conventional binary analyses: IDA Pro (v7.7.220118), Binary Ninja (v3.0.3233), Ghidra (v10.1.5) [38], and Nucleus (commit `e3ab49d`) [4]. One of them uses a deep-learning model to identify functions: XDA (commit `068007c`) [44]. We fine-tuned XDA on a subset of our benchmark following the recommendation of the authors [43]. Specifically, we randomly selected 20% (1296 binaries) of `x86` and `x86-64` binaries in our benchmark to create about 640k training byte sequences. XDA is a representative ML-based solution that is publicly available. To our knowledge, it had demonstrated the best function identification accuracy so far in the literature. While there is another noteworthy tool named DeepDi [60], it is closed-source, hence, we were not able to train a model using our benchmark.

5.1.4 Our Environment. We used a server machine with 88 Intel Xeon E5 cores, 512 GB of RAM, and 8 TITAN Xp GPUs to run our experiments except for fine-tuning XDA (as discussed in §5.5). We used Docker 20.10.3 for running comparison targets. We used Ubuntu 20.04 containers for FunProbe, Binary Ninja, and Ghidra, and a Ubuntu 16.04 container for Nucleus as the authors suggested. Lastly, we used a Windows 10 VM, but not a Docker container, for IDA Pro due to the license issue.

Table 2: F1-scores achieved with different parameter values.

		$\mathcal{P}^+$								
		0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95
$\mathcal{P}^-$	0.05	23.99	23.98	23.97	23.97	44.23	45.02	78.00	78.43	78.56
	0.1	23.99	23.99	24.00	44.50	48.01	78.23	78.47	96.10	96.11
	0.15	23.99	24.00	44.06	45.08	78.25	78.50	78.58	96.10	96.19
	0.2	24.04	24.02	44.89	78.22	78.50	96.13	96.12	96.17	82.19
	0.25	24.05	44.46	78.04	78.45	78.62	96.14	96.19	82.22	82.21
	0.3	24.06	45.06	78.44	78.63	96.17	96.20	82.23	82.21	79.45
	0.35	24.08	78.31	78.63	96.18	96.22	82.23	82.22	79.47	79.19
	0.4	45.10	78.65	96.23	82.23	82.27	79.51	79.37	79.06	78.80
	0.45	96.17	82.27	79.54	79.12	78.91	78.73	78.60	78.40	78.18

5.2 Probability Parameter Selection

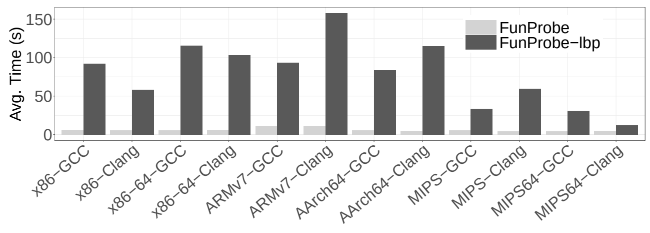
Recall from §3.3, FunProbe provides two user-configurable parameters $\mathcal{P}^+$ and $\mathcal{P}^-$ to assign probabilities for positive and negative hints, respectively. How do these parameters affect the accuracy of FunProbe? Which values should we use to maximize the accuracy? To answer these questions, we ran FunProbe on a subset of our benchmark with varying parameter values, and measured the accuracy (F1-score) for each setting.

We picked 10 random binaries from GNU Coreutils for each build configuration (out of 144 build configurations with varying compilers, architectures, and compiler options). This gives us a total of 1,440 binaries corresponding to 7% of the entire benchmark. We then ran FunProbe on these binaries with 81 different combinations of $\mathcal{P}^+$ and $\mathcal{P}^-$ values.

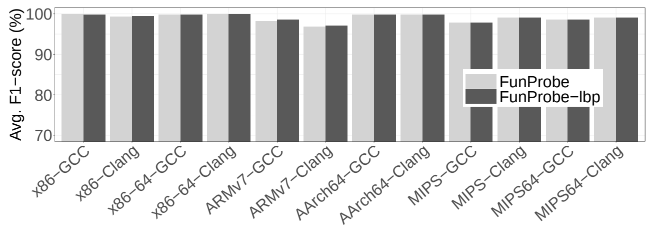
Table 2 presents the measured F1-scores for each parameter combination. The X-axis shows $\mathcal{P}^+$ values, and the Y-axis shows the $\mathcal{P}^-$ values. Each cell is filled with a grey-scale color, where the darker color indicates a higher F1-score.

The results show that $\mathcal{P}^+$ and $\mathcal{P}^-$ are negatively correlated. The bigger $\mathcal{P}^+$ value is chosen, the smaller $\mathcal{P}^-$ value needs to be selected to make FunProbe perform well. Therefore, the diagonal part of the table shows higher accuracy results. In addition, F1-scores shown in the table indicate that the performance of FunProbe is not too sensitive to the choice of parameters.

Since the highest value was achieved by ($\mathcal{P}^+ = 0.65$, $\mathcal{P}^- = 0.4$), we chose them as our default parameter values. Our experimental results on the entire benchmark (see §5.4) also show that our choice of default parameters is adequate. We leave it as future work to find a better combination of parameters with more fine-grained values.



(a) Average execution time per subset of benchmarks.



(b) Average F1-score per subset of benchmarks.

Figure 4: Comparison between FunProbe and FunProbe-lbp in terms of both execution time and accuracy.

5.3 Impact of Bogus Dependency Pruning

Recall from §4.3, bogus dependency pruning simplifies the given BN to be *loop-free*, which enables us to use traditional belief propagation. To measure the impact of bogus dependency pruning, we modified FunProbe to run *loopy* belief propagation without bogus dependency pruning, which is dubbed "FunProbe-lbp", and compared its performance against it of the original FunProbe.

Although loopy belief propagation is designed to handle BNs with loops, the solution may not converge in some cases. Therefore, it typically runs the algorithm only for a fixed number of iterations. In our implementation of FunProbe-lbp, we use the following convergence criteria for loopy belief propagation for maximum 10 iterations: For all basic block address α in the given binary, if $|P_{i-1}(f_\alpha) - P_i(f_\alpha)| < 0.0001$ holds, then we consider the algorithm is converged, where $P_i(f_\alpha)$ is a marginal probability obtained after i iterations of the algorithm. The algorithm stops when $i > 10$.

We ran FunProbe and FunProbe-lbp on the same benchmark shown in §5.1.1 using Docker containers assigned with one CPU core. Each binary in the benchmark was analyzed for maximum 12 hours. Figure 4 reports both their running time as well as F1-score for 12 different architecture-compiler combinations. Each bar in Figure 4a represents the average running time in seconds, and each bar in Figure 4b represents the average F1-score in percentage.

Overall, FunProbe was 12.6× faster than FunProbe-lbp. The graph shows the computation cost of FunProbe is almost negligible. Moreover, FunProbe-lbp failed to run 32 binaries within 12 hours of timeout. On the contrary, FunProbe was able to analyze all the binaries in our benchmark without any timeout. The average F1-scores were nearly the same for both. The total average difference was only 0.07%: FunProbe and FunProbe-lbp recorded 98.99% and 99.06% of F1-score, respectively. These results confirm the impact of bogus dependency pruning: it makes FunProbe scalable while not sacrificing much of the accuracy.

5.4 Comparison with Conventional Tools

To see how well FunProbe performs compared to conventional approaches, we selected four state-of-the-art tools: Binary Ninja, IDA Pro, Ghidra, and Nucleus. We ran each tool on our entire benchmark (§5.1.1). We set a timeout of one hour for each binary. When reporting the final accuracy results, we excluded those binaries that did not meet the timeout requirement.

Table 3 shows the precision (P), recall (R), F1-score (F1), execution time (ET), and the number of binaries failed to analyze due to the timeout (TO). Each row summarizes the results for each architecture and compiler combination. A dash (-) mark indicates that the tool does not support the architecture.

Overall, FunProbe achieves the *best performance* for all the criteria. On average, FunProbe showed 99.33% precision, 98.66% recall, and 98.99% F1-score. Notably, FunProbe outperformed all our comparison targets for every architecture and compiler combination in terms of F1-score.

It is also noteworthy that FunProbe shows consistent performance on every architecture and compiler combination we tested. This result aligns well with our design principle that FunProbe should be *architecture*- and *compiler-agnostic*. For example, on `x86`, every other tool except FunProbe shows better performance on `GCC`-compiled binaries than on `Clang`-compiled binaries. We can also note how existing tools are fine-tuned to handle `GCC`-compiled binaries, which are more common in practice.

In terms of execution time (ET), FunProbe spent only 6.30 seconds on average for analyzing a binary in our benchmark. Moreover, FunProbe was able to analyze all the binaries without hitting the timeout. Although Nucleus records the best performance (1.5 seconds on average), the difference is less than 5 seconds, while FunProbe significantly outperforms Nucleus in terms of function identification accuracy.

While Binary Ninja, IDA Pro, and Ghidra show significantly slow running time overall, it is important to note that those tools perform not only function identification, but also other complex analyses, making the comparison not entirely fair. However, given that FunProbe performs various CFG-related analyses, these results show the practical impact of FunProbe. We argue that our technique can practically be used to improve binary analysis results because FunProbe can correctly find more function entry points than others in a reasonable amount of time.

5.5 Comparison with Learning-based Tools

How does FunProbe compare to an existing learning-based tool? We now compare the performance of FunProbe against XDA to answer this question. The comparison is made by running the tools on `x86` and `x86-64` architecture binaries as XDA only supports these architectures. In addition, we excluded the binaries that we used to fine-tune XDA for a fair comparison. In total, we used 5,412 binaries for the comparison.

We downloaded the pre-trained model of XDA from the official repository, and then used it to fine-tune our own model using our training dataset (see §5.1.3). We used a separate machine with a

Table 3: Function identification performance in terms of Precision (P), Recall (R), F1-score (F1), Execution Time (ET), and # of binaries failed due to TimeOut (TO).

Arch.	Compiler	FunProbe					Binary Ninja					IDA Pro					Ghidra					Nucleus				
		P (%)	R (%)	F1 (%)	ET (s)	TO	P (%)	R (%)	F1 (%)	ET (s)	TO	P (%)	R (%)	F1 (%)	ET (s)	TO	P (%)	R (%)	F1 (%)	ET (s)	TO	P (%)	R (%)	F1 (%)	ET (s)	TO
x86	GCC	**99.91**	**99.82**	**99.87**	6.24	0	89.23	97.44	93.16	92.42	0	90.67	90.09	90.38	13.04	0	97.66	98.89	98.27	153.97	1	95.78	93.34	94.54	**1.60**	0
	Clang	**99.94**	**98.69**	**99.31**	5.42	0	86.90	96.09	91.26	96.68	1	91.83	77.45	84.03	11.62	0	98.84	67.79	80.42	129.08	0	67.36	96.95	79.49	**1.69**	0
x86-64	GCC	**99.80**	**99.69**	**99.75**	5.55	0	92.76	96.79	94.73	92.90	0	93.12	85.93	89.38	10.33	0	97.72	98.33	98.02	151.54	0	95.52	91.71	93.58	**1.60**	0
	Clang	**99.97**	**99.75**	**99.86**	6.00	0	89.48	94.93	92.12	82.64	0	93.05	83.93	88.26	10.17	0	99.82	98.63	99.22	143.19	9	92.07	94.52	93.28	**1.63**	0
ARMv7	GCC	**99.59**	96.76	**98.16**	**11.41**	0	93.69	**98.34**	95.96	56.34	0	92.34	96.97	94.60	30.87	0	99.41	61.44	75.94	138.54	2	-	-	-	-	-
	Clang	96.98	**96.69**	**96.83**	**11.67**	0	91.82	95.85	93.80	62.95	0	89.56	92.99	91.24	22.01	0	**99.76**	66.99	80.15	94.27	5	-	-	-	-	-
AArch64	GCC	**99.90**	**99.67**	**99.78**	5.30	0	92.00	93.50	92.74	79.87	0	92.78	98.44	95.52	24.29	0	89.26	98.15	93.50	95.50	0	88.92	76.56	82.28	**1.26**	0
	Clang	**99.87**	**99.77**	**99.82**	5.09	0	88.33	95.21	91.64	108.33	3	92.63	98.39	95.42	20.69	0	99.78	98.36	99.07	129.27	7	91.46	80.55	85.66	**1.20**	0
MIPS	GCC	97.36	**98.24**	**97.80**	**5.23**	0	84.28	96.42	89.94	112.58	10	86.70	81.79	84.18	38.93	0	**99.31**	79.67	88.41	176.35	6	-	-	-	-	-
	Clang	99.66	**98.53**	**99.09**	**4.16**	0	84.37	95.82	89.73	103.79	0	89.04	76.18	82.11	48.89	1	**99.75**	75.09	85.68	159.06	4	-	-	-	-	-
MIPS64	GCC	99.54	**97.55**	**98.54**	**4.42**	0	-	-	-	-	-	86.22	83.94	85.06	34.84	8	**99.81**	64.69	78.50	181.57	4	-	-	-	-	-
	Clang	99.48	**98.74**	**99.11**	**5.09**	0	-	-	-	-	-	86.09	76.80	81.18	38.64	0	**99.83**	63.80	77.84	165.53	3	-	-	-	-	-
	Total	**99.33**	**98.66**	**98.99**	6.30	0	89.26	96.04	92.53	88.83	14	90.04	86.94	88.64	25.19	9	97.85	81.10	88.69	143.16	58	86.81	89.19	87.98	**1.50**	0

The numbers in bold represent the best result per row.

Table 4: Performance of FunProbe and XDA in terms of Precision (P), Recall (R), F1-score (F1), and Execution Time (ET).

Arch.	Compiler	FunProbe				XDA			
		P (%)	R (%)	F1 (%)	ET (s)	P (%)	R (%)	F1 (%)	ET (s)
x86	GCC	**99.91**	**99.82**	**99.87**	**6.24**	99.77	99.62	99.70	35.29
	Clang	**99.94**	98.69	99.31	**5.42**	99.84	**99.89**	**99.87**	37.02
x86-64	GCC	99.80	**99.69**	**99.75**	**5.55**	99.48	99.35	99.41	35.83
	Clang	**99.97**	99.75	**99.86**	**6.00**	99.82	**99.73**	99.78	36.58
	Total	**99.90**	99.49	**99.70**	**5.80**	99.73	**99.65**	99.69	36.18

The numbers in bold represent the best result per row.

powerful GPU (GeForce 3090 Ti), but the fine-tuning process took more than *2.5 days*. On the other hand, FunProbe does not require any training process, and this makes FunProbe more practical to use on any benchmarks. For the inference process, we used our server machine described in §5.1.4. Specifically, we assigned a single CPU of our server machine to run both FunProbe and XDA, but we had to additionally assign a single GPU to run XDA as it also uses GPU in the inference phase.

Table 4 summarizes the comparison between FunProbe and XDA on x86 and x86-64 binaries. Overall, both tools show comparable performance, but FunProbe had a slightly better F1-score than XDA. It records 99.7% F1-score on average whereas XDA shows slightly a low (99.69%), F1-score.

In terms of execution time, FunProbe was considerably faster than XDA even though XDA utilizes additional GPU resources. To analyze a single binary, FunProbe spent 5.8 seconds on average whereas XDA spent 36.18 seconds on average. That is, FunProbe was overall 6.2× faster than XDA without regard to the training cost. Therefore, we conclude that FunProbe is more scalable than XDA while being as precise as XDA.

5.6 Combining FunProbe and XDA

Can FunProbe benefit deep-learning-based approach or vice versa? To answer this question, we conduct an additional experiment by simply using FunProbe as a post-processor of XDA. Specifically, we first ran XDA on the binaries used in §5.5. We then created a positive hint for each function entry point identified by XDA, and fed in those hints to FunProbe. This means FunProbe will consider those XDA-generated hints as positive evidence to find function entry points in the same probabilistic framework we used.

As a result, the combined tool achieved 99.91% in all accuracy measures (precision, recall, and F1-score). Given that both FunProbe and XDA already accomplished high accuracy, it is surprising that the combined tool achieves phenomenal performance. Additionally, running FunProbe as a post-processor of XDA is a matter of a few seconds for each binary. This result confirms that FunProbe and the state-of-the-art deep-learning approaches are indeed complementary to each other.

We further analyzed what kind of functions that the combined tool was able to identify while the original XDA failed. The most common error cases corrected by the combined tool were where a function contains only a few instructions. For example, a tiny function containing two instructions (`mov` followed by `ret`) was not identified by XDA. Although we cannot directly understand the reason for failure due to the lack of interpretability of the model, we conjecture that this is because such a function is considered to be a function epilogue than a prologue by XDA.

6 DISCUSSION AND FUTURE WORK

Supporting other file formats. Currently, FunProbe only supports ELF binaries. However, extending our support for more binary types, e.g., Mach-O and PE, is straightforward because all the hints we leveraged (summarized in Table 1) are easily applicable to binaries in other formats.

Parameter tuning. Although FunProbe currently uses empirically chosen probability parameters (0.65 for positive hints and 0.4 for negative hints), one may use different probabilities for each different hint. This is because each hint may have a different impact on function identification. We believe fine-tuning probability parameters for each different hint will further improve the performance of FunProbe, and it is a promising direction for future research.

Better pruning strategy. While our bogus dependency pruning significantly improves the performance of FunProbe, it can potentially remove critical (i.e., non-bogus) dependencies in our model. Although it is beyond the scope of this paper to find a better pruning strategy, one may consider using a more sophisticated pruning strategy that can identify true bogus dependencies.

Obfuscation. FunProbe does not consider obfuscated binaries. We leave it as future work to combine FunProbe with existing deobfuscation techniques such as [57].

7 RELATED WORK

7.1 Binary Function Identification

There has been significant research effort on binary-level function identification for more than two decades. Early research focuses on systematically recovering CFGs [9], thereby naturally identifying functions by following call edges from recovered CFGs. For example, Jakstab [27] performs data flow analyses to determine indirect call edges. There are several recent papers following this direction [2, 4, 11, 45, 59]. For example, Nucleus [4] constructs interprocedural CFGs to find call targets as well as unreachable targets. rev.ng [11] converts a binary to an LLVM IR, and performs a static analysis on top of IR code to recover CFGs. All these approaches suffer from the accuracy of the analyses, and thus, employ various heuristics to improve the effectiveness of their analyses. However, such heuristics are inherently architecture and compiler dependent.

Pattern-based approaches are widely adopted in mainstream binary analysis tools [1, 8, 16, 34, 38, 53, 54]. For example, BAP [8] and Dyninst [34] utilize pre-trained decision trees to identify functions. IDA Pro [17, 18] and Ghidra [39] provide their own pattern databases to match well-known function patterns. Recently FunSeeker [25] shows that one can precisely detect functions from Intel CET binaries by leveraging the usage patterns of `endbr` instructions. However, all these techniques rely on previously-known patterns, and thus, inherently suffer from handling binaries with unknown patterns. On the other hand, the hints that we use are *not* specific to an architecture nor a compiler.

There is a recent metadata-based function identification technique, named FETCH [41], which leverages exception handling information to identify functions. Their technique significantly outperforms existing techniques, except for binaries without exception information, such as C binaries generated from Clang.

There also have been various deep-learning-based approaches [44, 50, 56, 60]. Shin *et al.* [50] use a bi-directional and multi-layer RNN to predict function boundaries. FID [56] leverages symbolic execution to extract feature vectors from binaries, to make its model robust against highly optimized binaries. XDA [44] first pre-trains a transformer model using Masked Language Modeling and self-attention layers, and fine-tunes the model for specific disassembly tasks, such as function identification. DeepDi [60] uses the R-GCN model to learn instruction embeddings on Instruction Flow Graph, and identifies function entry points with trained instruction patterns near the function. Although those approaches usually show good accuracy, they suffer from the generalization problem, that is, their performance relies on their training dataset [4, 30]. Moreover, they consume a lot of computational resources during both training and inference stages. They often require the use of GPUs for efficiency, and even more resources when the model size is large. On the other hand, FunProbe does not have any dependency on training data, or need for computation resources.

7.2 Probabilistic Binary Analysis

There are also diverse probabilistic-model-based approaches on binary code analysis [6, 36, 46, 62]. Rosenblum *et al.* [46] consider the function identification problem as a classification problem, and train a conditional random field model. ByteWeight [6] creates a weighted prefix tree based on the patterns of function start instructions (or bytes). Miller *et al.* [36] suggest a probabilistic disassembly algorithm. They collect probabilistic hints for valid instructions to disassemble true-positive instructions. OSPREY [62] utilizes a probabilistic graph model to recover variables and their type information from stripped binaries.

7.3 Probabilistic Inference

Various approaches to derive the desired distribution from a given probabilistic graphical model are suggested. Belief propagation [42] can solve tree-like graphs to get an exact solution. However, it may fail to converge or converge to the wrong solution if the graph contains loops. To handle loopy graphs, loopy belief propagation [37] iteratively runs belief propagation algorithm until it converges or a certain number of iterations is reached. To approximate the solution from loopy graphs, Junction Tree algorithm [23] transforms the graphs into junction trees, and solves the graph. However, it does not scale to handle large graphs. Variational Bayesian inference [14] finds an alternative distribution to approximate the complicated distribution. It is known to converge fast, but it requires heavy computation. Monte Carlo method [35] is another approach to approximate the exact distribution based on random sampling. Despite its simplicity, it is hard to apply to our system because it does not work well if the space is high-dimensional.

8 CONCLUSION

In this paper, we introduced a novel way to identify function entry points from a stripped binary. The key idea was to regard a function identification heuristic as a producer of a probabilistic hint. We can then combine those hints in a graphical model, i.e., a Bayesian Network, and compute the probability of each address in the binary being a function entry point. To boost the speed of our probabilistic inference, we proposed a novel approach, named bogus dependency pruning, and empirically proved its effectiveness. We implemented FunProbe to realize these ideas, and showed its practical impact by comparing its performance against five state-of-the-art tools in terms of both speed and accuracy. Our experiments were performed based on the benchmark consisting of 19,872 binaries compiled for 6 major CPU architectures, which is, to our knowledge, the largest benchmark used in the field. As a result, FunProbe outperformed all of the state-of-the-art tools we tested.

DATA AVAILABILITY

Our tool is available at https://zenodo.org/record/7602175. Upon acceptance, we will publicize it on GitHub.

REFERENCES

[1] [n. d.]. Radare2. https://github.com/radare/radare2.

[2] Jim Alves-Foss and Jia Song. 2019. Function Boundary Detection in Stripped Binaries. In *Proceedings of the Annual Computer Security Applications Conference.* 84–96.

[3] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the USENIX Security Symposium.* 583–600.

[4] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of IEEE European Symposium on Security and Privacy.* 177–189.

[5] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Proceedings of the International Conference on Compiler Construction.* 250–254.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to recognize functions in binary code. In *Proceedings of the USENIX Security Symposium.* 845–860.

[7] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the Network and Distributed System Security Symposium.*

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the International Conference on Computer Aided Verification.* 463–469.

[9] Cristina Cifuentes and Mike Van Emmerik. 2000. UQBT: Adaptable Binary Translation at Low Cost. *Computer* 33, 3 (2000), 60–66.

[10] Gregory F Cooper. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial intelligence* 42, 2-3 (1990), 393–405.

[11] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the International Conference on Compiler Construction.* 131–141.

[12] Frank DiMaio and Jude Shavlik. 2006. Improving the efficiency of belief propagation in large, highly connected graphs. *Working Paper 06-1, UWML Research Group* (2006).

[13] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *Proceedings of the USENIX Security Symposium.* 1075–1092.

[14] Charles W Fox and Stephen J Roberts. 2012. A tutorial on variational Bayesian inference. *Artificial intelligence review* 38, 2 (2012), 85–95.

[15] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security.* 637–651.

[16] Hex-Rays SA. [n. d.]. IDA Pro. https://www.hex-rays.com/products/ida/.

[17] Hex-Rays SA. [n. d.]. IDA Pro FLIRT. https://hex-rays.com/products/ida/tech/flirt/.

[18] Hex-Rays SA. [n. d.]. IDA Pro Lumina Server. https://hex-rays.com/products/ida/lumina/.

[19] H.J. Lu. [n. d.]. gcc/ChangeLog-2010. https://github.com/gcc-mirror/gcc/blob/master/gcc/ChangeLog-2010.

[20] H.J. Lu. [n. d.]. Turn on -fomit-frame-pointer by default for 32bit Linux/x86. https://gcc.gnu.org/legacy-ml/gcc-patches/2010-08/msg00922.html.

[21] Saehan Jo, Jaemin Yoo, and U Kang. 2018. Fast and scalable distributed loopy belief propagation on real-world graphs. In *Proceedings of the ACM International Conference on Web Search and Data Mining.* 297–305.

[22] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. 2019. B2R2: Building an Efficient Front-End for Binary Analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research.*

[23] David Kahle, Terrance Savitsky, Stephen Schnelle, and Volkan Cevher. 2008. Junction tree algorithm. *Stat* 631 (2008).

[24] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* (2022), 1–23. https://doi.org/10.1109/TSE.2022.3187689

[25] Hyungseok Kim, Junoh Lee, Soomin Kim, SeungIl Jung, and Sang Kil Cha. 2022. How'd Security Benefit Reverse Engineers? The Implication of Intel CET on Function Identification. In *Proceedings of the International Conference on Dependable Systems and Networks.* 559–566.

[26] 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 International Conference on Automated Software Engineering.* 353–364.

[27] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the International Conference on Computer Aided Verification.* 423–427.

[28] Alec Kirkley, George T Cantwell, and MEJ Newman. 2021. Belief propagation for networks with loops. *Science Advances* 7, 17 (2021), eabf1211.

[29] Mieczysław A Kłopotek. 2006. Cyclic Bayesian network: Markov process approach. *Studia Informatica: systems and information technology* 1, 7) (2006), 47–55.

[30] Hyungjoon Koo, Soyeon Park, and Taesoo Kim. 2021. A Look Back on a Function Identification Problem. In *Proceedings of the Annual Computer Security Applications Conference.* 158–168.

[31] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the USENIX Security Symposium.* 340–353.

[32] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[33] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium.* 251–268.

[34] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the International Symposium on Software Testing and Analysis.* 24–35.

[35] Nicholas Metropolis and Stanislaw Ulam. 1949. The monte carlo method. *Journal of the American statistical association* 44, 247 (1949), 335–341.

[36] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings of the International Conference on Software Engineering.* 1187–1198.

[37] Kevin Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proceedings of the Converence on Uncertainty in Artificial Inteligence.* 467–476.

[38] National Security Agency. [n. d.]. Ghidra. https://ghidra-sre.org.

[39] National Security Agency. [n. d.]. Ghidra x86 patterns. https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Processors/x86/data/patterns.

[40] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly but Were Afraid to Ask. In *Proceedings of the IEEE Symposium on Security and Privacy.* 833–851.

[41] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. 2021. Towards Optimal Use of Exception Handling Information for Function Detection. In *Proceedings of the International Conference on Dependable Systems and Networks.* 338–349.

[42] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc.

[43] Kexin Pei. [n. d.]. xda/issue-4. https://github.com/CUMLSec/XDA/issues/4.

[44] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the Network and Distributed System Security Symposium.*

[45] Rui Qiao and R. Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *Proceedings of the International Conference on Dependable Systems and Networks.* 201–212.

[46] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code. In *Proceedings of the AAAI Conference on Artificial Intelligence.* 798–804.

[47] Victor Garcia Satorras and Max Welling. 2021. Neural enhanced belief propagation on factor graphs. In *International Conference on Artificial Intelligence and Statistics.* PMLR, 685–693.

[48] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. In *Proceedings of the ACM Conference on Computer and Communications Security.* 426–441.

[49] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In *Proceedings of the USENIX Security Symposium.* 353–368.

[50] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the USENIX Security Symposium.* 611–624.

[51] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy.* 138–157.

[52] Alexander L Tulupyev and Sergey I Nikolenko. 2005. Directed cycles in Bayesian belief networks: probabilistic semantics and consistency checking complexity. In *Mexican International Conference on Artificial Intelligence.* Springer, 214–223.

[53] UCSB SecLab. [n. d.]. Angr. https://github.com/angr/angr.

[54] Vector 35. [n. d.]. Binary Ninja. https://binary.ninja/.

[55] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the Network and Distributed System Security Symposium.*

[56] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution.* 388–398.

[57] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings*

of the IEEE Symposium on Security and Privacy. 674–691.

[58] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the Network and Distributed System Security Symposium*.

[59] Xiaokang Yin, Shengli Liu, Long Liu, and Da Xiao. 2018. Function Recognition in Stripped Binary of Embedded Devices. *IEEE Access* 6 (2018), 75682–75694.

[60] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *Proceedings of the USENIX Security Symposium*.

[61] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium*. 337–352.

[62] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *Proceedings of the IEEE Symposium on Security and Privacy*. 813–832.