Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata†

Soumyakant Priyadarshan, Huan Nguyen, R. Sekar
Stony Brook University
Stony Brook, NY, USA
{spriyadarsha,hnnguyen,sekar}@cs.stonybrook.edu

Abstract

Accurate disassembly of stripped binaries is the first and foremost step in binary analysis, instrumentation and reverse engineering. Complex instruction sets such as the x86 pose major challenges in this context because it is very difficult to distinguish between code and embedded data. To make progress, many recent approaches have either made optimistic assumptions (e.g., absence of embedded data) or relied on additional compiler-generated metadata (e.g., relocation info and/or exception handling metadata). Unfortunately, many complex binaries do contain embedded data, while lacking the additional metadata needed by these techniques. We therefore present a new approach that can accurately disassemble such binaries. Our approach combines a novel static analysis with another new, data-driven probabilistic technique in order to detect embedded data. These two techniques are combined using a prioritized error correction algorithm to achieve superior false-positive and false-negative rates on datasets used in previous work.

ACM Reference Format:
Soumyakant Priyadarshan, Huan Nguyen, R. Sekar. 2024. Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata[2] . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Binary analysis and instrumentation provide the foundation for a wide range of applications such as binary debloating [20, 46, 48], optimization [33, 38], code similarity detection [7, 11], reverse engineering [29, 31, 47], vulnerability discovery [13, 30, 42, 54, 63], malware analysis [10, 24], and security hardening [3, 14, 25, 45, 55, 57, 62]. *Disassembly* is the first step in all of these applications, and is concerned with lifting binary code (i.e., a sequence of bytes) to assembly instructions. Without additional symbolic information or metadata (as in stripped binaries), it has proven to be a very challenging problem despite numerous research efforts [6, 19, 34, 37, 41, 56–58, 62, 63].

There are two basic techniques for disassembly. *Linear sweep* [51] begins at the start of a code section in a binary and proceeds to disassemble all subsequent bytes. It can achieve high coverage but suffers from a high error rate on binaries containing embedded data. *Recursive disassembly* [51] copes better with such data, but suffers from low coverage because of the inability to decipher indirect control flow targets. State-of-art systems [19, 49, 54] combine the two techniques, using recursive disassembly in the first phase, and linear disassembly in the second phase to uncover code missed in the first phase. While many of these techniques achieve accuracies between 99% and 99.99% [39] on binaries that don't contain any data, their accuracy falls rapidly on some of the more complex binaries such as `openssl` that do contain such data. *In contrast, we develop new statistical and static analysis techniques and a conflict resolution algorithm that combines them to achieve high accuracies (99+%) on data-containing binaries, while approaching the ideal of 0% errors on data-free binaries.*

Existing tools such as Dyninst [23], Ghidra [49], and Angr [54] can take advantage of symbol and debugging information to improve disassembly accuracy. Unfortunately, such information is absent in most COTS binaries. *C++ exception handling* metadata is present in most stripped binaries on Linux, and researchers have shown that the accuracy of disassembly and related tasks can be improved using this information [40, 44]. However, this metadata is very large, causing some projects such as Chrome to disable it. Moreover, this metadata may be missing for C applications on platforms other than Linux. Finally, even when it is present, it may not cover all of the code, e.g., we find that exception handling metadata is missing for 1% of Firefox's code. For these reasons, we focus on an approach that doesn't require any compiler-generated metadata. *In addition to broadening applicability, a metadata-free emphasis enables us to explore the limits of what is achievable using static analysis and statistical techniques for disassembly.*

1.1 Approach overview and contributions

In this paper, we present a novel disassembly approach that combines static analysis and statistical techniques using a prioritized conflict resolution algorithm to achieve high accuracy on complex binaries without relying on compiler metadata. An interesting aspect of our approach is that we use ***properties of data to flag code*** and ***properties of code to flag data.*** Since data tends to be more uniform and random, it's *statistical properties* are more readily quantified. In contrast, code is highly variable in terms of opcode or operand values used. For this reason, we do not attempt to characterize the statistical properties of code, but instead, we flag a byte sequence as code whenever its statistical properties deviate drastically from that of data.

While the statistical properties of code are variable, its behavior is more constrained. In particular, we focus on constraints that code

†This manuscript is not camera ready version and is subject to minor revision. This work was supported by an ONR grant N00014-17-1-2891 and in part by NSF grants 1918667 and 2153056

must satisfy in order to run successfully and interoperate with other code. Our research identifies some of these constraints and develops *new static analyses* to determine conformance to them. Clearly, data is not meant to be executed and hence won't satisfy these properties. Consequently, we can flag byte sequences as data when they violate these properties.

Due to the high error rate of linear sweep disassembly on binaries with data, the first phase in our approach uses recursive disassembly. Unfortunately, as explained in Sec. 2, recursive disassembly misses a significant fraction of code, e.g., code reachable only through indirect branches, or code that is altogether unreachable.[1] Worse, because of the prevalence of non-returning functions, conventional implementations of recursive disassembly suffer from false positives as well. Previous work has tackled this challenge by constructing a catalog of all non-returning functions and avoiding the disassembly of follow-through code. Unfortunately, compiling such a catalog can be a lot of manual effort, and moreover, these catalogs will need to be updated constantly, as new versions of libraries are distributed. We, therefore, rely on a conservative static analysis for constructing the catalog, but the downside of such an approach is the significant overestimation of non-returning calls. This overestimation further exacerbates the false negatives of recursive disassembly. Consequently, less than half the code is discovered in this phase of our approach.

The second phase of our disassembly uses speculative techniques to identify all code. Specifically, (a) any constant value appearing within code or data sections of the binary is interpreted as a code pointer and the target is disassembled, and (b) all gaps remaining are disassembled. Such speculative steps lead to significant disassembly errors, so we develop new techniques for scoring, validation, and prioritized error correction to overcome these errors. First, disassembled snippets are assigned a score based on *statistical properties of data* as described in Sec. 3. Snippets are then processed in decreasing order of scores by our *prioritized error correction algorithm* described in Sec. 5.

Although our statistical scoring is very effective in identifying snippets that are mostly code, it is unable to pinpoint the start or end of code within a disassembled snippet. We develop *static analysis* techniques in Sec. 4 for this purpose. In particular, we identify code properties whose violation will lead to crashes or other serious problems. Any snippet that violates these properties is then flagged as data and discarded. In addition to validating snippets that are speculatively disassembled, these *invalid code properties* also serve as the basis for our solution to two other challenges: (i) non-returning functions, and (ii) accurate determination of jump table bounds. In summary, we make the following contributions in this paper:

- This paper develops a set of code properties that can accurately flag invalid code. We present a scalable static analysis technique for computing these properties.
- We present a method for identifying code using statistical properties of data and provide empirical validation for the probability estimates derived by this method.
- Although both of the above techniques have accuracy, they still have non-negligible error rates. In order to further reduce these errors, we present a *prioritized error correction* algorithm that combines these techniques to achieve an overall error rate that is near zero.
- We perform a comprehensive evaluation of our disassembly approach on two sets of benchmarks. One set was borrowed from Pang et al.'s work [39] and comprising of real-world binaries and SPEC benchmarks compiled with GCC and LLVM [27]. Another set of benchmarks was borrowed from StochFuzz [63], which includes a significant amount of embedded data within code.
- We compare our results with other state-of-the-art disassemblers, including Angr [54], Ghidra [49], Dyninst[23] and DDisasm [19]. We achieve error rates that are 3× to 4× lower than that of best among previous disassemblers.

Our system, along with the datasets and measurement procedures used, is available at http://seclab.cs.sunysb.edu/soumyakant/safer.

2 Challenges in disassembling complex binaries

Linear disassembly such as Objdump [21] can discover all code but suffers from a high false positive rate on binaries containing embedded data. *Recursive disassembly* [23, 49, 54] follows the control flow present in the code, and hence can skip regions that contain data. But it suffers from high false negatives because it cannot discover code that is reachable only via indirect branches. On average, recursive disassembly can miss 20% of the code [39], but the miss can be substantially larger in some programs.

Achieving high accuracy requires reducing both the false negatives and false positives. For this reason, many contemporary works [19, 23, 54, 56] combine recursive disassembly with other speculative techniques such as linearly scanning binary regions missed by recursive disassembly. Unfortunately, the increased coverage of speculative techniques comes with a high rate of disassembly errors. To reduce the scope for these errors, it is necessary to understand the limitations of recursive disassembly, and narrowly tailor speculative techniques to target these limitations. For this reason, we focus the rest of this section on the challenges faced by recursive disassembly.

Indirectly reached code: This is a well-established reason for the low coverage (i.e., high false negatives) of recursive disassembly. High-level programming language constructs such as virtual functions and switch-case statements get translated into indirect control flow transfers in binary, with targets known only at runtime. Indirectly reached code can be categorized into two types: (i) functions whose addresses are taken and stored in registers or memory, and (ii) jump table targets. (Jump tables result typically from the translation of switch statements.)

The use of relocation information to discover address-taken functions has been explored in recent works [16, 60, 61]. However, relocation information is not always available. Furthermore, it does not help in differentiating code and data. Other techniques speculatively scan for stored constants that fall within the range of code sections [62]. This discovers all address-taken functions, but introduces false positives due to possible confusion of integers and pointers.

Unlike address-taken functions, jump table targets are computed at runtime and hence require a different approach for identification. Recent techniques [16, 19, 23, 54, 56, 60, 62] use pattern matching to detect any arithmetic computation that resembles jump table target

[1]While unreachable code may not be of interest in some applications such as binary instrumentation, others such as reverse engineering, binary differencing, and malware analysis require all code to be disassembled.

computation. As shown by the evaluations in [39], Dyninst's jump table identification achieves 99% accuracy. We follow a similar approach of jump table detection. However, estimating the size/bounds of jump tables can be hard. Recent techniques overestimate the size of jump tables whenever they are unable to accurately predict the size. Pang et al. [39] make an observation that overestimated jump tables account for about 6% of Ghidra's false positives and 24% of Dyninst's false positives.

In this paper, we avoid the dependence on compiler metadata or heuristics for reducing these errors. Instead, we rely on our *valid behavior* checks and *statistical property* based error correction to remove spurious code pointers.

Non-returning calls: In traditional recursive disassembly, fall-through targets of all calls are generally considered as valid code locations. However, there exist special calls that never return, e.g., calls to standard library functions such as exit, abort, etc. The compiler may place data or intra-function padding bytes after non-returning calls. As a result, disassebly of fall-through can result in an error. Although recent works [9, 23, 49, 54] have tried to handle non-returning calls in an ad-hoc manner by manually listing standard library functions that do not return, a systematic investigation of this problem is lacking. According to Pang et al. [39], 40% of Dyninst's false positives result from the failure to identify non-returning calls. We present a systematic approach to this problem that relies on our code properties to validate fall-through code.

Unreachable code: Programs tend to have unused functions that are never called either directly or indirectly. Qiao et al. [47] estimate that about 15% of functions in binaries fall into this category of unreachable function. Since these functions are never referred anywhere, there is no evidence regarding the entry point of such functions, thereby forcing us to use speculative techniques such as linear sweep of code gaps. In presence of embedded data, such speculative approaches can result in a high error rate. However, our *valid behavior* checks help us to have significantly lower error rates while speculatively disassembling code gaps.

3 Identifying code using (statistical) data properties

Binary code is statistically different from data, and this fact has been used in previous works to improve disassembly accuracy [59]. Heuristics built into many disassembly tools, such as the use of function prologs to detect function starts (or "gaps" in disassembly), are based on such differences. However, they are used in an ad-hoc fashion, without a rigorous statistical underpinning. Machine-learning techniques [5, 41, 53] do not suffer from this problem, but they require large amounts of binary code for training. Moreover, machine learning techniques are generally opaque, making it hard to judge whether the features selected by these systems are meaningfully related to the underlying properties of code. As a result, it is difficult to gain confidence that accuracy results will carry over across different datasets. For instance, the accuracy of a machine-learning based function identification technique [5] fell from over 90% reported in the paper to just 60% on a second dataset [2]. This was because of an unrecognized bias in the training data that boosted accuracy in the original dataset.

Probabilistic disassembly [34] sidesteps the challenges of both training-based and heuristics-based techniques by *focusing on properties of data rather than code.* The central assumption is that ***data bytes are uniformly randomly distributed.*** Based on this assumption, they derive probabilities of observing certain byte sequences in data that match common control flow and dataflow patterns in code. If the probability is low, then the occurrence of that pattern in a snippet suggests that the snippet is very likely code. Miller et al [34] suggested two control flow properties — converging short jumps (two jumps that target the same location) and crossing short jumps (one short jump targeting the instruction that immediately follows another short jump); and one dataflow property — register define-use (i.e., a register assigned by one instruction is used in a subsequent instruction). For each pattern, they derived the probability that it would occur (by chance) in data. These data probabilities are propagated along control flow paths. When multiple patterns are found along the same control flow, their probabilities are multiplied. This compounding effect causes data probabilities to shrink rapidly, leading to most of the code in the binary to be recognized as such.

Probabilistic disassembly incorporates an elegant design based on solid foundations. In their original paper [34], they reported zero false negatives and 6.8% false positives. In subsequent work by the same group [63], 11.74% false negatives and 1.48% false positives were reported on a more complex dataset. These numbers are not competitive with the state-of-the-art. Based on the description in their paper and our experience with our implementation, we believe that their error rate stems from the following factors:

- *Choice of patterns and associated probabilities:* The probability of random bytes exhibiting a dataflow relationship is 1/2, which is much higher than the 1/16 they use in their calculations. (See Appendix A for details.) In addition, probabilities yielded by their analysis of short jumps are much higher than what can be derived for longer jumps. By focusing on long jumps, we show that accuracies can be significantly improved.
- *Uniform distribution assumption:* Although it seems to be a reasonable assumption, data bytes are not uniformly distributed. In fact, we find that about 30% of the data bytes are zeroes. If this is not taken into account, it leads to massive overestimations of probabilities, degrading accuracy. (See the difference between the second and third columns in Table 2, especially rows 2 and 3.)
- *Compounding propagation of probabilities:* The multiplicative rule for probabilities, although natural, causes larger snippets to accumulate very low probabilities of being classified as data, thereby leading to high false positives.

We describe a new approach below that overcomes these challenges. Similar to probabilistic disassembly, it avoids reliance on statistical properties of code because it can vary substantially across different binaries. At the same time, we avoid the uniform data distribution assumption and instead use simple statistical properties that are computed empirically. Our results show that these simple statistical properties of data tend to be stable across different binaries. In particular, Fig. 1 shows the distribution of byte values in data across three applications. Note that "data" in this regard includes both embedded data, as well as byte positions in code that do not correspond to an instruction beginning (as per ground truth).

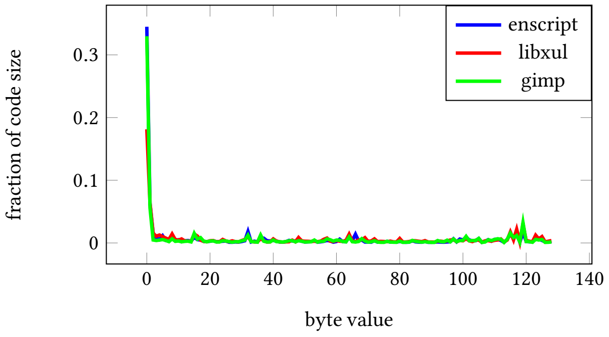


Fig. 1: Byte distribution across binaries.

Instruction pattern	Calculated probability		Experimental value
	Based on Uniform distribution	Based on Fig. 1 distribution	
SJ	2^{-4}	2^{-4}	2^{-5}
LU	2^{-17}	2^{-10}	2^{-11}
LC	2^{-22}	2^{-15}	2^{-16}

Table 2: Probability of observing control flow transfer (CFTs) instruction patterns within data. SJ stands for short jump instructions with 1-byte jump offset. LU and LC stand for control transfers with 4-byte offsets, with LU standing for unconditional transfers and LC for conditional transfers. Uniform distribution means every byte value occurs with a probability of $1/256$. Actual distribution refers to a simplified summary of Fig. 1, with $Pr(0) = 0.3$ and $Pr(x) = 0.7/255$ for non-zero byte values.

Table 2 shows that the probabilities estimated using the byte distribution of Fig. 1 differs greatly from that obtained using uniform distribution. In particular, note that for the last two rows, uniform distribution assumption leads to probability values that are smaller by more than 100×. The last column shows that the actual prevalence of these patterns in the ground truth of our entire dataset matches our calculation — it is within a factor of two of our calculation.

Note that we did not use the ground truth of the entire dataset in Fig. 1 — it uses a very small subset. Moreover, we only used the fact that zeroes occur with a probability of 0.3. All other values were approximated to be equal, at $0.7/255$.

We generated additional random subsets of our dataset to check if the distribution shown in Fig. 1 holds across different subsets. We found that it does — across 10 random subsets, the probability of zero varied between 0.23 and 0.32, with a roughly uniform distribution of other bytes. The 20 random subsets were such that their size was 0.1% of the total size of the dataset. The average probability of 0 across all subsets was 27% and the standard deviation was 7%.

3.1 Control Flow Transfers (CFTs)

We focus on control flow transfers for two key reasons: (a) the probability D_T of the target being data can be estimated *without knowing if the source location is code or data*; and (b) the likelihood of CFT offsets can be estimated from Fig. 1.

Regarding (a), observe that if the target *is* data, then the source "instruction" must itself correspond to data, or to an unaligned location in code. The prevalence of various byte values at these locations is shown in Fig. 1, compiled from three sample binaries that range in size from about 200KB to 190 MB. Despite these size differences, the probabilities are close across these binaries, indicating that we are relying on statistics that are stable.

As shown by these charts, byte value distributions are close to uniform, except at zero. Based on these probabilities, we now describe the calculation of $Pr[D_T|S \rightarrow T]$: the probability of a target location T being data, conditional on observing a CFT "instruction" from some source S to T. This calculation is organized into three groups below. Note that the probability C_T of the target being code is simply $1-D_T$.

Unconditional Long CFTs (LUs). There are two instructions in this group, `call` and `jmp`. Each has a 1-byte opcode plus a 4-byte offset. Recall that the target T of this transfer can be data *only if* the source is *not* a valid instruction. Thus, for the probability of these CFT opcodes, we should use the distribution from Fig. 1 that captures byte values in (i) data and (ii) at code locations *unaligned* with instruction boundaries. Since the distribution in Fig. 1 is roughly uniform at non-zero values, we take $2 \cdot (0.7/256) = 0.0055$ as the combined probability of these two opcodes.

For the CFT target T to be within the binary, the offset following the CFT must be less than the binary size B. The probability that such an offset follows the (unintended) LU instruction will be given by $B/2^{32}$ *if* the offset bytes are uniformly random. However, since zero bytes occur much more frequently in Fig. 1, we arrive at a higher value than $B/2^{32}$. To illustrate this calculation, let $B = 2^{22}$, i.e., 4MB. To stay within this range, the most significant byte of the offset should be zero, which occurs with a probability of about 0.3; the next byte should be less than 2^6, which works out to $0.3 + 0.7 \cdot (2^6/2^8) = 0.475$. The two least significant bytes can be arbitrary. Thus, the probability of such an offset is $0.3 \cdot 0.475 = 0.143$.

Combining the probability of an LU opcode with a valid offset, we arrive at the combined probability of $0.0055 \cdot 0.143 = 0.00079 \approx 2^{-10}$. If we follow the same calculation procedure but used uniform distribution assumption, then we will arrive at:

$$\frac{2}{256} \cdot \frac{1}{256} \cdot \frac{2^6}{2^8} = 2^{-17}$$

Note that the calculation based on uniform distribution yields a probability that is far lower than that based on the non-uniform distribution. Also note that non-uniform distribution derived from Fig. 1 yields a very close match to the probabilities observed in our dataset. (The average binary size in our test set is also 4MB.)

Note that although we used an average size of 4MB in our calculations in this section, in the actual implementation, probabilities can be computed from the actual binary size. This means that for smaller binaries, the observation of an LJ provides much higher confidence that the jump target is code. Even for a 4MB binary, a probability of $1-2^{-10}$ is obtained, which represents a much higher degree of confidence than the probability $1-2^{-4}$ derived for short jumps.

Long Conditional Jumps (LCs). This group consists of about 16 opcodes that are 2 bytes each, followed by a 4-byte offset. Following a procedure similar to that of LUs, the probability of occurrence of these opcodes is

$$\frac{0.7 \cdot 16}{256 \cdot 256} \cdot 0.143 \approx 2^{-15}$$

Using uniform probabilities, we arrive at the following calculation:

$$\frac{16}{256 \cdot 256} \cdot \frac{1}{256} \cdot \frac{2^6}{2^8} \approx 2^{-22}$$

Thus, D_T for LCs will be 2^{-22} and 2^{-15} with the uniform and Fig. 1 distributions respectively. Note that the experimental results are a

Instructions	Calculated probability	Experimental probability
2-byte push or 2x1-byte push	2^{-14}	2^{-10}
Stack decrement	$2^{-24.5}$	2^{-23}

Table 3: Probability of function prolog patterns

close match for the predicted probability from Fig. 1 distribution.

Short Jumps (SJs). Because of their small range, short jump offsets are almost always valid locations in code, even for the smallest binaries. Hence, D_T depends only on the probability of occurrence of the opcode bytes. There are about 16 short jumps, including several conditional and unconditional jumps. Thus, D_T for SJs is $16/2^8 = 2^{-4}$. Since offset values aren't significant factors for SJs, probability calculations under uniform and Fig. 1 distributions match in this case.

3.2 Function Prologs

Using function prolog pattern to detect function entries has been explored by many disassemblers such as Dyninst [23] and Angr [54]. However, rigid pattern matching still results in low coverage. In contrast, we treat function prologs as a statistical property. Instead of using it as a definitive evidence of function entry, we use it to prioritize code pointers and resolve conflicting disassembly.

The entry point of functions typically contains instructions to save callee-saved registers, and to allocate local variables by decrementing the stack pointer. This leads to three types of instructions in the prolog:

- *1-byte push:* 0x53 and 0x55 represent *push %rbx* and *push %rbp* respectively. The probability of finding these bytes in data is $0.7 \cdot 2^{-7}$.
- *2-byte push:* 0x41 followed by one of the bytes 0x54, 0x55, 0x56 or 0x57 used to push %r12, %r13, %r14 and %r15 respectively. The probability of finding this sequence in data is $0.7^2 2^{-14} = 2^{-15}$.
- *stack decrement:* Local variables are created at function entry using *sub %rsp, constant* instructions. First 3 bytes of such instructions are 0x48, 0x83/0x81 and 0xec, followed by an offset. The probability of finding such consecutive byte values in data is $\left(0.7/2^8\right)^3 * 2 = 2^{-24.5}$.

Since the probability associated with a single byte push instruction is low, we do not use it. Instead, we focus on prologs that contain (a) at least two 1-byte pushes or a single 2-byte push, or, (b) a single stack decrement. Table 3 summarizes the analytical probabilities calculated above, and the experimental values that we found. For(a), the calculated values is not a close match — it is off by 16×. But the last row matches closely.

3.3 Combining Evidence

Since our evidence is in the form of probabilities, a natural way to combine multiple pieces of evidence is to take the product of the corresponding probabilities. Taking a product implies that the underlying events are independent. This is not a sound assumption since patterns in data can often be repetitive. Moreover, the compounding effect of the product operation causes any large byte sequence to be assigned low probabilities of being data, thus contributing to higher false positive rates.

To overcome the above problem, we use an additive approach for accumulating evidence. Specifically, we define a (code) *score* $S(x)$ as follows:

$$S(x) = \frac{Pr[x \text{ is code}]}{Pr[x \text{ is data}]}$$

Here x denotes any of the patterns discussed above. For a byte sequence X that contains patterns $x_1,\ldots,x_n$, we define:

$$S(X) = \sum_{x \in X} S(x)$$

4 Identifying data using (invalid) code behaviors

To achieve low false negative rates, it is necessary to rely on speculative techniques such as linear disassembly, or the disassembly of targets pointed by constants found in the data or code regions of the binary. Although such speculative techniques can yield good results on binaries with no embedded data, they can lead to high false positives on more complex code that contains data. To bring the false positives down, it is necessary to develop techniques that can recognize data bytes and avoid marking them as code.

Some previous research works have tried to recognize data using pattern-matching techniques, e.g., null-terminated string data [19]. Unfortunately, it is difficult to develop a comprehensive approach that is capable of detecting all kinds of data. We, therefore, develop a novel alternative: ***identify data by detecting violation of code properties*** — properties universal to all code. We first compile a list of such properties, organized into *invalid control flow* and *invalid dataflow* properties. While control flow violations have been used in previous work, we propose new dataflow properties in this paper. We also present new scalable static analysis techniques for checking these dataflow properties.

4.1 Invalid control flow (InvCF)

Invalid byte sequences (which can't be disassembled into any valid instruction sequence) have been used by popular disassemblers to flag (some of the) data. In addition to that, we also take advantage of x86_64 system specifications. Specifically, we consider the presence of *privileged instructions* and *segment prefixes* such as CS, SS, DS, and ES as invalid [22].

Control flow transfers to invalid instruction boundaries have been used in previous works [19, 34, 62] to detect disassembly errors. The term *occlusion* has been used [34] to refer to this inconsistency. A challenge in using this criterion is that it is difficult to know whether the source or target disassembly is incorrect. However, self-occluding disassembly is always invalid, e.g., when recursive disassembly from certain location results in code that jumps to the middle of one of the instructions just disassembled. (Note that x86 assembly includes prefix instructions, e.g., the lock prefix. CFTs that target the location following the prefix are accepted by our method.)

CFT instructions that target outside of the current binary have also been used to detect disassembly errors [62]. In this case, the problem is clearly at the source, and hence this particular criterion is easier to use than the first one.

4.2 Invalid dataflow

Many properties one may expect from good high-level programs may not necessarily hold for all binary programs, e.g., absence of uninitialized reads or type errors. One possible reason is that some of these programs may have bugs. Clearly, we cannot insist on bug-free programs for disassembly. An even more important factor is that due to the inherent limitations of static analysis, an error may be flagged even when it does not exist. For instance, a certain program path may be infeasible under the conditions in which some code is invoked, but the static analysis may not be able to reason about this. Finally, even when the error is exercised, it may not have a serious impact. For these reasons, invalid dataflow should be flagged only in cases where (a) the underlying problem isn't due to programmer errors or the limitations of static analysis, and (b) the error has a serious impact.

Identifying program properties that satisfy these constraints turns out to be a challenging problem in itself. We have so far identified two property classes in this regard, and both have proved to be quite effective. The first set consists of properties that will cause a program to crash. Currently, this set consists of a single property type: the dereferencing of uninitialized pointers. The second set checks adherence to binary platform specifications, specifically, the ABI. Both property sets are highly conservative: buggy programs won't violate them, but buggy compilers may do so.

4.2.1 *Invalid pointer dereferencing (InvPtr)*

Use-before-definition is a common programming error that is flagged by compilers as well as runtime tools such as Valgrind [36]. It has also been suggested as a criterion for deciding function starts [47]. However, for the reasons noted above, we need to strengthen this condition further in order to prevent if from being triggered on potentially valid code. In particular, our analysis reports a violation only if:

- the violations are present on *every* program path,
- they aren't the result of programming errors, and
- the bug will lead to a major failure.

To satisfy the second constraint, we focus on low-level problems that can only be present in code generated by a buggy compiler. To satisfy the third constraint, we focus on pointer dereferencing. This is because an uninitialized pointer is likely to point to a random region of memory, so its dereferencing will lead to a memory protection fault. Specifically, we focus on the following errors:

- *Memory dereferencing* of an uninitialized data pointer;
- *Control flow transfer* using an uninitialized code pointer; and
- *Overwriting critical pointers* with uninitialized data, e.g., a return address, the stack points or the saved base pointer.

To reason about initialization, we must start from a known initial state. Hence we apply these checks on whole functions, relying on a conservative interpretation of the ABI (application-binary interface) specifications to determine what is initialized.

4.2.2 *Invalid callee-saved register modification (InvReg)*

The ABI [32] dictates that callee-saved registers (`r12–15`, `rbx`, `rsp`, and `rbp` on x86-64) must be preserved across function calls. If a callee intends to modify any of these registers, it must save it first, and then restore its value before returning to the caller. Callee-saved register preservation has previously been used for function identification [47]. Our main contribution here is to develop a more conservative and scalable analysis for detecting violations. In particular, our analysis flags *definite* rather than *possible* violations, and develops a linear-time analysis as opposed to previous techniques that analyzed each program path separately. (Note that the number of paths can be exponential in the size of a function, even for functions that don't contain any loops.)

4.3 Efficient static analysis for invalid dataflow

There are two key challenges in developing static analysis to support our disassembly approach. First, since our conservative recursive disassembly reaches less than 50% of the code, a majority of the code needs to be disassembled speculatively. Moreover, numerous overlapping candidates are considered as the starting points for such disassembly. As such, the total amount of code that needs to be statically analyzed can be an order of magnitude more than the binary size. For this reason, the efficiency of analysis becomes important. Secondly, we seek an analysis that has effectively zero false negatives: it should not report invalid dataflow for any legitimate function. We describe two such analysis techniques for *InvPtr* and *InvReg* below.

4.3.1 *Static analysis for InvPtr*

Like most previous work on binary analysis (e.g., value-set analysis (VSA) [4]) our analysis is based on the classic *abstract interpretation* [12] technique. During normal ("concrete") execution of a program, its variables take values over the program's input and output domains, referred to as the *concrete domain.* In abstract interpretation, variables range over a much smaller *abstract domain.* Each value in this domain represents a subset of values in the concrete domain. For instance, to reason about initialization, a 2-point abstract domain can be used: $\{\top, undef\}$. When a variable has the abstract value of $\top$, it means we know nothing about its value. In other words, $\top$ corresponds to the set of all possible concrete values. In contrast, $undef$ indicates that no value has been assigned.

Most of the speculatively disassembled code in our system corresponds to indirectly reached functions, so we use the ABI to determine which registers are undefined. Generally speaking, registers other than the argument registers and the stack pointer are considered $undef$. For memory, the region above the stack top is marked $undef$ and everything else as $\top$.

Like a normal interpreter, an abstract interpreter also executes a program, but in this execution, variable values range over abstract domains. Defining an abstract interpreter thus boils down to the specification of primitive program operations in terms of abstract values. Assignments (e.g., `mov`'s) simply propagate abstract values. Most arithmetic operations such as addition result in $undef$ if either of the arguments is $undef$.

To perform abstract interpretation, we lift assembly instructions into a low-level intermediate representation (IR) typical in compiler backends. After lifting, we identify basic blocks (BB) and construct a control-flow graph (CFG). A reverse post-order traversal of the CFG is used to determine the order of abstract execution of the BBs. At control flow merge points, the abstract state from the two branches are merged. Since our $undef$ represents a value that is *definitely undefined,* a variable is set to this value *only if* it is $undef$ on both branches. Finally, a violation is triggered if an $undef$ value is used in one of the critical contexts specified in Sec. 4.2.1.

Reverse post order traversal ensures that in non-recursive and loop-free functions, each instruction is abstractly executed at most once. This would enable a linear-time analysis of such programs. Loops and recursion are handled using an iterative approach called *fixpoint iteration,* which, unfortunately, has exponential worst-case complexity. Hence we make an approximation to speed this up, as described subsequently.

While its high-level design is a fairly direct application of abstract interpretation, a number of additional challenges need to be addressed in order to achieve the precision and scalability needed for our task. We describe these below.

Achieving high accuracy. Through experimentation, we identified a number of key steps to achieve the accuracy needed for our task, which is ~ 0 false negatives (i.e., flagging valid code as data) and sub-1% false positives. Some of these are:

- *Tracking abstract state at the granularity of bytes,* so that we can accurately reason about instructions that move data across registers of different sizes and/or memory.
- *Handling pointer escapes conservatively,* e.g., when a pointer to stack-allocated array is passed to a callee that may initialize this array.
- *Handling binary operators conservatively.* Arithmetic operations can produce defined results even if some arguments are $undef$, e.g., multiplication by 0, subtracting a register from itself, etc. Similarly, many logical operators can produce a defined result even if (some of) the arguments are $undef$.
- *Precisely capturing the ABI restrictions,* e.g., the lowest 8-bits of `rax` register may be defined at the function entry point, but not the other bits; and `xmm` registers 8 through 15. Similarly, registers `rax`, `rdx`, some `xmm` and floating point (`st`) registers are defined after a call.

Scalability. We take several steps in this regard as well:

- *Efficiently handling large abstract state.* Due to the large number of registers (including extended registers such as `xmm`) and the size of the stack, our analysis needs to maintain a large amount of abstract state. Eager propagation of the entire state after every instruction and control-flow merge point can be very expensive, so we developed a *lazy loading* approach. The idea is to start by storing the abstract state of a register (or memory location) within a BB only if that register (location) is updated in that BB. If a successor BB needs the value of a location not stored in the BB, then this BB will obtain it from its predecessor and then cache the value for future accesses.
- *Speeding up fixpoint iteration.* Fixpoint iteration normally begins with the initial approximation of $\perp$ and iterates until a fixpoint is reached. In the worst case, this can take time exponential in the number of variables. Moreover, stopping before reaching the fixpoint leads to unsound results, so one cannot improve performance by stopping in the middle. But there is an alternative, which is to start fixpoint iteration at $\top$. It can be shown that one iteration is all that is needed for reaching a fixpoint. This approach tends to overapproximate, i.e., the analysis results will lose precision. Note, however, that the primary goal of our analysis is to flag data. Incorrectly disassembled "code" rarely contains structures such as loops and hence it is not a serious concern if we lose precision in the handling of loops.[2]
- *Handling weak updates efficiently.* There are times when the stack is updated at an offset that cannot be statically computed. This is called a *weak update,* and is normally handled by marking all possible targets. But there are times when the range is too large. We have developed efficient techniques for handling many common cases where this happens.

4.3.2 Static analysis for InvReg

This analysis needs to answer the question of whether a register value at the end of a function is equal to its value at the beginning. In between, the register may undergo several types of changes, e.g., `rbp` may be decremented by some constant k, moved to `rax`, which is then pushed on the stack, popped back into `rcx`, incremented by k and then moved back to `rbp` before return. Our analysis needs to be able to answer whether `rbp` at exit equals `rbp` at entry, without knowing the initial value of `rbp`. Value set analysis (VSA) [4] is often used in binary analysis because it uses an abstract domain that has been carefully designed for tracking memory addresses as well as integer values. However, since it can only capture abstract *values,* it cannot answer this question unless `rbp`'s value is a known constant at the entry point. VSA incorporates a second component for reasoning about relationships between registers called *affine relation analysis.* In conjunction with a static single assignment transformation, this analysis can indeed be used to answer questions such as the one we seek to answer. Unfortunately, affine analysis is very expensive and does not scale beyond small programs.

An alternative approach is to use a domain that is custom-designed to express the content of abstract store in terms of the initial values of registers at the beginning of the function. Saxena et al. [50] have defined such a domain and a set of abstract operations on this domain. Points in this domain are of the form $\underline{R}+(l,h)$ where $\underline{R}$ represents the value of a register R at the entry point of a function, and l and h are integer constants. Such a point represents a range of values $[\underline{R}+l,\underline{R}+h]$. This domain is powerful enough to handle the example given above. We have developed an implementation of this domain that runs in time linear in the size of a program. We rely on some of the same optimization techniques described earlier for *InvPtr* to achieve this complexity. Fixpoint iteration is speeded up in the same way. Loss of precision is possible, but it is not a big concern because the analysis is often applied to data or incorrectly disassembled code. Even in legitimate code, register preservation relies on simple reasons such as saving and restoring a register, and not because of some complex changes that occur within a loop that is later undone in a way that requires accurate handling of the loop computations.

5 Disassembly algorithm

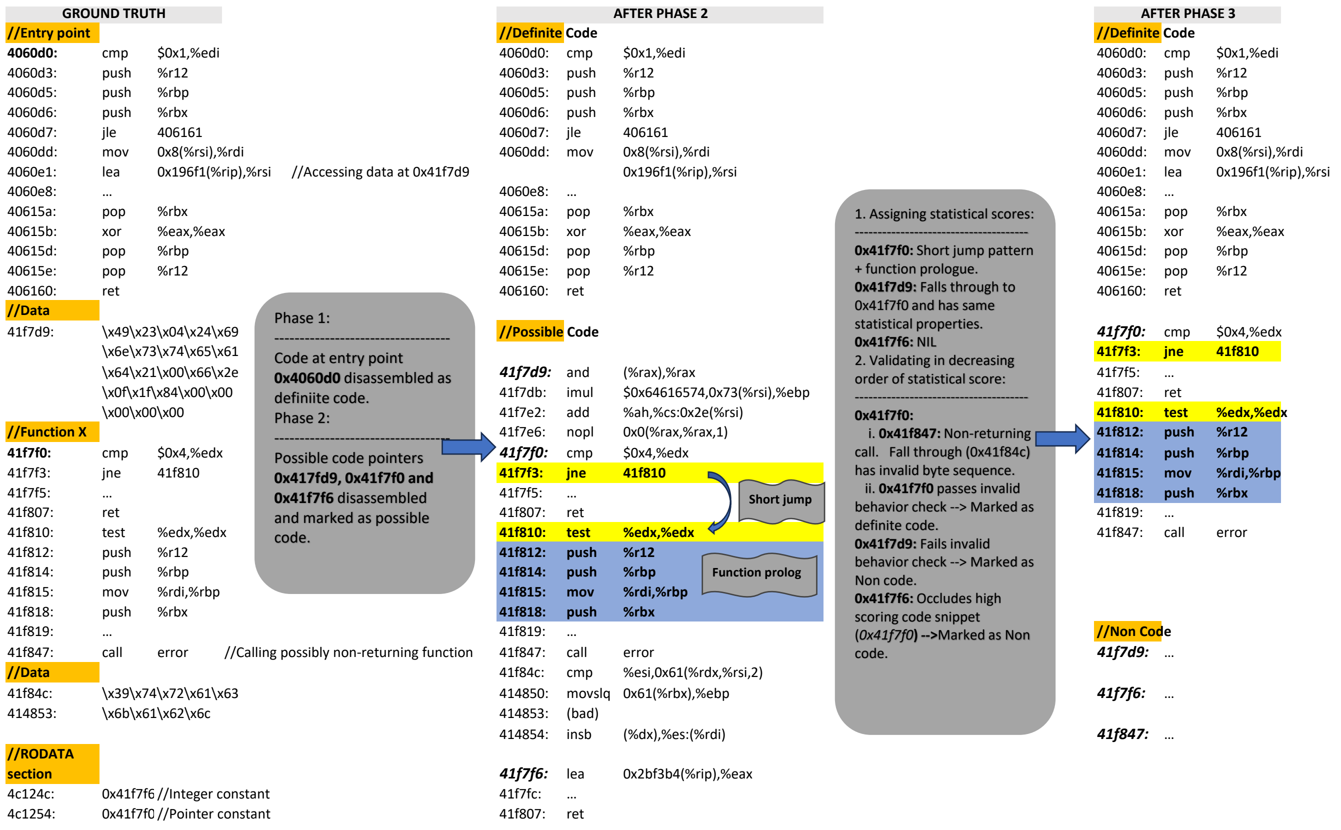
We now present our disassembly algorithm that uses the statistical and behavioral techniques from the last two sections. Figure 4 describes our approach at a high level.

5.1 Phase I: Disassembling Definite Code

The initial phase discovers *definite code.* It uses recursive disassembly in this phase, beginning with *well-known roots:*

- program entry point,
- entries in the dynamic symbol table, and
- entries of default initialization and cleanup functions.

[2]But this loss of precision will be a concern if this analysis is applied for other problems such as function identification. This is the main reason why we don't target function identification in this paper.



GROUND TRUTH

```
//Entry point
4060d0:    cmp     $0x1,%edi
4060d3:    push    %r12
4060d5:    push    %rbp
4060d6:    push    %rbx
4060d7:    jle     406161
4060dd:    mov     0x8(%rsi),%rdi
4060e1:    lea     0x196f1(%rip),%rsi    //Accessing data at 0x41f7d9
4060e8:    ...
40615a:    pop     %rbx
40615b:    xor     %eax,%eax
40615d:    pop     %rbp
40615e:    pop     %r12
406160:    ret
//Data
41f7d9:    \x49\x23\x04\x24\x69
           \x6e\x73\x74\x65\x61
           \x64\x21\x00\x66\x2e
           \x0f\x1f\x84\x00\x00
           \x00\x00\x00
//Function X
41f7f0:    cmp     $0x4,%edx
41f7f3:    jne     41f810
41f7f5:    ...
41f807:    ret
41f810:    test    %edx,%edx
41f812:    push    %r12
41f814:    push    %rbp
41f815:    mov     %rdi,%rbp
41f818:    push    %rbx
41f819:    ...
41f847:    call    error      //Calling possibly non-returning function
//Data
41f84c:    \x39\x74\x72\x61\x63
414853:    \x6b\x61\x62\x6c

//RODATA
section
4c124c:    0x41f7f6 //Integer constant
4c1254:    0x41f7f0 //Pointer constant
```

Phase 1:

Code at entry point **0x4060d0** disassembled as definiite code.

Phase 2:

Possible code pointers **0x417fd9, 0x41f7f0 and 0x41f7f6** disassembled and marked as possible code.

AFTER PHASE 2

```
//Definite Code
4060d0:   cmp     $0x1,%edi
4060d3:   push    %r12
4060d5:   push    %rbp
4060d6:   push    %rbx
4060d7:   jle     406161
4060dd:   mov     0x8(%rsi),%rdi
                  0x196f1(%rip),%rsi
4060e8:   ...
40615a:   pop     %rbx
40615b:   xor     %eax,%eax
40615d:   pop     %rbp
40615e:   pop     %r12
406160:   ret

//Possible Code

41f7d9:   and     (%rax),%rax
41f7db:   imul    $0x64616574,0x73(%rsi),%ebp
41f7e2:   add     %ah,%cs:0x2e(%rsi)
41f7e6:   nopl    0x0(%rax,%rax,1)
41f7f0:   cmp     $0x4,%edx
41f7f3:   jne     41f810
41f7f5:   ...
41f807:   ret
41f810:   test    %edx,%edx
41f812:   push    %r12
41f814:   push    %rbp
41f815:   mov     %rdi,%rbp
41f818:   push    %rbx
41f819:   ...
41f847:   call    error
41f84c:   cmp     %esi,0x61(%rdx,%rsi,2)
414850:   movslq  0x61(%rbx),%ebp
414853:   (bad)
414854:   insb    (%dx),%es:(%rdi)

41f7f6:   lea     0x2bf3b4(%rip),%eax
41f7fc:   ...
41f807:   ret
```

1. Assigning statistical scores:

0x41f7f0: Short jump pattern + function prologue.
0x41f7d9: Falls through to 0x41f7f0 and has same statistical properties.
0x41f7f6: NIL
2. Validating in decreasing order of statistical score:

0x41f7f0:
i. **0x41f847:** Non-returning call. Fall through (0x41f84c) has invalid byte sequence.
ii. **0x41f7f0** passes invalid behavior check --> Marked as definite code.
0x41f7d9: Fails invalid behavior check --> Marked as Non code.
0x41f7f6: Occludes high scoring code snippet (*0x41f7f0*) **-->**Marked as Non code.

AFTER PHASE 3

```
//Definite Code
4060d0:   cmp     $0x1,%edi
4060d3:   push    %r12
4060d5:   push    %rbp
4060d6:   push    %rbx
4060d7:   jle     406161
4060dd:   mov     0x8(%rsi),%rdi
4060e1:   lea     0x196f1(%rip),%rsi
4060e8:   ...
40615a:   pop     %rbx
40615b:   xor     %eax,%eax
40615d:   pop     %rbp
40615e:   pop     %r12
406160:   ret

41f7f0:   cmp     $0x4,%edx
41f7f3:   jne     41f810
41f7f5:   ...
41f807:   ret
41f810:   test    %edx,%edx
41f812:   push    %r12
41f814:   push    %rbp
41f815:   mov     %rdi,%rbp
41f818:   push    %rbx
41f819:   ...
41f847:   call    error

//Non Code
41f7d9:   ...

41f7f6:   ...

41f847:   ...
```

Fig. 4: High-level overview of our approach.

As these entries are needed for loading, linking, and operation of binaries, they are present in stripped binaries as well.

In the conservative version of recursive disassembly used in this phase, we don't assume that calls always return. Instead, we identify functions that *may not* return, and avoid disassembling bytes that follow calls to such functions. These bytes are explored in subsequent disassembly phases described below.

To identify non-returning calls, we begin with a small set of well-known functions that don't return, such as `exit` and `abort`. These are called *definitely non-returning functions*. Any function that calls a definitely non-returning function on all paths in its CFG is itself classified as definitely non-returning. If a function calls definitely non-returning functions on a proper subset of its code paths, it is classified as *possibly non-returning*. Finally, any function that calls a possibly non-returning function (on one or more paths in CFG) is also classified as possibly non-returning. These lists of non-returning functions can be constructed using a simple CFG analysis of all binaries on a system. However, for simplicity and expediency, we compiled these lists offline for use in our implementation.

5.2 Phase II: Disassembling Possible Code

The goal of this phase is to recover all potential code in a binary. We call the recovered code as *possible code*, in contrast with the definite code discovered in the first phase. Note that regions of "code" discovered in this phase may overlap and occlude each other. Any data within code regions will also be disassembled in this phase. These potential conflicts and errors will be resolved during a third phase described subsequently.

This phase also uses recursive disassembly but starts from a much larger set of *possible roots*. This includes:

- *Possible code pointers:* Every byte boundary within code and data sections of a binary is considered as the start of a pointer constant. If this constant falls within the code section of the binary, it is added to the set of possible roots.
- *Jump table targets:* We rely on an intra-function static analysis to discover jump tables and a (super)set of code pointers computed and used at runtime in this jump table. These code pointers are added to possible roots.
- *Locations after calls to possibly non-returning functions.*
- *Locations matching a function prolog:* Byte sequences that correspond to stack decrement or the saving of two or more callee-saved registers on the stack, are considered here.
- *Any 16-byte aligned location in the code section.*
- *Any "gap" location in the code section that has not been disassembled in previous steps.*

All the above speculative steps are applied only to the gaps left after discovering definite code.

5.3 Phase III: Prioritized Error Correction

Possible code discovered above is a superset of all valid code. The false positives in the possible code can be categorized as (i) Disassembled data (e.g., `0x41f7d9` in Fig. 4) and (ii) Conflicting code (e.g., `0x41f7f6` in Fig. 4). We now describe a conflict resolution algorithm (Fig. 5) that uses the statistical and behavioral properties discussed

```
   Input: FunctionEntries, CFG
1  Function validateCode:
2      PriorityQueue $Q$
3      for entry in FunctionEntries do
4          $Q$.push(entry, statScore(entry))
5      end
6      while ¬ $Q$.empty() do
7          entry = $Q$.top()
8          $Q$.pop()
9          if ¬ occlude(entry, CFG) then
10             resolveExitCalls($Q$, entry)
11             if ¬ invReg(entry) ∧
                 (statScore(entry) ≥ $S_{acc}$ ∨ ¬ invPtr(entry)) then
12                 CFG.add(entry)
13                 validateIndirectTargets(entry)
14             end
15         end
16     end
17 Function resolveExitCalls($Q$, entry):
18     foreach call in CFG.possiblyNoRetCalls(entry) do
19         if ¬ invCF(call.fallThrough) then
20             if ¬ invReg(call.fallThrough) then
21                 $Q$.push(call.fallThrough,
                     statScore(call.fallThrough))
22             end
23             else
24                 CFG.markRetCall(call)
25             end
26         end
27     end
28 Function validateIndirectTargets(entry):
29     foreach target in CFG.indirectTargets(entry) do
30         extCFG = CFG.extendIndirectPath(entry, target)
31         if ¬ occlude(target, CFG) ∧ ¬ invReg(extCFG, entry) then
32             CFG.add(target)
33         end
34     end
```

Fig. 5: Prioritized error correction algorithm

in Sec. 4 and 3 to select the correct code from this superset.

The top-level function in our algorithm is `validateCode` (Fig. 5, Line 1). This function first assigns statistical scores to each code region discovered in Phase II (Lines 2–5). This is done using the function `statScore`, which assigns scores as described in Sec. 3. For example, code snippet from `0x41f7d9` and `0x41f7f0` have a short jump target and function prologue consisting of one *2-byte push* (`push %r12`). Their net score will be $2^{15} + 2^4 \approx 2^{15}$. Where as, code snippet from `0x41f7f6` does not have any statistical property and will have 0 score.

Next, functions are considered in decreasing order of their scores (Lines 6–16). If a function passes the validation checks below, it is marked as *definite code*, and then we move on to the function with the next highest score, and so on. The first validation check (Line 9) is whether a function *occludes* itself or other definite code. A is said to occlude B if A includes (or transfers control to) instruction locations that are in the middle of valid instructions in B. For example, `0x41f7f6` in Fig. 4 occludes with `0x41f7f5` which is part of definite code sequence starting from `0x41f7f0`. Hence, `0x41f7f6` will be marked as non-code.

The next step in validation is to resolve potentially non-returning calls using `resolveExitCalls` defined at Lines 17–27. This function marks the call as non-returning if:

- disassembly of follow-on code leads to invalid instructions, or occlusions of definite code or the current function, or
- the follow-on code passes the valid function checks of Sec. 4, which means that it is likely an independent function.

Otherwise, the call is marked as returning, and the following code is added to the CFG of the current function. The statistical score of the function is updated to reflect this change. In the second case, the follow-on code is marked as independent function and added to the priority queue Q. In Fig. 4, potentially non-returning call at `0x414847` is marked as definitely non-returning because, fall through of this call contains invalid byte sequence at `0x414853` that cannot be disassembled into a valid instruction. As the lines 10–12 suggest, non-returning call resolution has to be done before applying any valid behavior checks on the current function. Apart from the obvious reason of avoiding false positives, this is also necessary for correct classification of the current function. We observed that in some cases compiler does not follow any standard (e.g., restoring callee-saved registers) while exiting a function via a non-returning function call (e.g., in Fig. 4 the code does not restore %r12, %rbp and %rbx before non-returning call at 0x41f847). Correctly identifying non-returning calls helps us in avoiding such non-standard cases and correctly classify true functions.

A function may have many calls marked potentially non-returning. Validation proceeds in a backward direction with calls at the bottom of control flow graph evaluated first.

After resolving non-exiting calls, the next validation step is based on the statistical score of the function. If the score is above an acceptance threshold S_{acc}, it is only subject to a subset of valid behavior checks, specifically *InvReg*. Otherwise, the full set of behavior checks are applied. Applying the behavior checks to the examples in Fig. 4, `0x41f7d9` will fail since it uses an undefined register (`%rax`) in a memory dereferencing operation. If the function passes these checks, then we proceed to the last step, namely, validating jump table targets (if present).

The function `validateIndirectTargets` (Lines 18–34) validates indirectly reached code within a function body. This validation is similar to that used for validating follow-on code after potentially non-returning calls. Disassembly and occlusion checks are first applied, followed by valid behavior checks applied to the whole function after adding the indirect target to the CFG of the current function.

A function that passes all the above checks is marked as *definite code* by `validateCode`, which goes on to deque the next function in possible code with the highest statistical score. This is repeated until Q become empty.

6 Evaluation

Our evaluation aims to answer the following questions:

(1) How effective are *invalid behavior* checks in identifying *invalid code*? (Sec 6.1.)
(2) How effective are *statistical properties* in prioritizing code over data? (Sec 6.2.)

Source	Type	Name
Pang et al.	Benchmark	SPEC CPU 2006
	Utilities	Unzip-6.0, Coreutils-8.30, 7-zip-19, Findutils-4.4, Binutils-2.26, Tiff-4.0
	Clients	Openssl-1.1.01, Putty-0.73 D8-6.4, Filezilla-3.44.2, Busybox-1.31, Protobuf-c-1, ZSH-5.7.1, VIM-8.1, XML2-2.9.8, Openssh-8.0, Git-2.23
	Servers	Lighttpd-1.4.54, MySqld-5.7.27, Nginx-1.15.0, SQLite-3.32.0
	Libraries	Glibc-2.27, libpcap-1.9.0, libv8-6.4, libtiff-4.0.10, libxml2-2.9.8, libsqlite-3.32.0, libprotobuf-c-1.3.2
StochFuzz	Google FTS	boringssl-2016-02-12, c-ares-CVE-2016-5180, freetype2-2017, guetzli-2017-3-30, harfbuzz-1.3.2, json-2017-02-12, lcms-2017-03-21, libarchive-2017-01-04, libjpeg-turbo-07-2017, libpng-1.2.56, libssh-2017-1272, libxml2-v2.9.2, llvm-libcxxabi-2017-01-27, openssl-1.0.1f, openssl-1.0.2d, openssl-1.1.0c, openthread-2018-02-27, pcre2-10.00, proj4-2017-08-14, re2-2014-12-09, sqlite-2016-11-14, vorbis-2017-12-11, woff2-2016-05-06, wpantund-2018-02-27

Table 6: Pang et al.'s[39] and StochFuzz's[63] Benchmark list

(3) How effective is the conflict resolution algorithm in combining our two techniques to achieve low error rates? (Sec 6.3.)
(4) How do our results compare with state-of-the-art disassemblers in the presence of data between code? (Sec 6.3.1.)
(5) How scalable is our static analysis? (Sec. 6.4.)
(6) How fast/scalable is the entire disassembly? (Sec. 6.5.)

Benchmarks Our disassembler was evaluated with x86_64 ELF binaries. We used two sets of benchmarks borrowed from StochFuzz[63] and Pang et al.[39]. The programs are listed in Table 6.

- Pang et al.'s benchmarks include programs and libraries written in C/C++ and containing hand-written assembly code and data in code. All the programs were compiled with GCC-8.1.0 and LLVM-6.0.0 on 6 optimization levels (O0, O1, O2, O3, Ofast and Os).
  Ground truth: Pang et al. replicated CCR[26] in customizing the compiler to record the needed information such as basic block addresses, instruction boundaries and jump tables. We reused their work to regenerate the ground truth.
- StochFuzz's benchmarks consists of binaries from Google Fuzzer Test Suite (Google FTS). The binaries have been compiled with clang-6.0 at O2 optimization level, with the compiler/linker instructed to inline read-only data within the code region.
  Ground truth: We used the symbol and debugging information available with the benchmark binaries and performed a recursive disassembly to generate the ground truth.

6.1 Effectiveness of invalid code properties

In this section, we evaluate the effectiveness of *invalid code properties* (Sec. 4) in isolation. Specifically, every snippet uncovered during Phase II of our conflict resolution algorithm was checked using these properties, and the result scored using ground truth. Table 7 summarizes the results of this evaluation. The first column shows the base results with just *InvCF* being active, i.e., a snippet is rejected only if it contains invalid instructions or a control transfer to an invalid target location. Note that this technique is able to achieve zero false negatives but comes with significant false positives.

Benchmark	Compiler	*InvCF*		*InvReg*		*InvPtr*		*All*	
		FP	FN	FP	FN	FP	FN	FP	FN
Stochfuzz	clang	4	0	3.2	0.01	1.5	0.01	1.3	0.01
Pang et al.	gcc	1.84	0	0.62	0.02	0.64	0.013	0.3	0.03
Pang et al.	clang	0.36	0	0.19	0.02	0.16	0.003	0.12	0.02

Table 7: Percentage of false positives/negatives from invalid code properties.

Benchmark	Compiler	No resolution		Conflict Resolution	
		FP	FN	FP	FN
Stochfuzz	clang	4	0	2.4	0
Pang et al.	gcc	1.84	0	0.16	0.02
Pang et al.	clang	0.36	0	0.03	0

Table 8: Effectiveness of statistical scores in prioritizing code.

The next column adds *InvReg* to *InvCF*. Recall that this amounts to checking if a code snippet preserves callee-saved registers as per the ABI. On Pang et al.'s benchmarks, this analysis removes 50% to 66% of FPs. However, *InvReg* alone is not effective on StochFuzz benchmarks where only 20% FP is reduced. This is because a large fraction of FP from StochFuzz benchmarks is due to disassembled data. It turns out that most of the recovered "instructions" don't touch any callee-saved register, and hence *InvReg* is satisfied.

The third column detects invalid behaviors that almost always cause program to crash. This property is 3× more effective compared to *InvReg* on StochFuzz benchmarks. For Pang et al.'s benchmarks *InvPtr* yields comparable results to that of *InvReg*.

The last column represents the conjunction of all properties. This is the most effective option and reduces roughly 70% FP across all benchmarks.

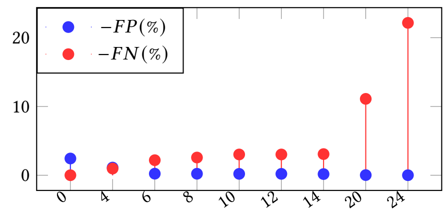
Result summary:

- The conservative nature of *invalid code properties* helps reduce FPs at a low FN rate of $\leq 0.03\%$ for our benchmarks.
- About 70% of the FPs can be reduced using this technique.

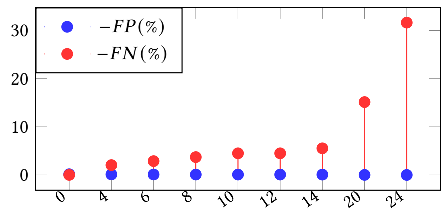
6.2 Effectiveness of statistical properties

In this section, we evaluate the effectiveness of statistical techniques on their own, without help from invalid code properties.

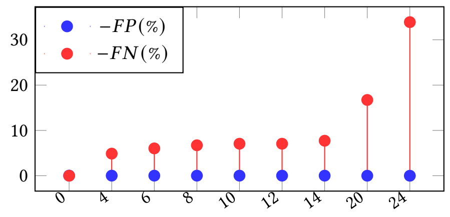
Effectiveness of statistical scores in prioritizing code. Specifically, we use our statistical techniques to assign a score to each snippet enumerated during Phase II. Then, among all snippets that have an overlap, we pick the one with the highest score, and discard the rest. The resulting disassembly is then scored using our ground truth. Note that in this experiment, we disable invalid code property checks, except *InvCF*. The "Conflict resolution" columns of Table 8 show these results. As compared to the base case where only *InvCF* is in play ("No resolution" columns), statistical scoring removes about 90% of the FPs on Pang et al.'s benchmarks, and 40% of the FPs on StochFuzz. Note that the conflict resolution introduces a minor increase in FN of 0.02%. In few exceptional cases, a false pointer quickly realigns with the subsequent function and inherits its score, thereby making its score higher than the true pointer. This forces our algorithm to accept the false pointer and reject the true one. This case is one of the underlying reasons why we chose to follow an additive approach in Section 3.3. The compounding effect of multiplying the



(a) Stochfuzz benchmarks



(b) Pang et al's gcc benchmarks



(c) Pang et al's clang benchmarks

Fig. 9: Effectiveness of statistical score threshold in eliminating false positives. A threshold T is used, and snippets with score below T are discarded as data. X-axis represents $log_2(T)$, and the Y-axis is the FP/FN rate. Note that thresholds are very effective in bringing the FP rate to zero, but the FN rate also increases.

scores would have resulted in much higher error rate in such cases.

Using a score threshold to further reduce FPs. In addition to conflict resolution, we wanted to evaluate the strength of statistical properties in identifying good code. Specifically, we wanted to see if we can achieve zero false positives using statistical properties alone. Hence, on top of conflict resolution, we added a threshold T. We reject any disassembled snippet that has a score less than the threshold. The plot in Fig. 9 shows the change in FP and FN with increasing T. Even in the presence of high volumes of data in Stochfuzz benchmarks, we obtain reasonable FP rate of below 0.5% even at a low statistical score of 2^6. The FP rate converges to 0 at $T = 2^{24}$. This shows that *statistical properties* are able to detect good code on their own. However, these thresholds also increase the false negative rate, demonstrating the need for our conflict resolution algorithm (Fig. 5).

6.3 Combining statistical with valid behavior checks

Our approach is based on the fact that code snippets with high enough statistical property score can be accepted as code pointer with high degree of confidence. Hence, if a code snippet has score $>= S_{acc}$ and preserves stack, we accept it as valid code. Everything else with lower score are subjected to all valid behavior checks (*InvReg* and *InvPtr*). Fig. 12 shows the plot of S_{acc} VS FP/FN rate. The FP rate converges to 0 at $S_{acc} = 2^{24}$ across all benchmarks. At this threshold we have the most balanced FP and FN rate. $S_{acc} = inf$, which represents the setting wherever code snippet is made to pass through all of the invalid behavior checks has comparable results to that of $S_{acc} = 2^{24}$ with a slightly higher FN rate. This is because, this setting will inherit the corresponding false negatives introduced by the valid behavior checks discussed in section 6.1. The threshold S_{acc} can be set to lower thresholds (from 2^4 to 2^{10}) to obtain a lower FN rate at reasonable FP rate. Note that even in presence of high volumes of embedded data for Stochfuzz benchmarks, we are able to achieve nearly 0 FN rate with below 1% FP rate.

Error distribution across benchmarks. In this section and the subsequent ones, we quantify error as FP% + FN% (*net error*). Traditionally F1 score has been preferred for representing effectiveness of classifiers. However, present day disassemblers approach 99% F1 score. Hence, its not the best metric for us to project effectiveness.

Our benchmark suite consists of hundreds of programs that are compiled using two different compilers, each at 6 different levels of optimization. As a result, the total number of benchmarks is over a thousand. For this reason, individual results are not shown in a table. Instead, we have divided the benchmarks into groups based on the net error, and show the fraction of benchmarks that fall into each net error group. Fig. 10 shows this histogram for the gcc compiler, while Fig. 11 shown it for the clang compiler. Both of them are for Pang et al's dataset. About 80% of clang-compiled binaries achieve perfect disassembly, and the highest net error is just 0.1%. Most gcc binaries have lower than 0.2% net error. Among the outliers at 1% to 2% net error, we observed that a significant number of them are small in size. Even error involving about 10 instructions results in ≈ 1% net error for such benchmarks.

6.3.1 *Comparing with contemporary disassemblers*

Table 13 compares net error (FP% + FN%) between state-of-the-art disassemblers and our approach at different S_{acc} values. Our net error is 3× to 4× lower than other tools. It is noteworthy that on Stochfuzz benchmarks that contain significant embedded data, net error of our approach is significantly lower than any other disassembler at any S_{acc}. Dyninst's sole reliance on function prolog matching results in low code coverage and hence high false negatives. Angr performs better than Dyninst, but net error of 10% are still quite high. Using linear scan to disassemble gaps, Angr achieves a higher coverage in comparison with Dyninst, but, at the same time, this technique also causes significantly higher FP rate in the presence

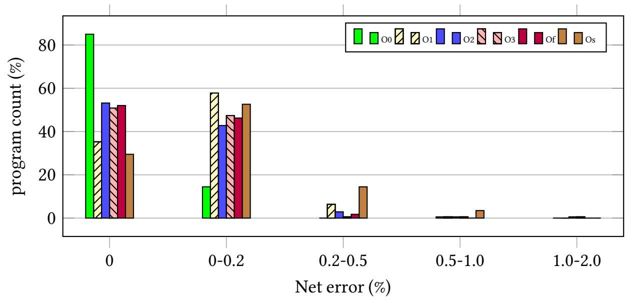


Fig. 10: Net error (FP+FN) on gcc-compiled binaries from Pang et al.

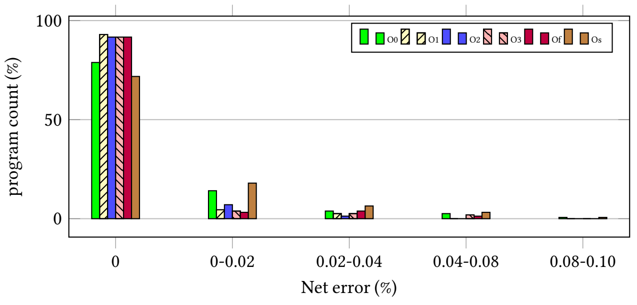
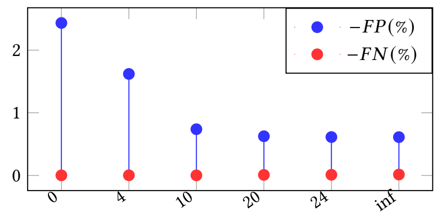
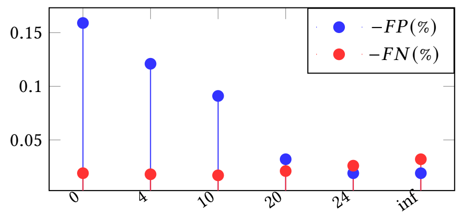


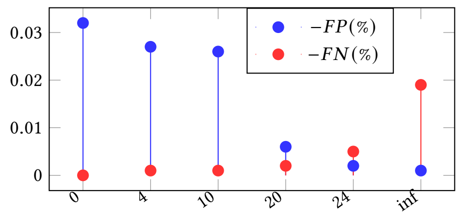
Fig. 11: Net error (FP+FN) on clang-compiled binaries from Pang et al.



(a) StochFuzz benchmarks



(b) Pang et al's gcc benchmarks



(c) Pang et al's clang benchmarks

Fig. 12: $log_2(S_{acc})$ (acceptance threshold) VS FP/FN rate for prioritized code validation with combination of statistical and valid behavior checks.

Benchmark	DDisasm	Angr	Dyninst	Ghidra	Ghidra NE	Our Approach				
						4	10	20	**24**	inf
StochFuzz	**2.43**	**10.3**	**29.7**	**2.07**	**14.77**	1.6	0.7	0.63	**0.62**	0.63
Pang et al-gcc	0.15	0.54	26.8	0.57	10.02	0.14	0.11	0.05	**0.045**	0.05
Pang et al-clang	0.02	0.71	25.2	1.52	6.91	0.028	0.027	0.008	**0.006**	0.02

Table 13: Comparison with contemporary disassemblers. The columns represent net error = FP% + FN%.

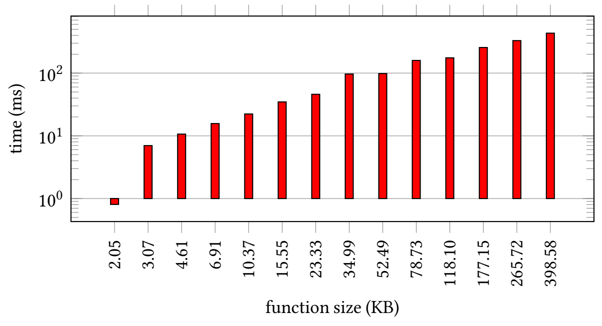


Fig. 14: Increase in static analysis time with function size.

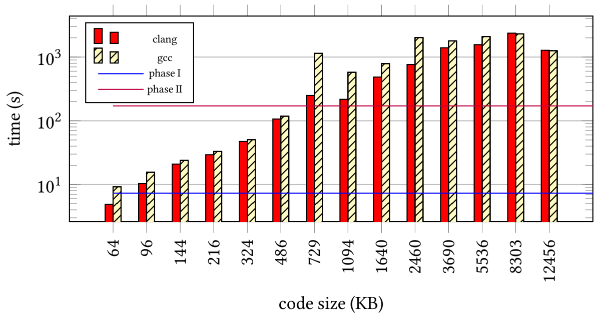


Fig. 15: Increase in disassembly time with binary size.

of embedded data. This is highlighted in 10% net error of Angr on StochFuzz benchmarks. Ghidra is known to use exception metadata, which results in significantly low FP rate. However, it still has a relatively high FN rate of around 2% and, thus, results in higher net error than our approach. Furthermore, we also found that the FN rate of Ghidra increases significantly to nearly 15% when exception metadata is not available (Ghidra-NE). DDisasm, on the other hand, has comparable net error for Pang et al.'s benchmarks if we artificially lower our statistical threshold S_{acc} to 2^4. However, when $S_{acc} \geq 2^{20}$, our approach has one-fourth of its net error.

6.4 Static analysis scalability

Because our static analysis operates at function level, its runtime performance directly depends on function size. Since there are millions of functions in our benchmarks, histogram seems to be an effective way to visualize the result. Fig 14 shows that our analysis performance is linear with function size.

Note that Fig 14 represents the cost of both the analyses InvReg and InvPtr described in Sec 4.2 combined. We choose to present this way because the cost of InvReg is already included in the cost of InvPtr. In more detail, our runtime performance depends on the number of abstract interpretations included in the analysis. Because the abstract domain used in InvReg is the foundation for memory address tracking, we need to run it as part of InvPtr. For this reason, we show the combined analysis time for both analysis. One may also interpret it as the cost of just InvPtr.

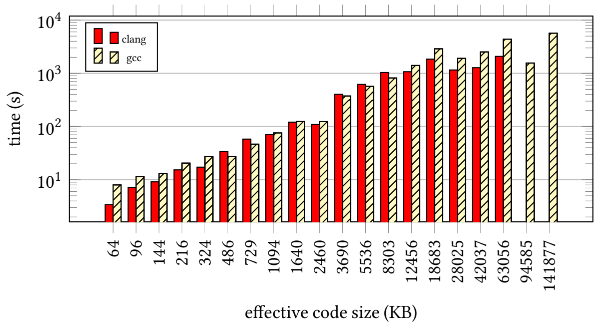


Fig. 16: Total disassembly time as a function of the total number of instructions considered in Phase III of our conflict resolution algorithm.

The chart shows that the analysis time increases linearly with function size. The largest functions are 100's of kilobytes in size, and they take hundreds of milliseconds to analyze. This shows that our static analysis, at its core, is quite efficient.

6.5 Total Disassembly Time

Our research goals are focused on accurate disassembly. We have not prioritized performance of disassembly because it is considered "compile-time" rather than a runtime cost. Moreover, disassembly is trivially parallelized, since we can disassemble different binaries on different cores.

Our dataset consists of thousands of binaries compiled by gcc and clang, so we again use histogram to visualize the relation between disassembly performance and code size. Fig 15 shows that a few binaries of size 0.5MB take longer to complete than those 2x and 3x in size. We therefore break down the disassembly process to investigate the sources of overhead.

In Sec 5, Phase I and II share the same goal of discovering a superset of all code. Unlike Phase III, they don't have to resolve conflicts. Therefore, we inspect performance in two categories: (a) Phase I and II, (b) Phase III. Note that the only nontrivial task in (a) is jump table analysis. Since static analysis already achieves linear time complexity, the only factor that could affect scalability is the number of functions to be analyzed. This is the case because we consider all possible function pointers in Phase II. In fact, more than a half of these entries are misaligned pointers from the actual entries by just a few bytes, and thus jump table analysis is repeated on similar control flow graph many times. To reduce performance of (a), we do a backward analysis from an indirect jump to identify a set of function entries that can reach the indirect jump. We then only perform jump table analysis on these function entries, and thus cut the amount of workload. We conduct experiment on all compiler/optimization on Pang et al's benchmark: it only takes about 8 seconds on average and 12 minutes on worst case to complete one binary.

Task (b) involves validation of non-returning calls, which complicates the process of jump table target validation. This is because non-returning calls affects the function boundary, which is critical in the valid function properties used in jump table target validation. This complexity also explains why some outlier binaries of size 0.7MB takes more than 15 minutes, as shown in Fig 15. Another source of overhead is in how jump table bounds are determined. Initially, we enforce that every jump table entry should not cause function properties being violated. A naive approach is to sequentially verify one entry at a time. Our approach is to verify a sequence of entries so binary search can be performed. Overall, task (b) takes 3 minutes on average and can be up to 2 hours for the largest binaries that are over 10MB in size. Since our static analysis is linear, the high overhead comes from resolving non-returning calls. This is shown in Fig 16, where effective code size represents the amount of code being analyzed. Note that this number increases by an order of magnitude over the sizes shown in Fig 15. As a result of this, Phase III ends up taking more than 10× the time taken by Phases I and II.

Amongst the contemporary disassemblers, DDisasm is our closest competitor in terms of accuracy. Ddisasm is also a static analysis based approach like ours and is about 4× faster than ours. However, it is unable to disassemble some large binaries such as clang compiled `wrf`, a SPEC 2006 binary. Primary reason behind our relatively higher performance overhead is the repititive static analysis by our conflict resolution algorithm (e.g., resolving non-returning calls and validating jump table targets), which ends up analyzing about 3× more instructions than the actual number instructions in a binary.

7 Related works

Binary disassembly can be categorized into two types: (i) dynamic and (ii) static disassembly. Dynamic binary tools such as PIN [43], Valgrind [36], DynamoRIO [8] and Strata [52] do not encounter error, but cannot achieve high code coverage. Static disassembly, in contrast, leans on completeness, but it is prone to error. Recent works [16, 60] assume that modern compilers such as gcc and clang do not inline data within code. However, this does not always hold, e.g., the precision of objdump drops to 85% for openssl [39].

Many reverse engineering tools such as ATOM [18], Diablo [15], Vulcan [17], and Pebil [28] rely on symbols and debugging information that are not available with stripped binaries. Recent works have shown [1, 40, 44] that exception-handling metadata can be used to accurately identify functions. Although this information is available in stripped binaries, many prominent C++ projects (e.g., chrome) disable its inclusion. CCFIR [61] utilizes relocation information present in Windows binaries to discover code pointers. It assumes that computed code pointers such as jump tables are also covered by relocation information. However, these assumptions are not universally applicable. Non-PIE binaries do not contain relocation information and x86_64 PIE binaries in Linux systems do not have relocation for jump tables.

Amongst the techniques that are metadata agnostic, BIRD [35] is one of the early works that explored use of function prologue patterns. Similarly, BAP [9] also relies on prologue matching. Ramblr [56] relies on static analysis to identify code pointers. As shown in evaluations of Ddisasm [19], Ramblr's static analysis has very high false positive and false negative rate. Another group of instrumentation [6, 58, 62] borrow the concept of runtime classification from dynamic binary instrumentation systems and combine it with static disassembly. Doing so, they side-step the disassembly errors. But they pay in form of performance penalty of up to 18% [64]. MULTIVERSE [6] relies on similar concept and speculatively disassembles from every byte, thereby producing output binary of huge size. Probabilistic disassembly [34] attempts to reduce the size overhead of MULTIVERSE with probabilistic properties, but it suffers from high FP rate, roughly 6%. Binary stirring [57] patches original code locations with jumps to reduce the performance cost. However, doing so makes it vulnerable to disassembly errors.

Lastly, some previous works [5, 53] employ machine learning technique to identify function entry, but their accuracy lacks behind state-of-the-art disassemblers. Not only that, the bias in training dataset may lead to a false sense of accuracy achieved by these works. In more specific, Nucleus [2] found that many functions were duplicated across training and test datasets, and BYTEWEIGHT accuracy drops to 60% when evaluated with another training dataset. A more recent work, XDA [41], leverages the contextual dependencies among byte sequences and achieve superior accuracy than BYTEWEIGHT and Shin et al. Despite that, its error rate is still roughly 10x compared to our approach on SPEC 2006 benchmarks.

8 Conclusion

In this paper, we present a *prioritized error correction* based disassembly technique that does not rely on any compiler generated metadata to achieve high accuracy. We have implemented and evaluated the effectiveness and performance of our system. Our system will be open-sourced before the publication of this paper. The *prioritized error correction* centers around a set of *invalid code behaviors* and *statistical data properties*. While the invalid behaviors are highly effective in identifying data, statistical properties help in resolving conflicting disassembly. Combining both the properties results in a superior accuracy (at least 10x better) in comparison with the other state-of-the-art disassemblers.

References

[1] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In *ACSAC*.

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *IEEE European Symposium on Security and Privacy*.

[3] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*.

[4] G. Balakrishnan and T. Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*.

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

[6] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.

[7] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. 2011. Bap: a binary analysis platform. In *Computer Aided Verification*.

[10] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference (DIMVA 2006)*.

[11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *ACM SIGSOFT*.

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Principles of programming languages*.

[13] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*.

[14] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM CCS*.

[15] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2005).

[16] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*.

[17] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. 2001. *Vulcan: Binary transformation in a distributed environment*. Technical Report. Technical Report MSR-TR-2001-50, Microsoft Research.

[18] Alan Eustace and Amitabh Srivastava. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *USENIX*.

[19] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *USENIX Security*.

[20] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[21] GNU. [n. d.]. Index of /gnu/binutils. https://ftp.gnu.org/gnu/binutils/. Accessed: 2023-03-03.

[22] Part Guide. 2011. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* (2011).

[23] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH* (2005).

[24] Xin Hu and Kang G Shin. 2013. DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In *Proceedings of the 29th annual computer security applications conference*.

[25] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *USENIX Security Symposium*.

[26] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Security and Privacy*.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code generation and optimization*.

[28] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS)*.

[29] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[31] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*.

[32] HJ Lu, Michael Matz, J Hubicka, A Jaeger, and M Mitchell. 2018. System V application binary interface. *AMD64 Architecture Processor Supplement* (2018).

[33] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization (CGO 2004)*.

[34] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[35] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. 2006. BIRD: Binary interpretation using runtime disassembly. In *Code Generation and Optimization*.

[36] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*.

[37] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. 2011. Retrofitting security in COTS software with binary rewriting. In *IFIP International Information Security Conference*.

[38] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[39] 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 *2021 IEEE Symposium on Security and Privacy (SP)*.

[40] 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 *DSN*.

[41] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2020. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770* (2020).

[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*.

[43] Pin [n. d.]. Pin - A Dynamic Binary Instrumentation Tool. http://pintool.org/.

[44] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. On the Impact of Exception Handling Compatibility on Binary Instrumentation. In *ACM FEAST*.

[45] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *ACSAC*.

[46] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating.. In *USENIX Security Symposium*.

[47] Rui Qiao and R Sekar. 2017. A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks (DSN)*.

[48] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards static binary debloating through abstract interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, (DIMVA 2019)*.

[49] Roman Rohleder. 2019. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*.

[50] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Code generation and optimization*.

[51] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*.

[52] Kevin Scott and Jack Davidson. 2001. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*.

[53] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*.

[55] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[56] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[57] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM CCS*.

[58] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *ACSAC*.

[59] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. 2014. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.

[60] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham

Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS*.

[61] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*.

[62] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.

[63] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[64] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. 2019. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

A Dataflow probability calculations

A.1 Valid def-use between registers

We say that there is a valid def-use between instruction I and J if (a) I writes to a register R, (b) J uses the same register R, and (c) there is no instruction between I and J that write R.

Given an instruction I, what is the probability that it is involved in a valid def-use relation? To answer this question, consider successive instructions that follow I until you find one at some I' that reads or writes R. If it reads R, then it is a valid def-use. If it only writes R, then it is an invalid def-use.

If the "instruction" at I' is random data, its probability of reading vs writing will follow the same overall probability that a random instruction reads or writes R. Note that, due to the fact that most memory operations also read a register to obtain the memory location, the probability of register reads is significantly more than that of register writes. This means that the probability that I' will read R is significantly higher than the probability that it will write R. Thus, there is more than a 50% probability that every "instruction" within random data that writes to a register will be part of a valid def-use pair.

Most "instructions" in random data will contribute to valid def-use relations between registers. Consequently, this criteria is unlikely to be useful for discriminating valid code from random data.