Compiler-Agnostic Function Detection in Binaries

Dennis Andriesse*†, Asia Slowinska‡, Herbert Bos*†
*{d.a.andriesse,h.j.bos}@vu.nl
Computer Science Institute, Vrije Universiteit Amsterdam
†*Amsterdam Department of Informatics*
‡*asia.slowinska@gmail.com*

Abstract—**We propose *Nucleus*, a novel function detection algorithm for binaries. In contrast to prior work, *Nucleus* is compiler-agnostic, and does not require any learning phase or signature information. Instead of scanning for signatures, *Nucleus* detects functions at the Control Flow Graph-level, making it inherently suitable for difficult cases such as non-contiguous or multi-entry functions. We evaluate *Nucleus* on a diverse set of 476 C and C++ binaries, compiled with `gcc`, `clang` and Visual Studio for x86 and x64, at optimization levels O0–O3. We achieve consistently good performance, with a mean F-score of 0.95.**

1. Introduction

Function detection is a binary analysis technique that categorizes the code within a binary into functions approximating the original (source-level) functions. It is a key building block in areas like binary instrumentation [1], [2], binary-level vulnerability search [3], [4], and binary protection schemes, including Control-Flow Integrity [5]–[8]. Moreover, accurate function detection is crucial for human reverse engineers, who rely on such compartmentalization to aid their reasoning about complex binary code.

Related work shows that while modern disassemblers and binary analysis platforms achieve high accuracy in terms of instruction recovery, their function detection capabilities are still lacking [9]. For instance, for stripped x64 ELF binaries generated with the common `gcc` compiler, our results show that the prominent IDA Pro disassembler misidentifies 25% to 40% (depending on optimization level) of functions *on average*, and up to 75% in the worst case. Moreover, up to 20% of the reported functions are false positives. Other disassemblers, such as Dyninst [2] and BAP [10], deliver comparable or worse performance.

The predominant approach to the function detection problem is to use a signature database to scan binaries for known function prologues and epilogues. This approach is used even in state-of-the-art work like ByteWeight, which uses machine learning to automatically generate signatures [11], [12]. While signature-based function detection can achieve reasonable accuracy for unoptimized binaries, its performance declines steeply for highly optimized binaries, where standard function prologues are often missing altogether. Moreover, signature databases require constant maintenance in order to support new compilers, compiler versions and architectures.

This paper proposes a new signature-less approach to function detection for stripped binaries, based on *structural Control Flow Graph analysis*. We provide an open-source implementation of our approach, called *Nucleus*.[1] Rather than scanning binaries for signatures, *Nucleus* is centered around an Interprocedural Control Flow Graph (ICFG), which it constructs by disassembling a binary and analyzing its control flow. *Nucleus* identifies functions in the ICFG by analyzing the control flow between basic blocks, based on our observation that intraprocedural control flow tends to use different types and patterns of control flow instructions than interprocedural control flow. We show that this property holds across different compilers and optimization levels, allowing *Nucleus* to identify functions in a completely compiler-agnostic way, without any compiler-specific signatures or heuristics. *Nucleus* also inherently supports difficult cases like non-contiguous and multi-entry functions. *Nucleus* can export its results directly to the popular IDA Pro disassembler, making it easy to use in real-world scenarios.

We evaluate *Nucleus* on a diverse set of 476 binaries, which includes binaries compiled with `gcc`, `clang` and Visual Studio for both Linux (ELF) and Windows (PE). Our evaluation covers both C and C++ code, compiled for x86 (32-bit) and x64 (64-bit), at optimization levels ranging from O0 to O3. *Nucleus* achieves mean precision and recall rates of 0.96 and 0.94, respectively; consistently outperforming IDA Pro and Dyninst, and matching the reported accuracy of state-of-the-art machine learning-based work [11], [12].

Further, our evaluation reveals a significant discrepancy between the accuracy reported for these machine learning approaches (specifically ByteWeight [11]), and the results they deliver in our tests. Upon closer analysis, we find a large overlap between the training set and test set used to evaluate *all* top-tier work on machine learning for function detection, including ByteWeight [11], [12]. We show that this leads to a large bias in the evaluations for these papers, underlining the need for future work to reassess the viability of machine learning for function detection.

1. https://www.vusec.net/projects/function-detection/

DOI 10.1109/EuroSP.2017.11

1.1. Contributions

Summarizing, our contributions are as follows.

- We introduce *Nucleus*, a novel compiler-agnostic function detection engine. *Nucleus* achieves high accuracy for all major compilers and platforms, without requiring any of the compiler-specific signatures used by current state-of-the-art algorithms.
- *Nucleus* is open source, and is easy to use in real-world environments due to its ability to integrate with IDA Pro, the industry-standard disassembler.
- In contrast to prior work, *Nucleus* can support new compilers without any training or maintenance.
- *Nucleus* provides inherent support for difficult cases, such as non-contiguous and multi-entry functions, without assuming anything about the memory or instruction layout of functions.
- We find a strong bias in the evaluations of top-tier work on machine learning-based function detection, demonstrating that these techniques need to be reassessed before the accuracy reported in their evaluations can be assumed to generalize.

1.2. Outline

The rest of this paper is organized as follows. First, we discuss the background of function detection in Section 2. Next, we provide an overview of *Nucleus* in Section 3, and discuss implementation details in Section 4. We evaluate the accuracy and performance of *Nucleus* in Section 5. In Section 6, we qualify our comparison of *Nucleus* with top-tier machine learning-based work by analysing in-depth the test suite used to evaluate this work. We discuss the implications of our results in Section 7, and contrast our approach to related work in Section 8. Finally, we present our conclusions in Section 9.

2. Background

This section provides a brief introduction to function detection. We discuss the definition and scope of the function detection problem, as well as challenging cases which need to be handled.

2.1. Definition of Function Detection

Function detection comprises two main problems: *function start detection*, and *function boundary detection*. In function start detection, the aim is to find all addresses in a binary that correspond to a function entry point, while function boundary detection attempts to find both the first and last address of each function. Our definitions of these are analogous to the definitions by Bao et al. [11].

We use these definitions to compare *Nucleus* to existing approaches in our evaluation (Section 5). However, *Nucleus* is not limited to detecting only function start and end addresses; as discussed in Section 3, *Nucleus* assigns *all* basic blocks to their containing functions.

(1) *Function start detection*: Given a binary P compiled from a set of source-level functions $F := \{f_1, f_2, \ldots, f_m\}$, identify a set of start addresses $S := \{s_1, s_2, \ldots, s_n\}$ in P such that s_i points to the machine instruction corresponding to the first line (*entry point*) of some $f_j \in F$. Note that for stripped binaries, F is *not* known to the function detector. Given a set of ground truth start addresses S_{gt}, we define the set of true positives as $TP := S \cap S_{gt}$, false positives as $FP := S \backslash S_{gt}$ and false negatives as $FN := S_{gt} \backslash S$.

(2) *Function boundary detection*: Given the same binary P compiled from functions in F, identify a set of (start, end) address pairs $B := \{(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n)\}$ in P such that s_i is the function start address of $f_j \in F$ and e_i is the last address in P corresponding to a line from f_j. Given again a set of ground truth function boundaries B_{gt}, we define the set of true positives as $TP := B \cap B_{gt}$, false positives as $FP := \{(s, e) \mid (s, e) \in B \wedge s \notin S_{gt}\}$, and false negatives as $FN := B_{gt} \backslash B$. Note that this implies that for TP, both the function start and end address must be correct; if either is incorrect this counts for FN.

2.2. Scope of Function Detection

For binaries with symbolic information, function detection is trivial—the symbol table specifies the set of functions, along with their names, start addresses, and sizes. Unfortunately, many binaries in practice are stripped of this information. This makes function detection far more challenging—source-level functions have no real meaning at the binary level, and their boundaries are frequently blurred by compiler optimizations. *Nucleus*, like other work on function detection [2], [11]–[13], focuses on the more challenging case of stripped binaries.

Though challenging, function detection in stripped binaries is important in virtually all forms of binary reverse engineering. Human reverse engineers often deal with stripped binaries, especially in malware analysis or security auditing of untrusted binaries [13], [14]. Decompilers attempt to facilitate human reverse engineering by deriving a high-level code representation from binaries, also operating at the function level [15], [16].

Automated reverse engineering and binary instrumentation systems also rely on accurate function detection for stripped binaries, such as legacy binaries or binaries for embedded systems (which are often stripped to save memory). For instance, Control Flow Integrity mechanisms often reason about security at the function level [5], [7], [8]. Moreover, automated bug detection systems [3], [4] and binary-level reoptimizers also commonly reason at the function level [17].

2.3. Signature-Based Approaches

The predominant strategy for function detection is based on signatures. This strategy is used in all well-known approaches, including IDA Pro [13], Dyninst [2] and machine learning approaches like ByteWeight [11] or Neural Network-based function detection [12].

Typically, signature-based function detection algorithms start with a pass over the disassembled binary to locate trivial functions that are directly addressed by a `call` instruction. To locate the remaining functions (such as indirectly called or tailcalled functions), these approaches scan for well-known signatures that indicate function prologues and epilogues. For instance, a typical pattern that many x86 compilers emit for unoptimized functions starts with the prologue `push ebp; mov ebp,esp`, and ends with the epilogue `leave; ret`. In practice, many more patterns are used, depending on the platform, compiler, and optimization level. Indeed, optimized functions may not have well-known function prologues or epilogues at all.

This wide variety of function patterns and calling conventions is a major problem for the scalability of signature-based function detection. Signature databases need to account for all these possibilities, and need constant maintenance to account for new platforms, compilers and compiler versions. Recent work by Bao et al. [11] and Shin et al. [12] has focused on automating the process of learning new function signatures. However, these approaches still require signatures tuned for specific compilers and an expensive learning phase for every configuration change. The scalability problems are especially apparent for open-source projects like GNU `gcc` and `llvm/clang`, which release new major versions roughly every six months, and minor versions with even higher frequency.[2,3]

2.4. Challenging Cases

We distinguish several constructs which are challenging for function detection. Typically, these result from compiler optimizations. Here, we provide a high-level overview of challenging constructs, while Sections 3–7 provide real-world examples, and discuss how *Nucleus* handles them. *Nucleus* successfully handles all of the cases discussed below, except where noted otherwise. We also provide a detailed discussion of cases not handled by *Nucleus* in Section 5.3.

(1) *Non-contiguous functions*: Many disassemblers, including IDA Pro, assume that each function is laid out in a single contiguous memory range. This assumption is convenient for signature-based function detection, which works by scanning for a function prologue and epilogue. Depending on the compiler and optimization level, functions may instead consist of multiple disjoint memory ranges, which require deeper analysis to be associated with the correct function.

(2) *Multi-entry functions*: Instead of a single entry point, a function may have multiple alternative entry points. For instance, `glibc` defines the `splice` function, which has an alternative entry called `__splice_nocancel` that may be called depending on whether thread safety is required. Function detectors which do not consider this may misclassify each alternative entry block as a separate function.

2. https://gcc.gnu.org/releases.html
3. http://llvm.org/releases/

(3) *Padding code and inline data*: Especially at high optimization levels, modern compilers add padding code between functions, and even between basic blocks within a function. This code is not intended to be executed, but to align functions and basic blocks in memory so they can be accessed with optimal efficiency. In addition, compilers like Visual Studio often intersperse data, such as jump tables, within code sections. As a result, function detectors must avoid inadvertently identifying padding or data as (part of) a function.

(4) *Unreachable code*: Recursive disassemblers like IDA Pro and Dyninst follow control flow to discover code. While this approach works well for separating code from data, it cannot discover functions that are never called, or are only called indirectly.

(5) *Tail calls*: In this common optimization, a function ends not with a return, but with a jump to another function. This makes it more difficult to detect where the optimized function ends. As we discuss in Section 5.3, this case requires dedicated handling by *Nucleus*.

(6) *Alternative prologues/epilogues*: Signature-based function detection tends to misidentify functions that use an unrecognized calling convention, or lack standard prologues and epilogues altogether (common in optimized code).

3. Overview

This section provides a high-level overview of our function detection algorithm. Implementation details are provided in Section 4. The main steps of our algorithm are illustrated in Figure 1.

Though our approach is conceptually simple, we show in Section 5 that it is able to detect both function starts and function boundaries with very high accuracy. Moreover, our evaluation shows consistently good results across multiple instruction sets, compilers and platforms, without requiring any compiler-specific heuristics. Additionally, in contrast to signature-based approaches, our analysis yields the complete set of basic blocks belonging to each function, rather than only a start and end address.

3.1. ICFG Generation

We start by generating the Interprocedural Control Flow Graph (ICFG) around which the rest of our analysis is centered (step ① in Figure 1). The ICFG for a binary B is a digraph $G = (V, E)$, where V is the set of all basic blocks in B, and E is the set of control flow edges $E \subseteq V \times V$ between basic blocks. E includes both intraprocedural and interprocedural edges (such as `call` edges). In contrast, the traditional definition of a (non-interprocedural) Control Flow Graph (CFG) is a function-level data structure that contains only the basic blocks and edges within a particular function. We operate on the ICFG because it can be generated without a priori knowledge of function boundaries. We generate the ICFG by disassembling the target binary, dividing it into basic blocks, and analyzing its control flow (see Section 4 for details).

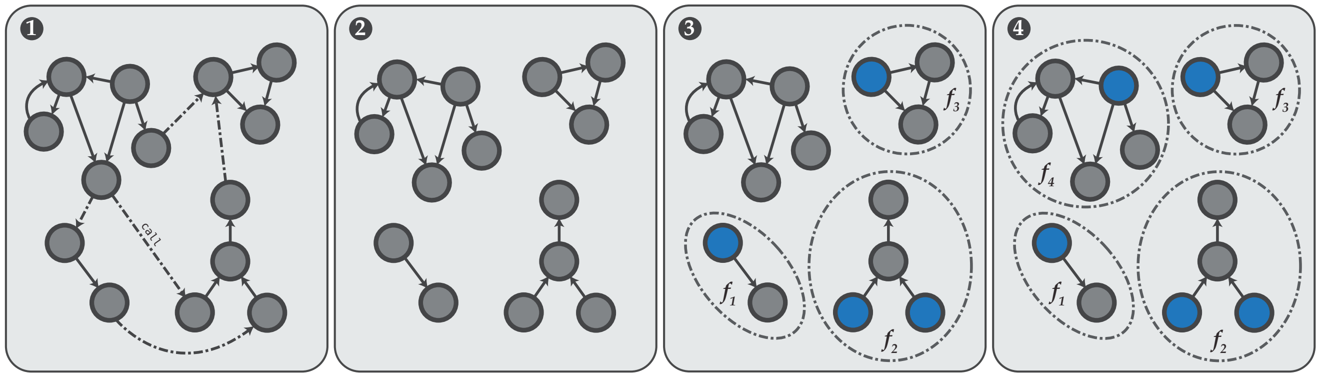

Figure 1: Overview of our function detection algorithm. ① Disassemble binary and generate ICFG. ② Hide edges $e \in E_{call}$. ③ Locate directly called entry points (shaded blue) and expand functions by following control flow (ignoring direction). ④ Find remaining functions through connected components analysis and detect entry points through intraprocedural control flow analysis.

To improve the accuracy of our analysis, we perform some preprocessing on the ICFG. Specifically, we use switch detection to resolve intraprocedural indirect jumps, and we use a combination of semantic analysis and reachability analysis to identify padding blocks and inline data. More details on our preprocessing algorithm are given in Section 4.

3.2. Connected Components Analysis

Next, we perform a weakly connected components analysis on the ICFG, temporarily excluding the `call` edges $e \in E_{call}$ (step ②). A weakly connected component is a subgraph of the ICFG, in which every two vertices are connected by an undirected path (i.e., ignoring the direction of control flow), and no vertex is connected to a vertex outside the component. By excluding `call` edges, the analysis finds all graph components that consist of basic blocks connected through only intraprocedural edges (some corner cases do exist; these are discussed in Section 5.3). In other words, we use the connected components analysis to find clusters of basic blocks, such that (ideally) each cluster contains all blocks belonging to a single function.

Note that this approach assumes nothing about the memory layout of functions, thus providing natural support for non-contiguous functions. Moreover, it does not require any kind of function prologue or epilogue detection (signature-based or otherwise), making our approach completely compiler agnostic. Our connected components analysis consists of several phases, as described below.

3.2.1. Directly Called Functions. First, we make a pass over the instructions of all basic blocks in the ICFG, scanning for direct `call` instructions. This allows us to detect the directly called function entry blocks, which we each expand into a complete function by following the edges from the entry block until a complete component is formed (step ③). This phase detects all components corresponding to directly called functions.

Listing 1 Effective `nop` instructions emitted by `gcc` v5.1.1 on x86. `eax` can be replaced by any general purpose register.

```
mov   eax,eax
xchg  eax,eax
lea   eax,[eax + 0x0]
lea   eax,[eax + eiz*1 + 0x0]
```

3.2.2. Unreachable/Indirectly Called Functions. Next, we find indirectly called or unreachable functions by iterating over all basic blocks in the ICFG, looking for basic blocks that are not yet part of a function (step ④). We expand each such block into a function using the aforementioned connected components analysis. Subsequently, we detect the function entry points by scanning the function for basic blocks that are not reached by any intraprocedural edge. (In practice, there may be loopback edges to an entry block; we describe our approach to dealing with these in Section 4.) We perform the same entry point analysis for the directly called functions, to detect possible multi-entry functions. If no suitable entry point can be found through other methods, our analysis assumes the function is entered at its lowest address (the default assumption in signature-based approaches).

4. Implementation

We implemented an open-source version of our function detection algorithm, called *Nucleus*, in 3278 C++ SLOC.The ICFG construction and function detection code consists of under 850 SLOC, while the remaining lines are attributed to our binary loader, disassembler, and utility code. We implemented a custom disassembly pass using Capstone v3.0.4 for instruction parsing [18]. *Nucleus* integrates with IDA Pro (the industry standard disassembler) by providing the option to generate an IDA Pro script that imports our function detection results into IDA Pro. This makes *Nucleus* straightforward to use in real-world environments.

4.1. Disassembly and ICFG Generation

To find indirectly called and unreachable functions, we use a linear disassembly approach, coupled with an analysis to detect padding code and inline data. Recent work has shown that linear disassembly, even with only simple detection of padding or data, can reliably achieve high code coverage with few disassembly errors [9]. After disassembly completes, *Nucleus* constructs the ICFG by breaking the code into basic blocks, and creating the control flow edges associated with each control flow instruction.

We then analyze each basic block to see if it consists of `nop` instructions or other do-nothing instructions, used for padding. Simply checking for `nop` instructions is not enough, because not all compilers use well-known `nop` instructions for padding. We therefore implement additional checks that look for instructions which move a source operand into a destination operand without modifying it. Listing 1 shows examples of such instructions, used for padding by `gcc` v5.1.1 on x86.

Moreover, we use reachability analysis to determine if a `nop` block is part of a function (reachable), or is padding (not reachable). We detect inline data by looking for basic blocks that contain invalid or privileged instructions. These blocks, and any basic blocks that can reach them via a jump or fallthrough edge, are marked as suspected data.

4.2. Switch Detection

Compilers typically implement switch statements as an indirect jump that selects its target from a *jump table* of code pointers, depending on which case should be executed. To correctly attribute all switch/case blocks to their associated function, we need to resolve these intraprocedural indirect jumps. *Nucleus* therefore implements a switch detection pass that performs a backward sweep starting from every indirect jump, looking for the instruction where the jump's target register is loaded. If this load instruction references a jump table, we scan this table for valid code pointers, adding these as targets of the indirect jump. More sophisticated switch detection is explored in related work [13], [19], [20], but is outside the scope of this work.

4.3. Function and Entry Point Detection

After ICFG generation is complete, we execute our connected components-based function detection algorithm as described in Section 3. As noted in Section 3, we implement several ways of detecting function entry points (in order of priority): (1) by following direct `call` edges, (2) using intraprocedural control flow analysis, and (3) by assuming the function's lowest address as the entry point (as a last resort).

The intraprocedural control flow analysis detects function entry points by looking for basic blocks that are not reached by any intraprocedural edge. However, we must also deal with entry blocks which *do* have incoming loopback edges. Such entry blocks can be identified in two ways: (1) Loopback edges typically target not the start of an entry block, but jump to an offset within it (skipping past the function prologue). Because *Nucleus* tracks the destination offset of each edge, we can identify these cases. (2) Alternatively, we use intraprocedural loop detection to determine that the entry block is reached only via a loopback edge (while the source of the loopback edge is also reached by other inbound control flow edges).

5. Evaluation

In this section, we evaluate four key aspects of *Nucleus*. (1) How accurate are our function detection results compared to existing work? (2) Does *Nucleus* achieve more stable cross-compiler/cross-architecture results than other approaches? (3) Which particular cases are handled well by *Nucleus*, and which cases cause false positives or false negatives? (4) How does the runtime performance of *Nucleus* compare to other approaches? We first describe our test setup, and then address each of these questions.

5.1. Test Setup

We evaluate *Nucleus* on a test suite consisting of 476 C and C++ binaries for x86 and x64—the most commonly targeted platforms in binary analysis research. Our test suite contains both Linux (ELF) and Windows (PE) binaries, compiled at optimization levels O0–O3. The ELF binaries are compiled with the popular `gcc` v5.1.1 and `clang` v3.7.0 compilers, while the PE binaries are compiled with Visual Studio 2015—these are the most recent versions at the time of our experiments. All of the binaries are stripped of any symbolic information.

Our test suite contains the SPEC CPU2006 C and C++ benchmarks, as well as the popular server applications `nginx` v1.8.0, `lighttpd` v1.4.39, `opensshd` v7.1p2, `vsftpd` v3.0.3 and `exim` v4.86. We choose this test suite for several reasons: (1) It contains a diverse range of realistic C and C++ binaries, ranging from very small to large; (2) By testing with C and C++, as well x86 and x64 binaries, we cover a wide range of both stack-based and register-based function calling conventions; (3) The tested binaries contain a wide variety of challenging corner cases—for instance, `perlbench` is known for containing many indirect function calls; (4) SPEC CPU2006 compiles on both Linux and Windows, allowing a fair comparison between `gcc`, `clang`, and Visual Studio.

We obtain ground truth on function starts and function boundaries by compiling the ELF binaries with full symbolic and DWARF v3 information, and the PE binaries with full PDB (Program Database) files. After parsing the required function information from these sources, we strip the binaries of all symbolic information before using them in our experiments.

We conduct our experiments on an Intel Core i5 4300U machine with 8GB of RAM, running Ubuntu 15.04. We compile our `gcc` and `clang` test cases on this same machine.

The Visual Studio binaries are compiled on an Intel Core i7 3770 machine with 8GB of RAM, running Windows 10.

We compare *Nucleus* with *IDA Pro v6.7*, *Dyninst v9.1.0* [2], and *BAP v0.9.9* [10], which uses *ByteWeight v0.9.9* [11] to obtain function start information. We choose these tools because they are capable of delivering both function start and function boundary information, are widely used, and are also used as a reference in the evaluations of related work [11]. Moreover, in Section 6, we provide a more detailed comparison with the results yielded by state-of-the-art machine learning-based approaches, including ByteWeight [11] and Shin et al. [12].

5.2. Function Detection Results

We report our experimental results using the F-score metric, and the related notions of precision and recall. The F-score is widely used (also in related work [11], [12]) because it provides a combined metric of the true positive, false positive and false negative rates of a system. Precision is defined as $p = |TP| / (|TP| + |FP|)$, while recall is defined as $r = |TP| / (|TP| + |FN|)$. For us, $p = 1.0$ means that all reported functions are true positives (no false positives), while $r = 1.0$ means that there are no false negatives. The F-score is the harmonic mean of precision and recall: $F = 2 \cdot p \cdot r / (p + r)$. The range of the F-score is again $[0.0, 1.0]$, with $F = 1.0$ denoting perfect accuracy (neither false positives nor false negatives).

In our Linux-based testing environment, Dyninst was unable to process PE binaries. We therefore report only ELF results for Dyninst. As our server applications are Linux-specific, we test them only for `gcc` and `clang`.

5.2.1. Function Starts. We begin by discussing results for function start detection; results for function boundary detection are discussed in Section 5.2.2. Figure 2 shows the F-scores achieved by *Nucleus* and the other approaches per platform (x86 versus x64), compiler, and optimization level, differentiating between the C and C++ tests. For each case, the graph shows the geometric mean result achieved for SPEC CPU2006. Additionally, Table 1 shows the decomposition of the F-scores into precision and recall rates, for both the SPEC and server tests. For space reasons, Table 1 shows average scores taken over the geometric means for all optimization levels.

Figure 2 shows that *Nucleus* achieves accurate results across all compilers, platforms and optimization levels. We achieve an overall average F-score for SPEC of 0.96, ranging between 0.92 (O3) and a perfect F-score of 1.00 (O0) for the C tests, and between 0.87 and 0.99 for C++. As shown in Table 1, *Nucleus* consistently outperforms all other disassemblers, especially in terms of recall, with the exception of Visual Studio on x64. Although *Nucleus* delivers accurate results for Visual Studio x64, IDA Pro is the most accurate for this compiler, with an average F-score of 0.97. This is due to the fact that for x64, Visual Studio 2015 uses only one calling convention [21], making IDA Pro's signature-based approach extremely effective. In contrast, other compilers use a variety of calling conventions.

As can be seen in Figure 2, *Nucleus* is more tolerant of varying compilers and optimization levels than other approaches, since *Nucleus* is compiler-agnostic. *Nucleus* shows stable accuracy across compilers and architectures, and the decrease in accuracy for high optimization levels is far less significant than for all other tested function detectors. In Section 5.3, we provide a detailed discussion of the specific challenging cases which occur in optimized binaries.

The standard deviations in F-score for *Nucleus* are limited to the range $[0.01, 0.04]$ for C, and $[0.00, 0.07]$ for C++. In contrast, deviations for IDA Pro range from $[0.00, 0.13]$ for C (with deviations below 0.11 only occurring for Visual Studio), to $[0.00, 0.19]$ for C++. Dyninst standard deviations range from $[0.01, 0.14]$ for C to $[0.01, 0.16]$ for C++, while BAP/ByteWeight ranges from $[0.04, 0.16]$ for C to $[0.02, 0.26]$ for C++. Again, this shows that *Nucleus* provides accurate results more consistently than signature-based approaches.

The results shown in Figure 2 and Table 1 for ByteWeight are based on the open-source ByteWeight version shipped with BAP v0.9.9, which we refer to as *BAP/ByteWeight v0.9.9*. BAP uses ByteWeight to detect function starts, which it uses as starting points for disassembly. We tested BAP/ByteWeight with the default ELF/PE signatures that are included with it. In our tests, BAP/ByteWeight achieves significantly lower accuracy than the other approaches, with an overall mean F-score of only 0.65, which is 0.32 points lower than reported in the Byteweight paper [11]. Results for the server tests (which do not include C++ binaries) are more accurate, at a mean F-score of 0.75, but this is still 0.22 points lower than expected. Note that for BAP/ByteWeight, we excluded `xalancbmk` at O3 from the tests because of scalability issues (see Section 5.4).

To investigate this discrepancy more closely, we requested the trained version of ByteWeight used in the original paper from the authors. Unfortunately, the authors replied that only an untrained version is still available. Given the uncertainties in attempting to exactly reproduce the training used in the ByteWeight paper, we instead performed a detailed analysis of the difference between our test suite, and the tests used in the ByteWeight paper. This analysis, which we discuss in Section 6, shows a significant overlap between the training set and test set used in the original ByteWeight evaluation. We show that this overlap causes a significant bias in evaluation results, which we believe is responsible for the accuracy discrepancy we observe. This is worrying, because the ByteWeight test suite has since been used to evaluate *all* top-tier work on function detection through machine learning.

5.2.2. Function Boundaries. Figure 3 and Table 2 show our results for function boundary detection. Recall from Section 2 that in contrast to function start detection, which only finds the first address of each function, function boundary detection involves finding both the first and the last address.

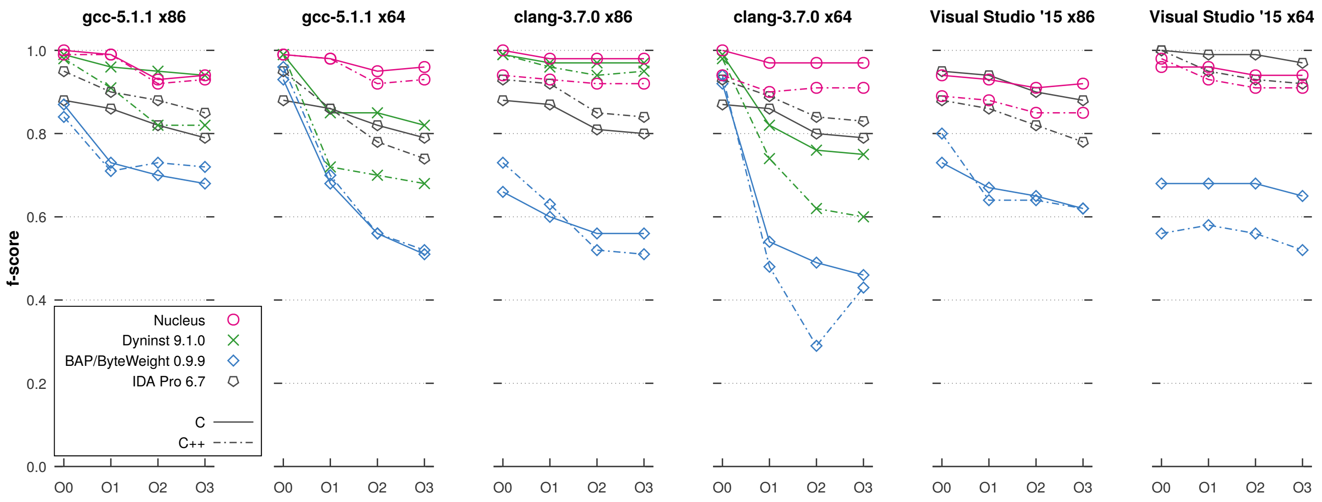

Figure 2: F-scores for *function start* detection (geometric mean for SPEC CPU2006).

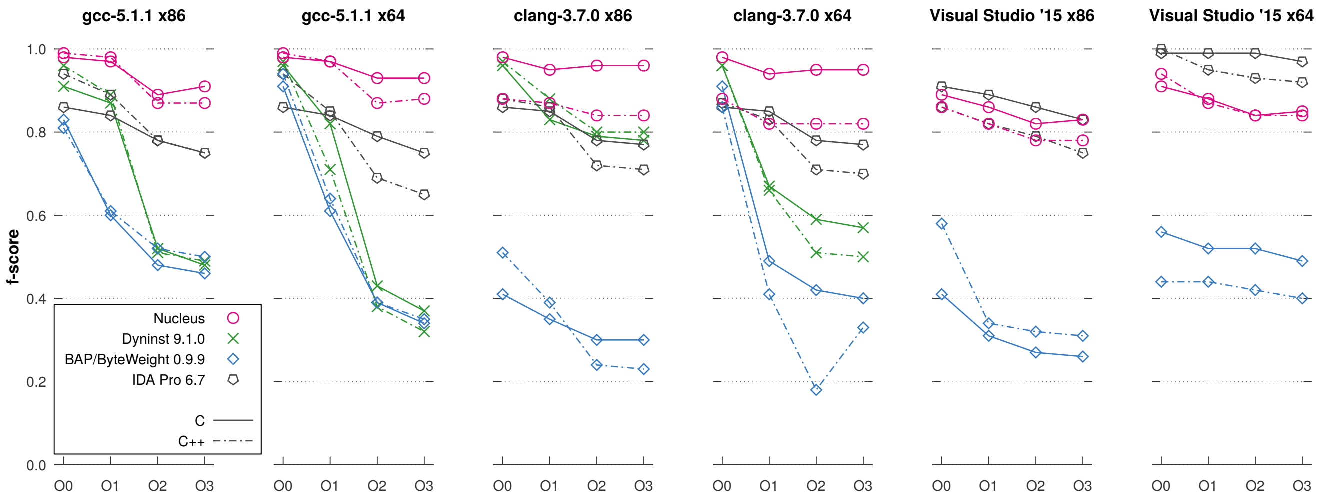

Figure 3: F-scores for *function boundary* detection (geometric mean for SPEC CPU2006).

As discussed in Section 3, *Nucleus* finds not only function boundaries, but all basic blocks belonging to each function. Nevertheless, for comparability with the results of other approaches, we measured our results for *Nucleus* by taking the lowest and highest address found for each function.

As before, Figure 3 graphs the F-scores achieved for SPEC CPU2006 in various configurations, while Table 2 decomposes these F-scores into precision and recall, and additionally shows results for our server tests. Again, *Nucleus* consistently outperforms other approaches, with the exception of IDA Pro on Visual Studio x64. (As discussed in Section 5.2.1, this is because Visual Studio uses only one calling convention on x64.) *Nucleus* achieves an overall mean F-score of 0.90 for the SPEC CPU2006 tests, while IDA Pro (the best performing alternative) yields a mean F-score of only 0.84, even including its extremely good results for Visual Studio x64. For our server tests (which do not include C++ code), *Nucleus* achieves an even higher overall mean F-score of 0.97.

In addition, the standard deviations in F-score for *Nucleus* are lower than those for other approaches, meaning that *Nucleus* provides more predictable accuracy. *Nucleus* achieves an average standard deviation of only 0.02 for C, and 0.04 for C++. In contrast, IDA Pro, the best performing other approach, has an average standard deviation of 0.10 for C, and 0.11 for C++. Moreover, Figure 3 shows that *Nucleus* again achieves more stable results across compilers and architectures than other approaches, while better retaining its accuracy for highly optimized binaries (including optimization levels O2 and up).

	gcc x86	**gcc x64**	**clang x86**	**clang x64**	**VS x86**	**VS x64**
IDA Pro 6.7	0.98/0.78	0.97/0.74	0.98/0.78	0.98/0.77	0.84/0.93	1.00/0.94
BAP/ByteWeight 0.9.9	0.68/0.83	0.70/0.66	0.52/0.71	0.73/0.49	0.63/0.74	0.69/0.56
Dyninst 9.1.0	0.93/0.91	0.96/0.74	0.98/0.95	0.88/0.72	—	—
Nucleus	0.98/0.96	0.98/0.96	0.96/0.97	0.96/0.95	0.86/0.96	0.95/0.94
$\Delta_{Nucleus}$	$+0.00/+0.05$	$+0.01/+0.22$	$-0.02/+0.02$	$-0.02/+0.18$	$+0.02/+0.03$	$-0.05/+0.00$

(a) SPEC CPU2006 (all binaries, optimization levels O0–O3)

	gcc x86	**gcc x64**	**clang x86**	**clang x64**
IDA Pro 6.7	0.93/0.88	0.92/0.86	0.93/0.85	0.91/0.84
BAP/ByteWeight 0.9.9	0.71/0.91	0.78/0.86	0.57/0.84	0.79/0.65
Dyninst 9.1.0	0.91/0.96	0.92/0.85	0.93/0.97	0.87/0.85
Nucleus	0.98/0.98	0.98/0.97	0.99/0.99	0.99/0.96
$\Delta_{Nucleus}$	$+0.05/+0.02$	$+0.06/+0.11$	$+0.06/+0.02$	$+0.08/+0.11$

(b) Servers (C only, tested at per-server default optimization ranging from O0–O2)

TABLE 1: Precision/recall for *function start* detection (average geometric mean). $\Delta_{Nucleus}$ shows the improvement in *Nucleus* over other approaches.

	gcc x86	**gcc x64**	**clang x86**	**clang x64**	**VS x86**	**VS x64**
IDA Pro 6.7	0.97/0.71	0.97/0.68	0.98/0.68	0.97/0.68	0.83/0.85	1.00/0.94
BAP/ByteWeight 0.9.9	0.60/0.60	0.63/0.53	0.34/0.34	0.68/0.41	0.40/0.32	0.61/0.40
Dyninst 9.1.0	0.89/0.60	0.91/0.51	0.98/0.75	0.85/0.57	—	—
Nucleus	0.97/0.89	0.97/0.90	0.95/0.88	0.94/0.86	0.85/0.84	0.94/0.85
$\Delta_{Nucleus}$	$+0.00/+0.18$	$+0.00/+0.22$	$-0.03/+0.13$	$-0.03/+0.18$	$+0.02/-0.01$	$-0.06/-0.09$

(a) SPEC CPU2006 (all binaries, optimization levels O0–O3)

	gcc x86	**gcc x64**	**clang x86**	**clang x64**
IDA Pro 6.7	0.93/0.83	0.92/0.81	0.93/0.83	0.92/0.82
BAP/ByteWeight 0.9.9	0.67/0.75	0.75/0.74	0.42/0.47	0.77/0.52
Dyninst 9.1.0	0.91/0.79	0.92/0.70	0.93/0.85	0.85/0.74
Nucleus	0.98/0.96	0.98/0.94	0.99/0.97	0.99/0.93
$\Delta_{Nucleus}$	$+0.05/+0.13$	$+0.06/+0.13$	$+0.06/+0.12$	$+0.07/+0.11$

(b) Servers (C only, tested at per-server default optimization ranging from O0–O2)

TABLE 2: Precision/recall for *function boundary* detection (average geometric mean). $\Delta_{Nucleus}$ shows the improvement in *Nucleus* over other approaches.

5.3. Analysis of Results

In Section 2.4, we discussed challenging constructs for function detection. As shown in Section 5.2, *Nucleus* achieves significantly more accurate results than other approaches. To gain a better understanding of the errors which do occur in *Nucleus*, and the tradeoffs compared to other approaches, we select and manually analyze a random sample of 100 false positives and false negatives from our experiments. The sample includes all compilers and platforms we tested, and covers both our function start and function boundary detection experiments.

5.3.1. False Positives. As discussed in Section 3, *Nucleus* uses connected components analysis of the ICFG to detect functions without assuming anything about their memory layout, and without requiring any prologue/epilogue signatures. This has several benefits: it allows *Nucleus* to be compiler-agnostic, detect non-contiguous functions, and find unreachable or indirectly called functions which are missed by signature-based approaches. The tradeoff is that *Nucleus* requires switch analysis and address-taken analysis to correctly handle intraprocedural indirect jumps.

All of the false positives we analyzed, for ELF as well as PE binaries, are caused by inaccuracies in resolving intraprocedural indirect edges. For C binaries, this is due to unresolved switch edges, which result in isolated case blocks. When *Nucleus* finds an isolated basic block, it flags this block as a possible indirectly called function entry, thereby producing a false positive. In C++ binaries, false positives are caused by both unresolved switch edges, and unresolved exception handling edges (again leading to isolated exception handling blocks). These results show that more sophisticated switch detection and exception handling detection, explored in related work [13], [20], [22], [23], could reduce the false positive rate in *Nucleus*.

Listing 2 False negative due to tailcall.

```
00000000005daf10 <rli_size_so_far>:
  5daf10: 48 8b 47 08     mov rax,[rdi+0x08]
  5daf14: 48 8b 77 18     mov rsi,[rdi+0x18]
  5daf18: 48 89 c7        mov rdi,rax
  5daf1b: e9 50 fc ff ff  jmp 5dab70 <bit_from_pos>
```

5.3.2. False negatives. False negatives in *Nucleus*, for both function start and function boundary detection, are caused almost exclusively by tailcalls. In the random sample we analyzed, tailcalls are responsible for 96% of false negatives. An example of a tailcall causing a false negative is shown in Listing 2.

In a tailcall, a function (`0x5daf10` in Listing 2) ends with a `jmp` to another function (`0x5dab70`). This is an optimization frequently used by compilers—instead of inserting a `call` instruction at the end of a function, the compiler instead uses a `jmp` to remove the need for two subsequent `ret` instructions. Recall from Section 3 that *Nucleus* starts by looking for functions that are directly called, and then expands these by following control flow edges. Function `0x5dab70` is never reached by a direct call, and is therefore not found in this first phase. However, `0x5daf10` *is* called directly. When expanding function `0x5daf10`, *Nucleus* follows the tailcall edge to `0x5dab70`, merging the two functions and producing a false negative.

This produces false negatives *only* if the tailcalled function (the tailcallee) is never called directly. If it is, then it is found in the first analysis phase, and the problem does not occur. We have also seen cases where the tailcallee is called directly (by another function than the tailcalling function), but the tailcaller is never called directly. To prevent function merging in these cases, *Nucleus* does not expand functions along inbound edges to directly called entry points (i.e., for directly called basic blocks the connected components analysis is directed rather than undirected).

Tailcalls are the main cause of false negatives in both function start and function boundary detection. For function boundary detection, a single tailcall can cause *two* false negatives: (1) a wrong start address for the tailcallee, and (2) a wrong end address for the tailcaller.

In some cases, merged functions are closely related, and the function performing the tailcall is merely a stub that sets up a parameter and performs the tailcall. *Nucleus* classifies such cases as multi-entry functions. Arguably, this could be considered correct. However, we count such cases as false negatives because the symbolic information specifies the merged functions as separate.

In most cases, the tailcaller and tailcallee are in distinct memory ranges. As such, extending *Nucleus* with the assumption that functions are contiguous in memory could remove these false negatives on platforms where this assumption holds. In this work, we chose not to add this assumption to *Nucleus*, as our aim is to show that *Nucleus* achieves accurate function detection *without* such assumptions.

As *Nucleus* does not currently implement detection for non-returning functions, it must assume that a `call` to such a function can return normally. This can cause false negatives, if a `call` to a non-returning function directly precedes another function that is itself never called. We show an example in Listing 3. Here, there is a `call` instruction at address `0x44a37a` that targets a non-returning function (`fancy_abort`). Directly after the `call` is the start of another function, at `0x44a380`. Since this function is never called directly, it is merged into the preceding function through the fallthrough edge from the `call`. This case is responsible for 4% of false negatives in our analysis, and again occurs only if the function after the non-returning `call` is itself never called from any direct call site.

Listing 3 False negative due to fallthrough from non-returning `call`.

```
  44a36b:  mov   edi,0x628882
  44a370:  mov   esi,0x213
  44a375:  mov   edx,0x62888e
  44a37a:  call  47ce90 <fancy_abort>
  44a37f:  nop

000000000044a380 <cfg_layout_initialize>:
  44a380:  push  rax
  44a381:  mov   edi,0x20
  44a386:  call  444970 <alloc_aux_for_blocks>
```

5.4. Runtime Performance

Figure 4 compares the runtime performance for *Nucleus* to other approaches. The measured runtimes include not only the function detection phases, but also the disassembly and (I)CFG analysis phases of all compared tools. *Nucleus* is among the fastest of the compared approaches, providing runtime performance comparable with Dyninst, and completing all of its analysis in under 20 seconds even for binaries with code sections in the order of 1×10^6 instructions. Both *Nucleus* and Dyninst scale roughly linearly. IDA Pro performs a more extensive analysis phase for each binary, and therefore requires 45 seconds to process the largest binary. Note that for BAP/ByteWeight, we were forced to exclude the largest binary (`xalancbmk` at O3) from our tests due to scalability issues, which can be observed from the steep increase in runtime for BAP/ByteWeight for increasing binary size.

6. Analysis of Machine Learning in Function Detection

Several recent papers have investigated the use of machine learning techniques to automatically learn signatures for function recognition. Bao et al. use a machine learning system (ByteWeight) to construct a weighted prefix tree of known code sequences that delineate functions [11], while Shin et al. train Neural Networks to recognize functions [12]. Both of these papers report extremely accurate results.

Unfortunately, Shin et al. have not released an open-source version of their system, preventing us from directly

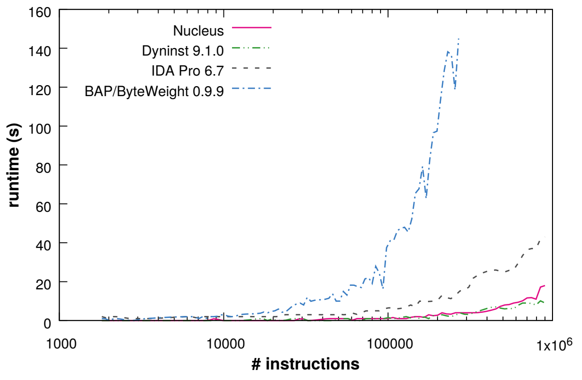

Figure 4: Runtime performance for function boundary detection. The x-axis (number of instructions) is logarithmic.

comparing it with *Nucleus*. ByteWeight is available open-source, and is used for function detection in recent BAP versions [10]. The results for our tests with this version of ByteWeight (which we refer to as BAP/ByteWeight) are reported in Section 5.2. As described there, we were unable to reproduce the performance reported by Bao et al. for this ByteWeight version. For instance, our BAP/ByteWeight tests with `gcc` on `x64` produced a mean F-score for function start detection of only 0.65, which is 0.32 points lower than the result presented in the original ByteWeight paper.

As mentioned in Section 5.2.1, we additionally requested the trained ByteWeight version tested in the original paper by Bao et al., in order to run it on our own test suite. Unfortunately, the authors were unable to provide us with this version of ByteWeight, as they did not retain the trained ByteWeight version used for their tests. We therefore provide an additional comparison of *Nucleus* against the results as presented by Bao et al. and Shin et al. in their respective papers (Section 6.1). Subsequently, we provide an in-depth analysis of the reasons for the diminished performance we observed in BAP/ByteWeight (Section 6.2). This analysis reveals inadvertent methodological errors in the evaluations of both Bao et al. and Shin et al., which cause a strong bias in the test suite they used for evaluation. This bias provides a likely explanation for the observed performance discrepancy.

6.1. Function Detection Performance

Table 3 compares the function detection results achieved in *Nucleus* to those presented by Bao et al. and Shin et al. Both machine learning papers use the same test suite, which consists of `coreutils`, `binutils` and `findutils`, and a number of Windows applications (see Section 6.2). We repeated our experiments with *Nucleus* for this same test suite, the only difference being that *Nucleus* is evaluated on `gcc`, `clang`, and Visual Studio, while both Bao et al. and Shin et al. evaluated on `gcc`, `icc` and Visual Studio.

The table shows that *Nucleus* achieves F-scores comparable to those presented by Bao et al. and Shin et al.

	Start			Boundary		
	p	*r*	*F*	*p*	*r*	*F*
ByteWeight [11]§	0.97	0.97	0.97	0.93	0.93	0.93
Neural Nets [12]§	0.99	0.99	0.99	0.97	0.94	0.95
Nucleus†	0.96	0.94	0.95	0.96	0.88	0.92

TABLE 3: Precision/recall/F-scores for function start and boundary detection (average scores for the test suite of Bao et al. [11]). † *Nucleus* results are for `gcc`, `clang` and Visual Studio. § Bao et al. [11] and Shin et al. [12] results are for `gcc`, `icc` and Visual Studio.

For function start detection, the precision, recall and F-scores for the different approaches are within 0.05 points from each other. The function boundary detection scores are also comparable—*Nucleus* achieves higher precision than ByteWeight (fewer false positives), though with slightly lower recall (more false negatives). The overall F-scores are within 0.03 points from each other.

Though *Nucleus* performs well on the test suite used by Bao et al. and Shin et al., we opted to use our own test suite for our main evaluation. The reasons for this choice are explained in Section 6.2, which provides an in-depth analysis of the differences between our test suite and the tests done by Bao et al. and Shin et al.

6.2. Evaluation Methodology

As mentioned in Section 6, we observed a large discrepancy in the results achieved by ByteWeight on our own SPEC-based test suite compared to the results reported in the ByteWeight paper [11]. The mean F-score was 0.32 points lower than expected, and this observation persisted across `gcc`, `clang` and Visual Studio. It also persisted across different versions of `gcc`, ranging from version 4.7 (used in the original ByteWeight evaluation) to version 5.1.1.

Upon closer inspection of the test suite used by Bao et al. to evaluate ByteWeight, we found that it contains many binaries with large amounts of common functions. In the remainder of this section, we show that this leads to a large bias in the results reported by Bao et al., due to a significant overlap between training set and test set. This is problematic, because ByteWeight is a machine learning approach, and thus the validity of its evaluation relies on a strong separation between training and test binaries. Moreover, the exact same problem occurs in the Neural Network-based approach by Shin et al., as they used the same evaluation test suite as Bao et al. for their own evaluation.

6.2.1. Test Suite for ELF Binaries. Bao et al. build their ELF test suite (for `gcc` and `icc`) from three popular open-source binary suites: `coreutils`, `binutils` and `findutils`. These contain 106, 16 and 7 binaries, respectively. Though all of these binary suites contain large amounts of shared code, we focus here on `coreutils`, as it comprises the majority of the Linux test suite. We perform our analysis on the binaries as compiled and used by Bao et al., which they make available online.[4] We focus our discussion here on the binaries compiled at optimization level O0, but we verified that the same effects occur at all optimization levels up to O3.

The `coreutils` binaries, as compiled by Bao et al. with `gcc` at O0, contain 1839 unique functions, distributed over 106 binaries (excluding PLT stubs and commonly named functions like `main`). There are 102 functions which occur in at least 90% of these binaries—mostly utility functions such as `xmalloc` and `quotearg`. We took a random subset of 50 such functions, comparing 2 randomly selected binaries for each function. In each case, the function body was shared *verbatim* between binaries, the only difference being in code addresses (which are normalized by ByteWeight).

Moreover, 87 functions occur in *all* binaries.[5] Since the average `coreutils` binary has 160 functions, this means that for the average binary, if selected for the test set, 54% of its functions are *guaranteed* to occur in the training set. The three binaries with the most functions are `mv`, `ginstall` and `vdir` (388, 358 and 355 functions, respectively). Thus, even these binaries share nearly 25% of their functions with *all* other `coreutils` binaries. The largest degree of overlap is found in `true` and `false`; 94% of their functions are guaranteed to occur in the training set. In contrast, the average binary in our own SPEC-based test suite contains less than 1% of such shared functions (the ones that are present are bootstrap functions such as `_start`).

The average `coreutils` binary shares 94% of its functions with *at least* one other binary in the test suite. This is because many `coreutils` binaries are extremely simple, often having only a `main` and `usage` function in addition to the shared utility functions.

Both Bao et al. and Shin et al. use 10-fold cross validation in their evaluations. This means that the set of binaries B is divided into two sets B_E and B_T, such that $B_E \cup B_T = B$. B_T consists of 90% of the binaries, and is used for training the system. The trained system is then evaluated on B_E, which contains the remaining 10% of binaries. This is repeated 10 times, such that each binary occurs in B_E exactly once.

To determine the precise probability of overlap between binaries in B_E and B_T, let b_f and c_f be two binaries that share the same function f. b_f has an $11/106$ probability of being chosen for B_E. Supposing that $b_f \in B_E$, c_f will be in B_T with probability $95/105 \approx 0.91$. Given that the average `coreutils` binary shares 94% of its functions with at least one other binary, this means that for the average binary in B_E a fraction of *at least* $0.94 \times 0.91 \approx 0.86$ (86%) of its functions are expected to occur in B_T.

6.2.2. Test Suite for PE Binaries. A similar situation occurs in the PE test suite used by both Bao et al. and Shin et al. for testing Visual Studio. For space reasons, we simply report the number of related binaries in the PE suite, rather than repeating the argument made for the ELF tests.

4. http://security.ece.cmu.edu/byteweight/

5. Except `make-prime-list`, which shares less code than other `coreutils` binaries

The PE test suite contains a total of 17 applications, from 7 open-source projects: `putty`, `7zip`, `vim`, `libsodium`, `libetpan`, `HID API`, and `pbc`. Out of these, 7 applications belong to the `putty` project: `pageant`, `plink`, `pscp`, `psftp`, `putty`, `puttygen` and `puttytel`. All of these share a common code base. Related applications are also found for the `7zip` project (3 related), `vim` (2 related) and `libetpan` (2 related). Overall, only 3 of the applications in the PE test suite do not have a relative that also occurs in the test suite.

In summary, it is clear that both the ELF and PE test suites used by Bao et al. and Shin et al. cause a strong bias in their evaluation results, preventing us from directly comparing these results to *Nucleus*. We believe this bias is the most likely explanation for the drop in accuracy when testing ByteWeight on our own test suite. Unfortunately, given this bias, the results presented by Bao et al. and Shin et al. cannot currently be assumed to generalize. Thus, further research in this area is needed to reassess the viability of machine learning for function detection.

7. Discussion

Previous work has shown that in existing approaches, function detection is among the most compiler-specific and error-prone stages of the binary analysis process. To the best of our knowledge, *Nucleus* is the first approach which shows that accurate function detection can be achieved in a completely compiler-agnostic way, with significantly fewer false positives and false negatives than existing work. This enables function detection for binaries compiled with new or unknown compilers, and eliminates the need for maintaining signature databases.

We show in Section 6 that existing work, which aims to reduce maintenance costs through machine learning [11], [12], suffers from a significant evaluation bias due to overlapping training and test sets. In principle, it should be possible for these approaches to match the accuracy of other signature-based approaches, such as IDA Pro. Unfortunately, the question of whether or not they can exceed this accuracy remains to be answered in future work. While machine learning approaches do succeed in their aim to reduce manual maintenance, *Nucleus* eliminates maintenance completely, while achieving higher accuracy than any of the other approaches that we tested.

To demonstrate the generality of *Nucleus*, in this work we have limited our assumptions on function structure to a minimum. We assume only that intraprocedural control transfers follow a different general pattern than interprocedural control flow. In contrast, existing work, including machine learning approaches, inherently relies on compiler-specific function prologue and epilogue patterns [11]–[13], which are not always present at high optimization levels.

Section 5.3 shows that most false negatives in *Nucleus* (resulting from tailcalls) can be eliminated if it can be assumed that functions are laid out contiguously in memory. Although we opted not to make this assumption in this paper, the open source version of *Nucleus* contains a command-line option to enable this assumption when it is known that functions are contiguous. We stress that this feature is strictly optional; it is not required by *Nucleus*, and is disabled in all tests presented in this paper.

Since the vast majority of false positives are a result of unresolved indirect intraprocedural control transfers (Section 5.3), *Nucleus* directly benefits from advances made in switch detection and reverse engineering of exception handling constructs [13], [20], [22], [23].

Though *Nucleus* does not explicitly target malware or obfuscated binaries, its lack of assumptions on low-level code structure enable *Nucleus* to handle some common types of obfuscations more accurately than signature-based work. For instance, *Nucleus* is agnostic to instruction-level polymorphism, and to obfuscators that intentionally rewrite function prologues and epilogues to non-standard variants [24]. Additionally, *Nucleus* provides inherent support for finding indirectly called functions, making it immune to obfuscations that obscure function calls by using branching functions, or transforming direct calls to indirect calls [25], [26]. We chose not to evaluate these aspects of *Nucleus*, due to the large variety of obfuscation techniques used in practice and the lack of ground truth for malware samples. Instead, we defer proper handling of obfuscated malware to related work on deobfuscation, which can be used in unison with *Nucleus* [24], [26].

For compilers which emit highly predictable function patterns, our results show that traditional signature-based approaches perform extremely well (Section 5.2.1). In our tests, this was only the case for Visual Studio on x64, where IDA Pro took full advantage of the predictability in calling convention. However, as our results also show, the majority of compilers use a variety of calling conventions and function prologue/epilogue patterns, causing a decline in the accuracy of signature-based approaches. In all these cases, *Nucleus* provides significantly higher accuracy.

8. Related Work

Previous work has used machine learning to automatically generate signature databases, reducing maintenance costs by eliminating the need for manually crafted signatures [11], [12]. However, these approaches still require a learning phase for every new compiler version, and cannot handle unknown compilers. In contrast, *Nucleus* is completely compiler-agnostic, and is a zero-maintenance approach.

Concurrent work has explored graph-based function detection for the ARM Thumb architecture [27]. While this graph-based approach shares some principles with *Nucleus*, it operates from different assumptions, such as the assumption that functions are laid out contiguously in memory.

Signature-based function detection is currently used in all major disassemblers [2], [10], [13], [20], [28]. Several papers have measured the accuracy of binary analysis and disassembly techniques, finding function detection to be significantly more inaccurate than other primitives such as instruction or CFG recovery [9], [29]. At the same time, function detection is a widely used primitive in binary analysis, ranging from binary-level Control Flow Integrity [5], [7], [8], [23], [30], [31] to automatic vulnerability detection [3], [4], binary instrumentation [1], [2], and manual binary analysis [15], [16]. Thus, our results for *Nucleus* facilitate work in a large range of binary analysis applications.

Our approach to disassembly is based on linear disassembly with fault correction. Similar approaches have been explored in the context of high-coverage Control Flow Integrity [30], deobfuscation [26], and binary instrumentation [1], [32]. In all these cases, linear disassembly results have proven extremely accurate, a finding confirmed in recent work on disassembly accuracy [9].

9. Conclusion

This work has shown that compiler-agnostic function detection can achieve high accuracy. We have shown that *Nucleus*, our function detection approach, provides significantly more accurate results than existing approaches in terms of both function start and function boundary detection, without making any compiler-specific assumptions. *Nucleus* provides inherent handling of complex cases such as non-contiguous and multi-entry functions, and functions with unknown prologues or epilogues, which are not handled in current signature-based work. Moreover, we have found a significant bias in the evaluations of existing approaches that aim to reduce maintenance costs for function signature databases through machine learning, showing the need for future work to reassess the viability of these approaches. In addition to achieving more accurate results than existing work, *Nucleus* is zero-maintenance, supporting new or unknown compilers without any additional effort. We provide *Nucleus* open-source, including the option to transfer results to IDA Pro, making *Nucleus* straightforward to use in real-world environments.

References

[1] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient Static Binary Instrumentation for Linux," in *ISPASS*, 2010.

[2] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *PASTE*, 2011.

[3] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *S&P*, 2015.

[4] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *NDSS*, 2016.

[5] D. Andriesse, V. van der Veen, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *CCS*, 2015.

[6] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries," in *NDSS*, 2015.

[7] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control-flow integrity and randomization for binary executables," in *S&P*, 2013.

[8] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *NDSS*, 2015.

[9] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in *USENIX Sec*, 2016.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *CAV*, 2011.

[11] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *USENIX Sec*, 2014.

[12] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing Functions in Binaries with Neural Networks," in *USENIX Sec*, 2015.

[13] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 2nd ed., 2011.

[14] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. Dietrich, and H. Bos, "P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *S&P*, 2013.

[15] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring," in *USENIX Sec*, 2013.

[16] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations," in *NDSS*, 2015.

[17] T. Koju, R. Copeland, M. Kawahito, and M. Ohara, "Reconstructing High-level Information for Language-specific Binary Reoptimization," in *CGO*, 2016.

[18] N. A. Quynh, "Capstone: Next-Gen Disassembly Framework," in *Blackhat USA*, 2014.

[19] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *WCRE*, 2002.

[20] J. Kinder, "Static Analysis of x86 Executables," Ph.D. dissertation, Technische Universität Darmstadt, 2010.

[21] Microsoft Developer Network, "Overview of x64 Calling Conventions," 2015, https://msdn.microsoft.com/en-us/library/ms235286.aspx.

[22] I. Skochinsky, "Compiler Internals: Exceptions and RTTI," in *RECON*, 2012.

[23] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables' Integrity," in *NDSS*, 2015.

[24] H. Saïdi, P. Porras, and V. Yegneswaran, "Experiences in Malware Binary Deobfuscation," 2010.

[25] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *CCS*, 2003.

[26] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *USENIX Sec*, 2004.

[27] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Speculative Disassembly of Binary Code," in *CASES*, 2016.

[28] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," 2015.

[29] B. P. Miller and X. Meng, "Binary Code is Not Easy," 2015, technical report, University of Wisconsin-Madison.

[30] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX SEC*, 2013.

[31] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *NDSS*, 2015.

[32] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A Platform for Secure Static Binary Instrumentation," in *VEE*, 2014.