Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata†

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

Abstract

Accurate disassembly of stripped binaries is the first 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 novel approach for accurate disassembly that uses *statistical properties of data* to detect code, and *behavioral properties of code* to flag data. We present new static analysis and data-driven probabilistic techniques that are then combined using a prioritized error correction algorithm to achieve results that are 3× to 4× more accurate than the best previous results.

ACM Reference Format:
Soumyakant Priyadarshan, Huan Nguyen and R. Sekar. 2023. Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata[2]. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3623278.3624766

1 Introduction

Binary analysis and instrumentation provide the foundation for a wide range of applications such as binary debloating [26, 58**?**], optimization [43, 49], code similarity detection [12, 15], reverse engineering [38, 41, 59], vulnerability discovery [17, 40, 53, 66, 77], malware analysis [14, 33], and security hardening [6, 11, 18, 34, 57, 60, 67, 68, 70, 73, 74, 76]. *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 [4, 9, 10, 25, 36, 45, 52, 63, 66, 69, 71, 75, 77].

There are two basic techniques for disassembly: *linear sweep* and *recursive disassembly* [63]. Linear disassembly 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 copes better with such data, but suffers from low coverage because of the inability to decipher indirect control flow targets. State-of-the-art (e.g., [25, 61, 66]) combines the two, using recursive disassembly in the first phase and linear disassembly in the second phase to uncover code missed in the first phase. On binaries containing no embedded data, many of these techniques are highly (over 99%) accurate [50], but their accuracy falls rapidly on 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 binaries with embedded data, while approaching the ideal of 0% errors on data-free binaries.*

Existing tools such as Dyninst [30], Ghidra [61], and Angr [66] 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 [3, 51, 56]. 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 new disassembly approach for accurate disassembly of complex binaries without relying on 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, its *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

†Work supported by ONR grant N00014-17-1-2891 and NSF grants 1918667 and 2153056

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624766

more constrained. We focus on constraints that code must satisfy in order to run successfully and interoperate with other code. Clearly, data is not meant to be executed and hence may not satisfy these properties. Consequently, we can flag byte sequences as data when they violate these properties. Our overall contributions are:

- *Static analysis to flag data:* In Sec. 4, we propose *valid code properties* that all code must possess in order to (a) execute without crashing, and (b) interoperate with other code. For (a), we develop a static analysis to detect uninitialized pointer dereferences. To our knowledge, such an analysis has not previously been proposed or studied for disassembly. For (b), we develop a static analysis to detect violations of application-binary interface (ABI). Although versions of this property have previously been studied for function identification [59], we make several new contributions: (i) the development of an efficient, linear-time algorithm for inferring these properties, and (ii) formulating conservative subsets of ABI rules that are (almost) never violated by any code. As a result, our static analysis rejects valid code as data at the rate of 0.02% or less.
- *Statistical analysis to flag code:* We analyze statistical properties of data in Sec. 3 to derive the probability that a sequence of bytes corresponds to code. We make several advances over previous work [45] in this regard:
  - We show that the uniform data distribution assumption made in the previous work [45] can overestimate confidence levels by up to two orders of magnitude. We then present probability derivations based on distributions observed in real-world binaries, and validate these estimates through experiments.
  - Deriving probability estimates involves a "chicken-and-egg" problem: whether a location contains data depends a lot on whether other locations (e.g., the preceding bytes) contain code or data. We show how to break this cyclic dependency to derive robust probability estimates *without any prior knowledge about the prevalence of data in a binary.*
  - Finally, we show that previous work [45] significantly overestimates probabilities derived on the basis of valid dataflows.

  Through these improvements, our statistical techniques achieve 0.003% false negatives with 0.14% false positives on SPEC 2006 — an average error rate of about 0.07%, a significant improvement over the 3.4% error rate reported in [45] for the same suite.
- *Using valid code properties to address non-returning calls and missing jump table bounds.* Our valid code properties significantly mitigate the false positives due to (a) non-returning calls, which, if not recognized, cause any embedded data following them to be incorrectly flagged as code; (b) missing jump table bounds, which cause adjacent data to be interpreted as code pointers even though they may be pointing to data.
- *Conflict resolution using prioritized error correction:* In Sec. 5 we present an algorithm for combining the above techniques to further reduce disassembly errors. While such combinations can easily improve over the false positives OR the false negatives of either method, it is difficult to simultaneously improve both. We show how to use statistical scores to overcome this difficulty.
- *Experimental evaluation:* We perform a systematic evaluation of our disassembly approach in Sec. 6. We show that it achieves error rates that are 3× to 4× lower than that of contemporary disassemblers (e.g., Angr [66], DDisasm [25], Dyninst [30] and Ghidra [61]), while its runtime scales linearly with binary size.

Our system Dassa (Disassembly Aided by Statistical and Static Analysis) is available as open-source software from http://seclab.cs.stonybrook.edu/download.

2 Challenges in disassembling complex binaries

Linear disassembly embodied in tools such as `objdump` [27] provides high code coverage but suffers from a high false positive rate on binaries containing embedded data. *Recursive disassembly* [30, 61, 66] follows the control flow present in the code, and hence can skip data regions. But it incurs 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 [50], but this rate can be substantially higher on some programs.

Achieving high accuracy requires reducing both the false negatives and false positives. For this reason, many contemporary works [25, 66, 69] 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/case statements.)

The use of relocation information to discover address-taken functions has been explored in recent works [22, 72, 73]. 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 [75]. This discovers all address-taken functions, but introduces false positives due to 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 [22, 25, 30, 66, 69, 72, 75] use pattern matching to detect any arithmetic computation that resembles jump table target computation. As shown by the evaluations in [50], Dyninst's jump table identification achieves 99% accuracy. We follow a similar approach. However, estimating the bounds of jump tables can be hard. Recent techniques overestimate the size of jump tables whenever they are unable to accurately infer the size. Pang et al. [50] report 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 to reduce 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 calls are considered 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, disassembly of a fall-through path can result in an error.

Many recent works [30, 66] rely on a manually generated list of standard library functions that do not return, and avoid disassembly of fall-through code for such functions. Unfortunately, manually compiled lists can be incomplete, so a common strategy is to extend it using static analysis. In particular, the code of the callee can be analyzed to determine that it is *definitely non-returning* because it calls known non-returning functions or other definitely non-returning functions [44]. But this analysis won't recognize calls to *possibly non-returning* functions such as `error` in libc, whose returning behavior depends on argument values. Compilers can have this knowledge built into them, so they skip fall-through code in those instances where such a function won't return. Unfortunately, designing sound static analysis to detect such argument-dependent behavior is very challenging. Consequently, existing approaches for handling non-returning functions are not accurate enough. For instance, Pang et al. [50] attribute 40% of Dyninst's false positives to gaps in identifying non-returning calls.

Since a static analysis of the callee is unlikely to yield sufficient accuracy, we suggest a novel alternative: statically analyze the *caller* rather than the *callee* code. Specifically, we apply our valid code properties to recognize instances where the fall-through code is an independent function, and if so, mark the call as non-returning.

Unreachable code: Programs tend to have unused functions that are never called either directly or indirectly. Qiao et al. [59] 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 the 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 properties of data

Binary code is statistically different from data, and this fact has been used in previous works to improve disassembly accuracy [71]. 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 [8, 52, 65] 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 [8] fell from over 90% reported in the paper to just 60% on a second dataset [5]. This was because of an unrecognized bias in the training data that boosted accuracy in the original dataset.

Probabilistic disassembly [45] sidesteps the above challenges 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. They 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 shrinks data probabilities quickly, helping to separate code from data.

While probabilistic disassembly presents an elegant approach based on a principled foundation, its accuracy is not competitive with the state of the art. On the SPECint 2006 dataset, probabilistic disassembly [45] reports 6.8% false positives with 0% false negatives. On the same dataset, our statistical techniques achieve 0.14% false positives and 0.003% false negatives. Worse, in subsequent work by the same group [77] using a refinement of their original approach, 1.48% false positives and 11.74% false negatives were reported on a more complex dataset. Based on our experience with Dassa design and implementation, we believe that the following are significant factors contributing to their error rate:

- *Choice of patterns and associated probabilities:* The probability of random bytes exhibiting a dataflow relationship is over 50%, 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 accuracy 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 can lead to as much as a 100× overestimate. (Compare the second and third columns of rows 2 and 3 in Table 2.)

We present a new approach below to overcome these drawbacks. Similar to probabilistic disassembly, our approach avoids reliance on the statistical properties of code because of its variability across binaries. We also 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 that range in size from about 200 KB to 190 MB. (Note

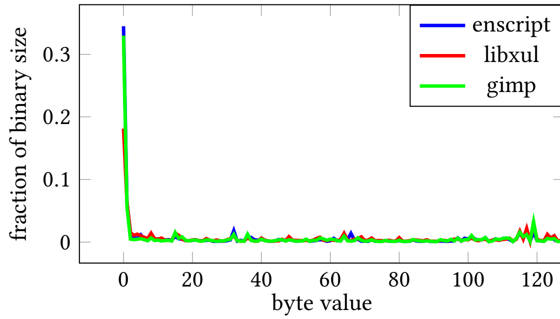


Fig. 1: Byte distribution across binaries.

that "data" in this regard means embedded data as well as byte positions in code that, as per the ground truth, don't correspond to an instruction beginning.) Despite these size differences, the probabilities are close across these binaries, indicating that we are relying on statistics that are stable. To further validate this claim, we generated 20 random subsets that were each about 0.1% of the total dataset size. The average probability of a zero-valued byte across these datasets was 27%, with a standard deviation of 7%. Nonzero byte values were essentially uniformly distributed.

Table 2 shows that the probability estimated using (a) the uniform distribution assumption, and (b) the distribution of Fig. 1, i.e., $Pr(0) = 0.3$ and $Pr(x) = 0.7/255$ for non-zero byte values. Note that the two estimates differ by as much as 100× on the last two rows of the table. The last column shows that the actual prevalence of these patterns in our dataset matches our calculation based on Fig. 1 distribution — indeed, it is within a factor of 2.

3.1 Control Flow Transfers (CFTs)

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

- D_T denote the event that the location T contains data, and
- $S \rightarrow T$ the event that the location S contains a control transfer "instruction" with the target T.

Our claim is that $Pr[D_T | S \rightarrow T]$, the conditional probability of D_T given $S \rightarrow T$, can be computed *without knowing if S is code or data.* We divide this derivation into three cases based on the CFT type.

We relied on Akshintala et al. [2] for information on x86_640 opcodes and their distribution in C/C++ programs on Ubuntu 16.04. Their queryable dataset at [1] was particularly handy.

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. Note that the target T of this transfer can be data *only if* the source itself is data. (Otherwise, we will have a case of a legitimate instruction at S that is jumping into data.) Since S is data, we can use the distribution from Fig. 1 to calculate the probability of these two opcodes. 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

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.

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 without relying on the 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 the 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.)

Although we used an average size of 4MB in our calculations in this section, in the actual implementation, probabilities are computed from the actual binary size. This means that for smaller binaries, the observation of an LJ provides a higher level of confidence (that the jump target is code) than larger binaries.

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:

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

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 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

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

SJs, probability calculations under uniform and Fig. 1 distributions match in this case.

3.2 Function Prologs

Using function prolog patterns to detect function entries has been explored by many disassemblers such as Dyninst [30] and Angr [66]. However, rigid pattern matching still results in low coverage. In contrast, we treat function prologs as a statistical property.

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

- *1-byte push:* 0x53 and 0x55 are the opcodes for *push %rbx* and *push %rbp* respectively. The probability of finding one of 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 $0.7^2 2^{-14} \approx 2^{-15}$.
- *stack decrement:* Local variables are created at function entry using *sub %rsp, constant* instructions. The 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 $\left(0.7/2^8\right)^3 * 2 \approx 2^{-24.5}$.

The probability 2^{-7} associated with a single byte push instruction is much too large, so we rely on prologs that contain (a) two 1-byte pushes or a single 2-byte push; or, (b) a single stack decrement. Table 3 compares the analytical probabilities calculated above with experimentally determined values. They are somewhat apart for (a) but match well in the case of (b).

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, for a code pattern x, we define a (code) *score* $S(x)$ as:

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

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 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 [25]. 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 [28].

Control flow transfers to invalid instruction boundaries have been used in previous works [25, 45, 75] to detect disassembly errors. The term *occlusion* has been used [45] 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 a certain location results in code that jumps to the middle of one of the instructions just disassembled.[1]

CFT instructions that target outside of the current binary have also been used to detect disassembly errors [75]. 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., the 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

[1] 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.

turns out to be challenging in itself. We have so far identified two such property classes, and both have proven to be very effective. The first set will cause a program crash, e.g., dereferencing an invalid memory location. The second set checks adherence to application-binary interface (ABI) specifications. Neither set is affected by programming errors in source code.

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 [47]. It has also been suggested as a criterion for deciding function starts [59]. However, disassembly shouldn't fail because of such programming errors. So, our analysis reports a violation only if:

- the violation is present on *every* program path,
- it is *not* the result of a programming error in source code, and
- when exercised, the bug will cause 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 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 specifications to determine what is initialized.

4.2.2 Invalid callee-saved register modification (InvReg)

The ABI [42] 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 them before their use and restore them before returning to the caller. Callee-saved register preservation has previously been used for function identification [59]. 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 [59] that analyzed each program path separately. (Even for loop-free programs, the number of paths can be exponential in code size.)

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 will never 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., VSA [7]) our analysis is based on the classic *abstract interpretation* [16] 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 Dassa 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. Specifically, we use LISC [32] because it supports the full x86 instruction set. Moreover, it produces a concise IR, which helps with the analysis speed. We then 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 is 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:

- *Byte-granularity tracking of abstract state,* so that we can accurately reason about instructions that move data across registers

of different sizes and/or memory.

- *Conservative pointer escape handling,* e.g., when a pointer to an on-stack array is passed to a callee that then initializes it.
- *Careful treatment of binary operators.* Arithmetic operations can produce defined results even if some arguments are $undef$, e.g., multiplication by 0, or subtracting a register from itself. Similarly, many logical operators can produce a defined result even if (some of) the arguments are $undef$.
- *Precise modeling of ABI restrictions,* e.g., the lowest 8-bits of `rax` register may be defined at function entry, but not the other bits.

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, one cannot simply stop after a few iterations since the result at this point is unsound. But there is an alternative, which is to start with the initial approximation of $\top$. In this case, the results are sound after every iteration, allowing the process to stop at any point. Starting at $\top$ does reduce precision, but this is not a concern because accuracy matters only when the analysis is applied to data, and data is unlikely to have structures such as loops. (Specifically, if we were analyzing code, the precision loss would lead to a missed non-code property, the result of which is to flag code as code!)
- *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 returning. 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) [7] is often used in binary analysis because its abstract domain has been purpose-designed for tracking memory addresses as well as integer values. However, since it can only capture abstract *values*, it cannot answer our 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* that can answer our question, but affine analysis is very expensive and does not scale beyond small programs [29, 46].

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. In previous work [62], we have developed such a 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. This 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 also speeded up in the same way.

5 Disassembly using Statistical and Static Analysis

We now present our disassembly algorithm that uses the statistical and behavioral techniques from the last two sections.

5.1 Phase I: Disassembling Definite Code

The initial phase discovers *definite code* using recursive disassembly from *definite roots:*

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

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

We use *conservative* recursive disassembly in this phase, avoiding the assumption that calls always return. In particular, we use a static analysis to identify functions that *may not* return, and avoid disassembling bytes that follow calls to such functions. (These bytes will be explored in subsequent disassembly phases.)

To identify non-returning calls, we begin with a small set of well-known non-returning functions 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 any possibly non-returning function is also classified as possibly non-returning. Although this analysis can be performed during disassembly, our current implementation uses an offline phase, with the resulting function lists used during disassembly.

5.2 Phase II: Disassembling Possible Code

The goal of this phase is to recover all potential code (*possible* code) in a binary. Any data within code regions will also be disassembled in this phase. As a result, disassembly may result in errors (e.g., invalid instructions) and/or conflicts, e.g., an instruction from one code region starts in the middle of another instruction from another code region. These conflicts and errors will be resolved in Phase III described in the next section.

Phase II also uses recursive disassembly but starts from a much

larger set of *possible roots*. This includes:

- *Possible code pointers:* Any constant appearing in the binary, with a value that falls within its code regions, is deemed a possible root. We examine all 64-bit constants (not necessarily aligned).
- *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. They are applied in order so that more likely candidates can be confirmed earlier on, reducing the need for the highly speculative steps listed at the bottom.

5.3 Phase III: Prioritized Error Correction

The possible code discovered above is a superset of all valid code. We now describe a conflict resolution algorithm (Fig. 4) that uses the statistical and behavioral properties discussed in Sec. 3 and 4 to select the correct code from this superset. The top-level function in our algorithm is `validateCode` (Fig. 4, 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. First, a statistical score is calculated for every instruction address in the code region. If a particular address matches multiple criteria, then it is assigned the sum of the scores for these criteria. For instance, if a location is targeted by k unconditional long jumps, it gets a score of $k \cdot 2^{10}$. The net score of a code region is the sum of the statistical scores assigned to individual addresses in the region.

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.

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 an independent function and added to the priority queue Q. A function may have many potentially non-returning calls. If so, resolution proceeds in the backward direction,

```
    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. 4: Prioritized error correction algorithm

starting with calls that are farthest from the function entry.

After resolving non-returning 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. If the function passes these checks, then we proceed to the last step, namely, validating jump table targets.

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. Occlusion check is applied first, 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 then deques the next function in possible code with the highest score. This repeats until Q is empty.

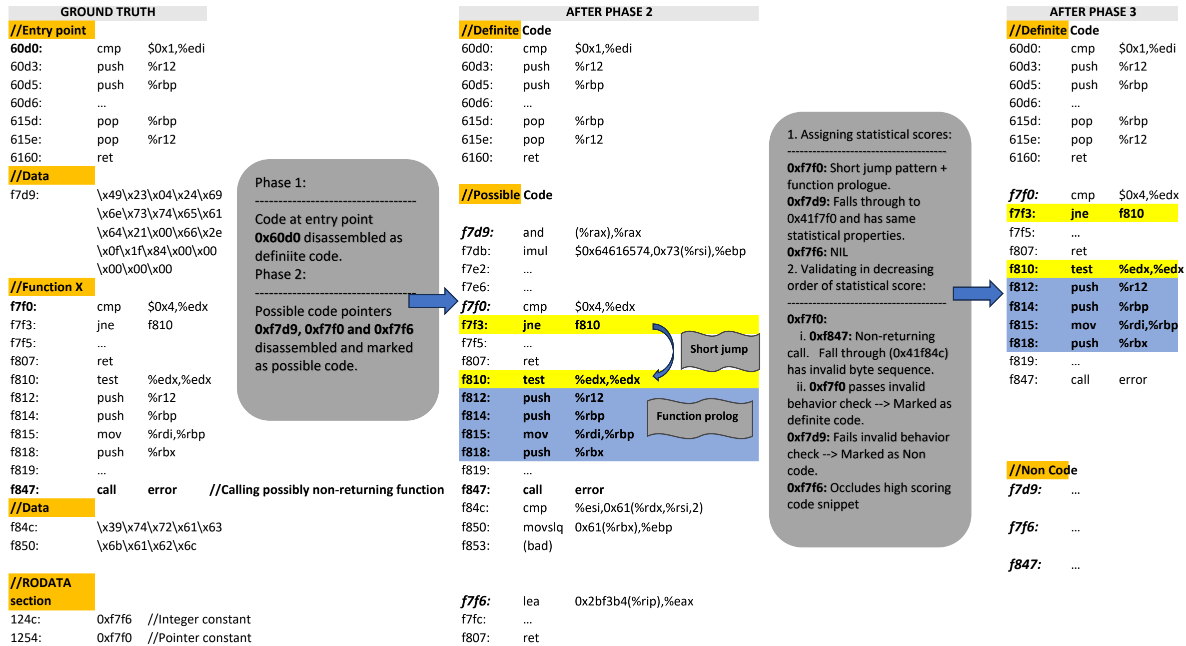


Fig. 5: Illustration of our disassembly algorithm and prioritized conflict resolution.

5.4 Illustration of Disassembly Algorithm

We now illustrate our disassembly algorithm using an example in Fig. 5. After completing Phase II of disassembly we have: (i) Definite code at program entry point `0x60d0` and (ii) Possible code snippets at `0xf7f0`, `0xf7d9` and `0xf7f6`.

Statistical scoring. Code snippets at `0xf7d9` and `0xf7f0` have a short jump target and function prologue consisting of one *2-byte push* (`push %r12`). Their net score will be $2^{15}+2^4 \approx 2^{15}$. In contrast, the code snippet at `0xf7f6` does not have any statistical property and will have a 0 score.

Prioritized evaluation. Since evaluation is done in decreasing order of statistical score, we evaluate `0xf7f0` first:

- *Checking occlusion:* It does not occlude any definite code.
- *Resolving non-returning call* at `0xf847`: The fall through of this call contains an invalid byte sequence at `0xf853` that cannot be disassembled into a valid instruction. Hence, it is marked as definitely non-returning.
- *Checking valid behavior*: It will pass *InvPtr* check. But the function does not preserve the callee saved registers before exiting (*InvReg*). We observed that this is the general tendency of functions exiting via a non-returning call. Hence, we do not mark such functions as invalid.

Applying the behavior checks to `0xf7d9` will fail since it uses an undefined register (`%rax`) in a memory dereferencing operation.

`0xf7f6` will be evaluated last. It will be marked as non-code since it occludes with definite code at `0xf7f5`.

6 Evaluation

Our evaluation aims to answer the following questions:

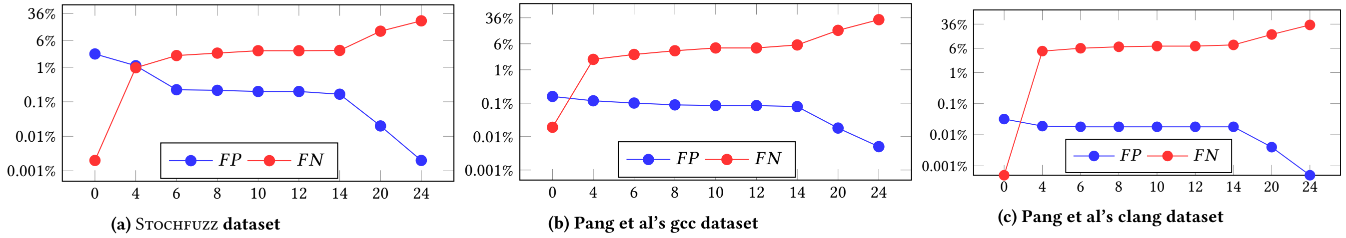
(1) How effective are *invalid behavior* checks in identifying *invalid code*? (Sec. 6.1.)
(2) How effective are *statistical properties* in prioritizing code over data? (Sec. 6.2.)
(3) How effective is the conflict resolution algorithm in combining our two techniques to achieve low error rates? (Sec. 6.3.)
(4) How do our results compare with state-of-the-art disassemblers in the presence of data between code? (Sec. 6.4.)
(5) How scalable is our static analysis? (Sec. 6.5.)
(6) How fast/scalable is the entire disassembly? (Sec. 6.6.)

Datasets. Table 6 summarizes our evaluation datasets:

- The dataset from Pang et al. [50] includes programs and libraries written in C/C++. It includes hand-written assembly code and a few instances of data within 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 [35] technique to modify a compiler to emit additional information such as basic block addresses, instruction boundaries and jump tables. We reused their ground truth for this dataset.
- The dataset from StochFuzz [77] 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 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



(a) STOCHFUZZ dataset (b) Pang et al's gcc dataset (c) Pang et al's clang dataset

Fig. 7: Effectiveness of statistical rejection threshold S_{rej} in reducing false positives. X-axis shows $log(S_{rej})$, while the Y-axis the FP and FN rates.

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 [50] and STOCHFUZZ's [77] Benchmark list

Phase II of our conflict resolution algorithm was checked using these properties, and the result is scored using the ground truth. Table 8 summarizes the results of this evaluation. The second column shows the base results with just *InvCF*, i.e., a snippet is rejected only if it contains invalid instructions or a control transfer to an invalid target location. This technique is able to achieve zero false negatives but comes with significant false positives.

The next column adds to *InvCF* the *InvReg* property, which indicates if a code snippet preserves callee-saved registers as per the ABI. The addition of this check removes 50% to 66% of FPs on Pang et al.'s dataset, but only 20% of the FPs are eliminated from the STOCHFUZZ dataset. Recall that the false positives in the latter dataset arise from the disassembly of embedded data. It turns out that most "instructions" resulting from this disassembly don't touch any callee-saved register, and hence *InvReg* is satisfied.

The fourth column evaluates *InvPtr* property that identifies the dereferencing of invalid code or data pointers. This property is 3× as effective as *InvReg* on the STOCHFUZZ dataset. For Pang et al.'s dataset, *InvPtr* is about as effective as *InvReg*.

Dataset	InvCF		InvCF+InvReg		InvCF+InvPtr		All	
	FP	FN	FP	FN	FP	FN	FP	FN
Stochfuzz	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 8: Percentage of false positives/negatives from invalid code properties.

Dataset	Compiler	InvCF		+Statistical score	
		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 9: Effectiveness of statistical scores in prioritizing code.

The last column shows that combining all the invalid code properties yields a 3- to 6-fold reduction in FPs as compared to just *InvCF*.

Result summary:

- The average false positive reduction achieved across these datasets by *InvReg* and *InvPtr* is 70%.
- Due to the conservative nature of *invalid code properties,* FP reduction is achieved at a very low false-negative rate (≤ 0.03%).

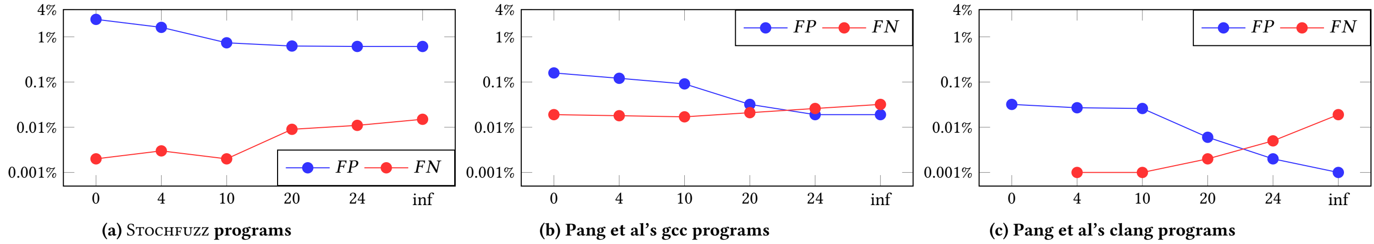
6.2 Effectiveness of statistical properties

We now evaluate the effectiveness of statistical techniques in isolation. To accomplish this, we modify our prioritized error correction algorithm to use statistical scores rather than static analysis results for conflict resolution. Table 9 shows our results. The third column shows the false positives and false negatives when the only mechanism for discarding a disassembled snippet is if it violates *InvCF*. The last column shows the improvement over this when a statistical score is applied in addition to *InvCF*. Specifically, we use our statistical techniques to assign a score to each snippet enumerated during Phase II. Then, among snippets that occlude each other, we pick the one with the highest score and discard the rest.

As compared to the base case where only *InvCF* is in play, conflict resolution using statistical scores leads to more than a 10-fold reduction in FPs on the dataset from Pang et al. On the STOCHFUZZ dataset, the reduction is more modest at 40%. A slight increase of 0.02% in false negatives was observed for one of the datasets.

Comparing the last columns of Tables 8 and 9, we see that the false negative rates are very similar. On the false positive front, however, statistical properties seem weaker. To improve the FP rate of statistical techniques, we investigated another modification to the error correction approach described above. Specifically, *possible code* snippets are discarded if their statistical score is below a *rejection threshold* S_{rej}. (Clearly, there is no need to apply such a threshold to *definite code.*) Fig. 7 plots the change in FP and FN, as S_{rej} is varied. It is easy to see that S_{rej} is effective in reducing FPs, but the FNs shoot up, increasing by 100× even at a relatively low $S_{rej} = 2^6$.

The above results demonstrate the limitations of using statistical techniques in isolation: they incur significant false positives or false negatives. In order to reduce both of them simultaneously, we need



(a) Stochfuzz programs (b) Pang et al's gcc programs (c) Pang et al's clang programs

Fig. 10: Effectiveness of prioritized error correction in reducing overall error rate. X-axis shows $\log(S_{acc})$, where S_{acc} is the acceptance threshold used in Fig. 4.

Dataset	DDisasm	Angr	Dyninst	Ghidra	Ghidra NE	Dassa				
						4	10	20	24	∞
Stochfuzz	1.22	5.15	14.85	*1.04*	7.39	0.8	0.35	0.32	**0.31**	0.32
Pang et al/gcc	*0.08*	0.27	13.4	0.29	5.01	0.07	0.06	0.03	**0.023**	0.03
Pang et al/clang	*0.01*	0.36	12.6	0.76	3.46	0.014	0.014	0.004	**0.003**	0.01

Table 11: Comparison with contemporary disassemblers. The lowest error rates of Dassa and competitors are highlighted in bold face and italics, respectively.

our conflict resolution algorithm (Fig. 4) that combines statistical and invalid code properties.

6.3 Combining statistical with valid behavior properties

Ideally, we can combine statistical and invalid code properties to obtain better accuracy than what is possible with either one of them. Unfortunately, typical combinations can reduce either the FPs or the FNs, but not both. For instance, if we accept a snippet as code only if both techniques accept it, then we will have a method with lower FPs than either technique but the FNs will be strictly greater than either one. On the other hand, if we accept a code snippet if it is accepted by either technique, then our FNs will be lowered but the FPs will be strictly higher than either technique.

One way around this challenge is to make better use of the degree of confidence rather than just a yes/no answer. Such a confidence measure is unavailable for invalid code properties, but statistical scores are indeed a measure of confidence. Hence our prioritized error correction algorithm uses a high statistical score (i.e., $\geq S_{acc}$) as a basis to skip all the code properties except stack pointer preservation. This enables us to avoid the FNs introduced by the other code properties whenever the statistical score is above S_{acc}.

When the score isn't high enough, our prioritized error correction algorithm applies all of the code properties. In these cases, the FNs of statistical and invalid code properties can add up. However, in our experiments, most disassembled snippets receive a statistical score above S_{acc}, and hence this worsening of FNs has been uncommon in our experiments. As a result, we obtain the results shown in Fig. 10. This chart plots FPs and FNs as a function of S_{acc}. It shows that $S_{acc} = 2^{24}$ provides the lowest error rate across these benchmarks. We conclude this section with two comments:

- We did not use a rejection threshold in the final algorithm because it leads to unacceptable FNs even at the low end of S_{rej}.
- Low statistical scores can only reduce FPs, but high scores can be used to improve FNs as well as FPs. The reduction of FNs is already explained above. For FP reduction, note that our conflict resolution will eliminate lower-scoring code snippets that occlude higher-scoring snippets. (Without such elimination, lower-scoring snippets will end up adding to the FPs.)

6.4 Comparison with State-of-Art Disassemblers

For simplicity and convenience, we combine the FP and FN metrics used so far into an *average error*[2], defined as $(FP + FN)/2$. Table 11 shows that the average error is minimized at $S_{acc} = 2^{24}$.

Our average error is 3× *to* 4× *smaller than the best among state-of-art disassemblers.* Overall, DDisasm comes closest to our method in terms of accuracy, but its error rates are nearly 4× that of our method. The gains of our method are particularly pronounced on the Stochfuzz dataset that contains a significant amount of data embedded within code. It is also noteworthy that our error rate is 3 to 10+ times lower than that of Ghidra despite the fact that it uses compiler metadata — specifically, exception handling metadata. Without this metadata, Ghidra's error rates increase by 5× to 17×.

Among other systems, Dyninst's high error rate stems from its reliance on function prolog matching, which leads to low code coverage and hence high FNs. Using linear scan to disassemble gaps, Angr achieves better coverage than Dyninst, but this increases its FP rate on binaries with embedded data, as shown by its 5.2% error rate on the Stochfuzz dataset.

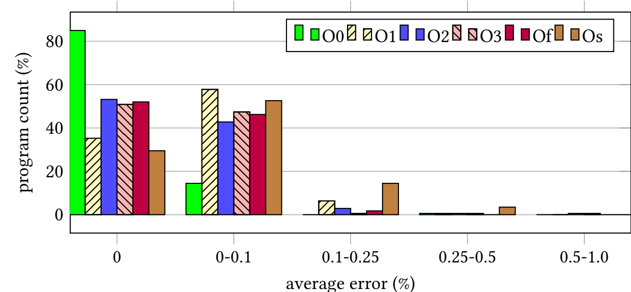


Fig. 12: Average error on gcc-compiled binaries from Pang et al.

Error distribution across programs. Fig. 12 and 13 further break down the error rates of our method on the dataset from Pang et al. Since our datasets consist of hundreds of programs compiled at 6

[2]Since the ultimate goal of disassembly is to achieve 100% accuracy, the error rate provides a better basis to compare various disassembly techniques and highlight the accuracy differences between them, in comparison with metrics such as F1-score. For completeness, the false positive and false negative rates are shown separately in Table 17 in the appendix.

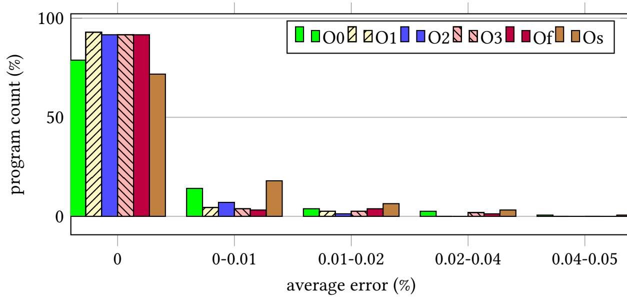


Fig. 13: Average error on clang-compiled binaries from Pang et al.

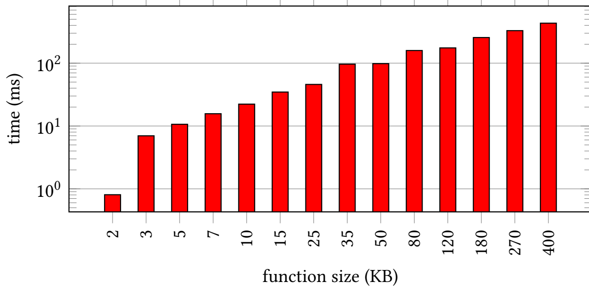


Fig. 14: Increase in static analysis time with function size.

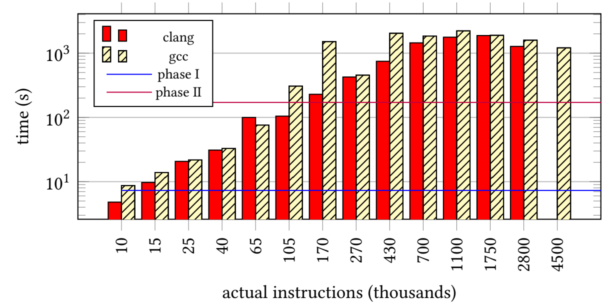


Fig. 15: Increase in disassembly time with actual instructions.

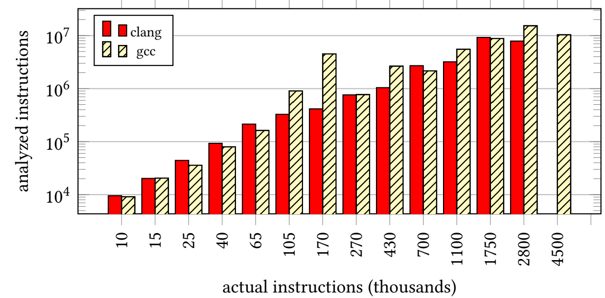


Fig. 16: Number of analyzed instructions with actual instructions.

different optimization results, it is not feasible to show the individual error rates. So we divided the error rate into 5 bins and show the percentage of programs whose error rate falls within each bin.

Note that the vast majority of clang-compiled binaries result in zero disassembly errors. The highest error rates experienced are about 0.05%, but this applies to a minuscule fraction of clang binaries. Error rates are somewhat higher on gcc-compiled binaries, but even here, about 90% experience error rates below 0.1%. Most of the outliers at 0.5% to 1% average error rate are programs that are so small that missing just 10 instructions can lead to an error rate of 0.5%.

6.5 Static analysis scalability

Since the scope of our static analysis is limited to a single function, its runtime complexity is determined by the sizes of functions. Fig. 14 demonstrates that this complexity is approximately linear. Specifically, we have divided functions into groups based on their size, and report the average runtime for each group. The high ends of successive bins increase by a ratio of 1.5. (This means that both axes use a logarithmic scale.)

Note that Fig. 14 shows the combined runtime of both InvReg and InvPtr analyses. This is because InvPtr analysis relies on the contents of registers and memory, both of which are computed as part of InvReg analysis. Thus, by the time the InvPtr analysis is completed, InvReg would also have been completed.

Fig. 14 shows that the runtime increases linearly with the group size. The largest functions are hundreds of KBs in size and they take hundreds of milliseconds to analyze. This shows that our static analysis, at its core, is reasonably efficient.

6.6 Total Disassembly Time

Our goal is on disassembly accuracy and scalability to larger binaries. Thus, our performance focus has been on the overall complexity of disassembly rather than absolute runtimes. To demonstrate that our disassembler exhibits linear-time complexity, we first organize binaries into groups based on the number of actual instructions and report the average disassembly time for each group. Fig. 15 shows the results and exhibits an overall linear trend.

Note that in some cases, smaller binaries take a longer time to disassemble than those that are 2× to 4× in size. To better understand the sources of such performance inversion, we zoomed in on the times spent in the three phases of our disassembly algorithm. Specifically, the only potential source of non-linear complexity[3] is that the code snippets enumerated in Phase III can be overlapping. Recall that Phase III treats each snippet as a possible function, constructing a CFG for it, performing static analysis, etc. As a result, a single instruction may be analyzed many times, belonging to a different candidate function each time. Fig. 16 plots the average number of analyzed instructions with respect to the number of actual instructions. Note that we can quantify the amount of overlapping code, called *reanalysis ratio*, by dividing the y-value by the x-value. In fact, this ratio is largely a constant across different binary sizes, except for one group of binaries containing 105K to 170K instructions. This shows that the number of analyzed instructions grows roughly linearly with the number of instructions in the binary. Moreover, the similarity between Fig. 15 and Fig. 16 shows a strong relation between the runtime performance and the amount of overlapping code.

Among contemporary disassemblers, DDisasm is our closest competitor in terms of accuracy. While it is about 4× less accurate, on average, it is also faster than our technique by about 4×. However, its performance does not seem to scale linearly with binary size. Because of this, it can run out of memory or take far too long to complete on some of the larger binaries. For instance, it could not disassemble clang-compiled `wrf`, a SPEC 2006 binary.

[3]By design, all other aspects of our design, including the implementations of disassembly, lifting and static analysis, take linear time.

7 Related Work

Binary disassembly can be categorized into two types: (i) dynamic and (ii) static disassembly. Dynamic binary tools such as PIN [54], Valgrind [47], DynamoRIO [13] and Strata [64] rely on just-in-time disassembly of basic blocks, i.e., basic blocks are disassembled when they are about to be executed. This side steps all of the complexities of static disassembly, but the downside is the overhead and complexity of doing everything at runtime.

Early work on binary analysis and instrumentation targeted RISC architectures [24, 37]. Disassembly is often straightforward on these architectures because instructions are of the same length, and data regions are separated from code [19]. Schwarz et al. [63] describe the challenges faced by linear and recursive disassembly algorithms on CISC architectures, specifically x86. In order to reliably disassemble such binaries, many researchers and tools relied on additional compiler metadata [23], e.g., relocation information.

Disassembly based on metadata. Although relocation information was generally unavailable in stripped binaries, this changed with the wide deployment of ASLR on Microsoft Windows. Loading a Windows library at random locations requires it to be first "rebased" at this address [39], and this process requires relocation information. Zhang et al. [73] were the first to take advantage of this fact for robust disassembly of complex Windows binaries. In contrast, 32-bit UNIX-based systems (e.g., BinCFI [75]) continued to rely on a combination of linear and recursive disassembly since position-independent code on these systems didn't require rebasing. On 64-bit Linux, position-independent code does require relocation information. Exploiting this, some recent binary instrumentation tools have been based almost exclusively on linear disassembly [22, 57, 72]. Unfortunately, when there is embedded data, relocation information is insufficient for accurate disassembly, causing these systems to fail in unpredictable ways on such binaries.

Recent works have shown that exception-handling metadata, preserved even in stripped binaries, can be used to accurately identify functions [3, 51, 56] and improve disassembly [51]. A key drawback of this approach is that it may not be present in non-C++ binaries. Moreover, some prominent C++ projects (e.g., Chrome) disable its inclusion due to its large size. Finally, even in binaries that include this information, it may not have 100% coverage.

Due to the above drawbacks and incompleteness of metadata-based approaches, we have focused on a metadata-free approach.

Exhaustive disassembly. Instead of relying on compiler-generated metadata, Kruegel et al. [36] propose exhaustive disassembly that treats *every* byte as a possibly valid instruction boundary. Despite being exhaustive, it is important to note that it is efficient, i.e., takes time linear in the binary size. Heuristics were proposed to prune out some of the unlikely instructions, and these were further improved in subsequent works [10, 71].

Multiverse [9] combines exhaustive disassembly with runtime address translation in order to support robust static binary instrumentation of complex binaries. Its main drawback is an average 9× increase in code size. To reduce this overhead, probabilistic disassembly [45] develops an approach for recognizing and eliminating the vast majority of non-code from the disassembler's output. This is achieved by estimating the probabilities of occurrence of certain code patterns in data, propagating these probabilities along control-flow paths, and then removing "instructions" with very low probabilities. Stochfuzz [77] significantly improves on the performance of their propagation algorithm, and proposes a way to reduce false negatives using fuzzing. The statistical component of our approach improves and corrects the probabilities used in these papers in important ways so as to achieve better overall accuracy.

Static analysis. Static analysis of control flows is widely used in disassembly and binary analysis, including control-flow graph construction and the identification of non-returning functions [44]. More complex analyses that reason about data flows have been used in jump table analysis. Our identification of possible jump table targets is based on techniques similar to those used in previous works [20, 21, 25, 30, 44]. At the same time, we address an important drawback of existing approaches when it comes to spurious jump table targets, which often result from the difficulty of analyzing jump table bounds. Specifically, our approach for validating jump table targets using code behaviors is able to significantly reduce false positives among jump table targets.

Rui et al. [59] proposed the use of static analysis to reason about the preservation of stack pointer and other callee-saved registers in order to detect function boundaries. We advance this work in several important ways. First, we propose a new dataflow property on crash-inducing memory accesses and use it to detect invalid code. Second, we present linear-time algorithms for all of the properties, while further refining the properties to reduce false positives and negatives. Third, we show how our properties can improve the identification of non-returning functions.

Disassembly using learning techniques. ByteWeight [8] and Shin et al. [65] use learning-based techniques to identify function entry. More recently, XDA [52] achieves a higher accuracy with contextual dependencies among byte sequences. Despite that, its error rate is still 10× compared to Dassa on SPEC 2006 datasets.

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 Dassa. 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 with roughly 3× to 4× lower in error rate in comparison with the other state-of-the-art disassemblers. The high accuracy offered by Dassa enabled us to build Safer [55], a system for robust and efficient static instrumentation of complex binaries that may contain embedded data and/or other features that can lead to undefined behaviors in previous low-overhead approaches.

A Artifact Appendix

A.1 Abstract

This artifact submission is for our binary disassembly tool named Dassa. It is a static binary disassembler that achieves 3× to 4× more

accurate disassembly than the state-of-the-art tools. DASSArelies on two sets of properties: (i) *statistical data properties* and valid code behaviors to achieve low error rates while disassembling stripped binaries. The two properties are combined via a prioritized error correction algorithm. We evaluate this algorithm using the datasets used by recent works of STOCHFUZZ [77] and Pang et al. [50].

The artifact consists of a VM image in which our disassembler has been pre-installed. The source code and the test suite for our disassembler are also present in the VM. The test suite consists of the above mentioned datasets, ground truth and scripts that run disassembly and compare with the ground truth. The final output consists of assembly code for each of the dataset binaries and the overall accuracy results.

A.2 Artifact check-list (meta-information)

- **Algorithm:** None
- **Program:** None
- **Compilation:** None
- **Transformations:** Binary disassembly
- **Binary:** Disassembly program binary and dataset binaries are included in artifact.
- **Model:** None
- **Data set:** Datasets are included with artifacts. They are borrowed from recent works of STOCHFUZZ [77] and Pang et al. [50].
- **Run-time environment:** Ubuntu 20.04.
- **Hardware:** An x86-64 machine with 16GB of RAM and 256GB of storage space.
- **Run-time state:** None
- **Execution:** None
- **Metrics:** False positive and false negative rate.
- **Output:** Assembly code, false positive and false negative rate
- **Experiments:** Shell scripts provided to disassemble datasets and produce accuracy results.
- **How much disk space required (approximately)?:** 100 GB
- **How much time is needed to prepare workflow (approximately)?:** None
- **How much time is needed to complete experiments (approximately)?:** 30 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Gpl V3
- **Data licenses (if publicly available)?:** NA
- **Workflow framework used?:** None
- **Archived (provide DOI)?:** 10.5281/zenodo.10121698

A.3 Description

DASSA's current prototype requires Ubuntu 20.04 operating system. It further requires additional packages (Capstone and Ocaml) for disassembly and static analysis. We are submitting our artifact as an Oracle VirtualBox VM. DASSA is already installed in the VM along with all the prerequisite packages.

A.3.1 How to access

DOI: 10.5281/zenodo.10121698
VM username: dassa
Password: ubuntu123

A.3.2 Hardware dependencies

An x86_64 machine with 16GB RAM and 256GB storage.

A.3.3 Software dependencies

The VM image is in .ova format and can be opened with Oracle virtual box.

A.3.4 Data sets

Datasets are included with artifacts. They are borrowed from recent works of STOCHFUZZ [77] and Pang et al. [50]. Note that we did not recompile the datasets. Rather we reused the datasets as released by the respective authors.

STOCHFUZZ DATASETS This dataset consists of 24 binaries and was compiled with clang at O2 optimization. They were compiled by forcing the compiler to inline data within code.

Pang et al. datasets This includes programs and libraries written in C/C++. 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). The total number of binaries in this data set is around 2100.

A.4 Installation

Installation is not required. The VM comes pre-installed with the the disassembler and other additional dependencies.

A.4.1 Basic step

To run the disassembler on a single program:

```
> cd /home/dassa/DASSA/run
> make clean
> make
> ./run.sh /bin/ls disasmonly
```

The output assembly: /home/dassa/DASSA/run/tmp/ls_defcode.s.

To disassemble and dump the control flow graph (CFG):

```
> ./run.sh /bin/ls disasmonly dumpcfg
```

The CFG is dumped in text files as below:

- *Function entry:* It contains two types of entry points: (i) Definite entries – classified as true function entries by DASSA and (ii) Possible entries – classified as invalid by DASSA.

  ${HOME}/DASSA/run/tmp/cfg/functions.lst

- *Definite Basic blocks:* These are classified as true code by DASSA.

  ${HOME}/DASSA/run/tmp/cfg/definite_basicblocks.lst.

- *Possible Basic blocks:* These are classified as invalid.

  ${HOME}/DASSA/run/tmp/cfg/psbl_basicblocks.lst.

- *Jump tables:*

  ${HOME}/DASSA/run/tmp/cfg/jmptables.lst

A.5 Experiment workflow

Experiments described in the subsequent sections are aimed at producing the final results of our disassembly algorithm. Specifically, we will be describing steps to produce DASSA's accuracy results mentioned in Table 11 of the paper. This involves running two scripts:

(1) *disasmAllFiles.sh*: This script disassembles all binaries of a given dataset. It requires four arguments: (i) compiler, (ii) optimization level, (iii) disassembly method and (iv) dataset name. Additionally, it also takes a 5th optional argument that indicates whether we want to disassemble *all* or a *custom* set of binaries.

(2) *compareAllBenchmarks.sh*: This script compares the disassembly output with the ground truth and generates the accuracy report.

The usage of both these scripts is described in the subsequent sections.

A.6 Evaluation and expected results

All the steps below are for $S_{acc} = 2^{24}$, which is the default configuration of DASSA. This experiment takes approximately 2 hours to complete. Therefore, it must be run in background using nohup.

A.6.1 Evaluating Pang et al. datasets

These datasets consists of binaries compiled with gcc and clang at 6 optimization levels. Disassembly for each optimization level requires 1.5-2 hours to complete. Total time required for all optimization levels is around 10-12 hours. To run disassembly for all optimizations of gcc:

```
> cd /home/dassa/DASSA/disasm-testsuite
> nohup ./disasmAllOpt.sh gcc DASSA_Sacc_24 \
      PangEtAl all > PangEtAl.log &
```

To track the progress, please use `tail -f PangEtAl.log`.

clang datasets Same steps needs to be followed to test clang compiled binaries.

Disassembly output The output assembly code for each binary will be present in the below directory.

```
> cd /home/dassa/DASSA/disasm-testsuite/
> cd DASSA_Sacc_24/PangEtAl/gcc/O0
> ls -lrt *defcode.s
```

Expected results The average error $((FP + FN)/2)$ should match the number reported in Table 11 of paper (e.g., 0.003 for Pang et al.'s clang datasets). The results may vary by a small fraction. For example, results Pang et al.'s clang datasets can vary from 0.003 to 0.005.

Varying S_{acc} Note that Table 11 in paper reports DASSA's accuracy at various S_{acc} levels. To run the disassembly for different levels, we need to replace the number in DASSA_Sacc_number in all the above mentioned steps. For example, to run for $S_{acc} = 2^4$.

```
> cd /home/dassa/DASSA/disasm-testsuite
> nohup ./disasmAllOpt.sh gcc DASSA_Sacc_4 \
      PangEtAl all > PangEtAl.log &
```

A.6.2 Evaluating STOCHFUZZ *datasets*

Note that STOCHFUZZ dataset consists of binaries compiled by clang at one optimization level (O2) only.

```
> cd /home/dassa/DASSA/disasm-testsuite
> nohup ./disasmAllFiles.sh clang O2 DASSA_Sacc_24 \
      stochfuzz all > stochfuzz.log &
```

To track the progress of the run:

```
> cd /home/dassa/DASSA/disasm-testsuite
> tail -f stochfuzz.log
```

After the disassembly run is complete, we need to produce accuracy results as below:

```
> cd /home/dassa/DASSA/disasm-testsuite
> ./compareAllBenchmarks.sh DASSA_Sacc_24 stochfuzz clang
```

A.6.3 Applying instrumentation

This section is about extending DASSA's disassembly to instrument stripped binaries. Specifically, we apply our instrumentation approach discussed in our parallel paper SAFER [55] and show error free instrumentation.

(1) Copy the binary to the test folder.
```
> cp /usr/bin/ls ~/DASSA/test
```
(2) Find the dependencies of ls.
```
> cd ~/DASSA/testsuite
> ./find_libs.sh ${HOME}/DASSA/test/ls
> truncate -s 0 randomized.dat
```
(3) Run instrimentation on ls.
```
> cd ${HOME}
> nohup ./instrument_prog.sh ls &
```
Wait for the instrumentation to finish.

(4) Run the instrumented ls. Contents of the directory should be printed.
```
> cd ${HOME}/instrumented_libs/
> ./ls
```
(5) Applying CFI and shadow stack:
```
> cd ${HOME}
> nohup ./instrument_prog.sh ls ptr_trans=CFI_SHSTK &
```
Wait for completion and run the instrumented ls as shown in the previous point.

References

[1] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. 2019. Occurrence of instructions among C/C++ binaries in Ubuntu 16.04. http://x86instructionpop.com/.

[2] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. 2019. X86-64 instruction usage among C/C++ applications. In *SYSTOR*.

[3] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In *ACSAC*. https://doi.org/10.1145/3359789.3359825

[4] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security*.

[5] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *IEEE S&P*. https://doi.org/10.1109/EuroSP.2017.11
[6] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security*.
[7] G. Balakrishnan and T. Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*. https://doi.org/10.1007/978-3-540-24723-4_2
[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security*.
[9] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.
[10] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative disassembly of binary code. In *CASES*.
[11] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*.
[12] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.
[13] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *CGO*.
[14] 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)*.
[15] 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*.
[16] 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*. https://doi.org/10.1145/512950.512973
[17] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static detection of vulnerabilities in x86 executables. In *ACSAC*.
[18] 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*.
[19] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time binary rewriting techniques for program compaction. *ACM TOPLAS* (2005).
[20] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. https://doi.org/10.1145/2968455.2968514
[21] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. https://doi.org/10.1145/3033019.3033028
[22] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE S&P*.
[23] 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.
[24] Alan Eustace and Amitabh Srivastava. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *USENIX*.
[25] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *USENIX Security*.
[26] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on CCS*.
[27] GNU. [n. d.]. Index of /gnu/binutils. https://ftp.gnu.org/gnu/binutils/. Accessed: 2023-03-03.
[28] Part Guide. 2011. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* (2011).
[29] Sumit Gulwani and George C Necula. 2003. Discovering affine equalities using random interpretation. In *POPL*.
[30] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH* (2005). https://doi.org/10.1145/1127577.1127590
[31] Niranjan Hasabnis and R Sekar. 2016. Extracting Instruction Semantics Via Symbolic Execution of Code Generators. In *ACM FSE*.
[32] Niranjan Hasabnis and R Sekar. 2016. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*. https://doi.org/10.1145/2872362.2872380
[33] Xin Hu and Kang G Shin. 2013. DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In *ACSAC*.
[34] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *USENIX Security*.
[35] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Security and Privacy*. https://doi.org/10.1109/SP.2018.00029
[36] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX Security*.
[37] James R. Larus and Eric Schnarr. 1995. EEL: machine-independent executable editing. In *PLDI*.
[38] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).
[39] Lixin Li, Jim Just, and R. Sekar. 2006. Address-space randomization for windows systems. In *ACSAC*.
[40] 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).
[41] 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*.
[42] HJ Lu, Michael Matz, J Hubicka, A Jaeger, and M Mitchell. 2018. System V application binary interface. *AMD64 Architecture Processor Supplement* (2018).
[43] 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 *CGO*.
[44] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *ISSTA*. https://doi.org/10.1145/2931037.2931047
[45] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *IEEE/ACM ICSE*.
[46] Markus Müller-Olm and Helmut Seidl. 2004. A note on Karr's algorithm. In *International Colloquium on Automata, Languages, and Programming*. https://doi.org/10.1007/978-3-540-27836-8_85
[47] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*.
[48] Huan Nguyen, Niranjan Hasabnis, and R Sekar. 2019. LISC v2: Learning Instruction Semantics from Code Generators. http://www.seclab.cs.sunysb.edu/seclab/liscV2/. Accessed: 2023-08-06.
[49] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *IEEE/ACM CGO*. https://doi.org/10.1109/CGO.2019.8661201
[50] 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 *IEEE S&P*. https://doi.org/10.1109/SP40001.2021.00012
[51] 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*.
[52] 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).
[53] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *IEEE S&P*.
[54] Pin [n. d.]. Pin - A Dynamic Binary Instrumentation Tool. http://pintool.org/.
[55] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. SAFER: Efficient and Error-Tolerant Binary Instrumentation. In *USENIX Security*.
[56] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. On the Impact of Exception Handling Compatibility on Binary Instrumentation. In *ACM FEAST*.
[57] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *ACSAC*. https://doi.org/10.1145/3427228.3427292
[58] 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*.
[59] Rui Qiao and R Sekar. 2017. A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2017.29
[60] Rui Qiao, Mingwei Zhang, and R Sekar. 2015. A Principled Approach for ROP Defense. In *ACSAC*.
[61] Roman Rohleder. 2019. Hands-on Ghidra - A Tutorial about the Software Reverse Engineering Framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*. https://doi.org/10.1145/3338503.3357725
[62] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*. https://doi.org/10.1145/1356058.1356069
[63] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*.
[64] Kevin Scott and Jack Davidson. 2001. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*.
[65] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *USENIX Security*.
[66] 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)*. https://doi.org/10.1109/SP.2016.17
[67] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. 2008. Practical

Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE S&P*.
[68] 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 *ACM CCS*. https://doi.org/10.1145/2810103.2813673
[69] 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*. https://doi.org/10.14722/ndss.2017.23225
[70] 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*.
[71] 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*.
[72] 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*. https://doi.org/10.1145/3373376.3378470
[73] 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 S&P*. https://doi.org/10.1109/SP.2013.44
[74] Mingwei Zhang, Michalis Polychronakis, and R Sekar. 2017. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. In *ACSAC*.
[75] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.
[76] Mingwei Zhang and R Sekar. 2015. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*.
[77] 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 *IEEE S&P*.

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 writes 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.

Note that the probability of valid def-uses involving memory locations is significantly smaller, but their accurate recognition is more difficult because indirect memory accesses are so common.

B Implementation

Our disassembler is implemented in C++ and consists of 40KLoC. It consists of three modules: (i) an ELF parser, (ii) Disassembler and (iii) Static analysis module.

ELF parser. ELF stands for Executable and Linkable Format and is the standard format for Linux binaries. We have developed our own ELF parsers instead of using Linux utilities such as `readelf`. Our parser performs the following tasks prior to disassembly:

- Given a binary, our ELF parser first parses the ELF header present at the beginning of the binary file to find out (i) the program entry point and (ii) location of the section header table.
- It then parses the section header table. The section header table is an array of C structures of type `Elf64_Shdr`. There is one entry for every section present in the binary. We are interested in finding (i) The dynamic symbol table (.dynsym), (ii) .init_array and .fini_array sections, and (iii) address of all code and data sections (.text, .plt, .init, .fini, .rodata, etc).
  The dynamic symbol table contains the address of all exported functions. .init_array and .fini_array sections contain pointers to default initialization and clean-up functions added by the compiler to every binary. These are used as definite code roots in our disassembly's Phase I described in Section 5.

Disassembler. Our recursive disassembler is implemented using `Capstone`. The roots or starting points of recursive disassembly are obtained as described in Section 5. Given a root, it reads the byte sequence from the file and asks `Capstone` to convert them into assembly instructions. It then organizes the sequence of assembly instructions into basic blocks and constructs a control flow graph (CFG). If the CFG has indirect control flow transfers, it refers to the static analysis engine to find any associated jump tables. If any jump table is found, it is decoded to obtain the targets and the targets are fed into the disassembler as roots for further disassembly. This process continues until no new jump table is discovered.

Static analysis module. Static analysis module is responsible for finding jump tables and checking invalid code behaviors. These analyses are performed on an intermediate representation (IR), specifically, gcc's RTL. We first use our architecture-neutral approach for [31, 32] for lifting assembly. Our system Lisc [48] lifts assembly to gcc's RTL. This lifter implementation consists of 3.5KLoC of Ocaml code.

Dataset	DDisasm		Angr		Dyninst		Ghidra		Ghidra NE		DASSA									
											4		10		20		24		∞	
	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
Stochfuzz	1.88	0.55	9.6	0.7	0.002	29.7	0.067	2	0.07	14.7	1.62	0.003	0.738	0.002	0.626	0.009	0.613	0.011	0.612	0.015
Pang et al/gcc	0.14	0.006	0.04	0.5	0.005	26.8	0.016	0.55	0.02	10	0.121	0.018	0.091	0.017	0.032	0.021	0.019	0.026	0.019	0.032
Pang et al/clang	0.02	0	0.01	0.7	0.01	25.2	0.02	1.5	0.01	6.9	0.027	0.001	0.026	0.001	0.006	0.002	0.002	0.005	0.001	0.019

Table 17: Comparison with contemporary disassemblers.