DSV: Disassembly Soundness Validation without Assuming a Ground Truth*

Xiaoxin An, Freek Verbeek, and Binoy Ravindran

Virginia Tech, Blacksburg, USA
{xxan15,freek,binoy}@vt.edu

Abstract. Disassembly is a crucial step in binary security, reverse engineering, and binary verification. Various studies in these fields use disassembly tools and hypothesize that the reconstructed disassembly is correct. However, disassembly is a challenging and undecidable problem. Even state-of-the-art industrial disassemblers suffer from issues ranging from incorrectly recovered instructions to incorrectly assessing which addresses belong to instructions and which to data. We thus present DSV: a systematic and automated approach to validate whether the output of a disassembler is sound with respect to the binary. No source code, debugging information, or annotations are required. We apply DSV to 102 binaries of Coreutils with eight different state-of-the-art disassemblers from academia and industry. DSV is able to find soundness issues in the output of all these disassemblers. Using DSV to validate the output of a disassembler increases trust in any research effort built on top of it.

Keywords: reverse engineering, disassembly soundness, concolic execution, bounded model checking

1 Introduction

Disassembly is a crucial part of many reverse engineering and related subdisciplines such as decompilation, binary analysis, binary verification, and binary rewriting. Practitioners have a plethora of tools available [1,2,3,4] to recover assembly instructions from an executable binary. Still, disassembly is not a solved problem: new techniques are developed based on, among others, machine learning [5], advanced heuristics, and inference [1,2,3]. These new techniques improve accuracy and soundness.

In most of the reverse engineering work, practitioners implicitly take the premise that the disassembly process is trustworthy. This premise is based on well-developed commercial and open-source disassemblers. For example, Ramblr [6] uses static binary rewriting to implement binary reassembling. The developers take angr [2] as the base platform to disassemble the binary and to

* This is the authors' version of the work posted here per the publisher's guidelines for your personal use. Not for redistribution. The final authenticated version was published in the Proceedings of the 14th International Symposium on NASA Formal Methods, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, LNCS, volume 13260, and is available online at: https://doi.org/10.1007/978-3-031-06773-0_34

rebuild the control flow graph (CFG), which means the correctness of Ramblr highly relies on the correctness of angr. As another example, Ghidra [3] is a state-of-the-art tool for decompilation. Its capabilities include control-flow reconstruction, type-inference, and pointer-analysis. However, all the functionalities are based on the assumption that disassembly is done correctly.

Disassembly, however, is by its very nature inherently an *untrustworthy* process. It is an undecidable problem [7,8]. In a context where only the binary is available (e.g., legacy systems or third-party proprietary software), there is *no ground truth* as to what the "correct" assembly instructions are. Even state-of-the-art disassemblers suffer from issues when, e.g., instructions are overlapping, data and instructions are mixed, indirect jump/call targets are unresolved, or a security vulnerability leads to unexpected control flow. Although mainstream disassemblers, such as objdump, Hopper, and IDA Pro, are developed by numerous researchers and are elaborately tested, different kinds of issues of these disassemblers have been discovered and reported [9,10].

In this paper, we propose a formal definition for the soundness of disassembly. Based on this definition, we implement a tool called DSV (short for Disassembly Soundness Validation) to validate whether a binary has been soundly disassembled or not. DSV takes a binary file and the assembly file disassembled from the binary file as inputs, generates "sound" or "unsound" as output, and reports all the "unsound" disassembled instructions. A key characteristic is that DSV does *not* assume a ground truth; in other words, DSV does not presume the availability of source code or debug information.

Essentially, DSV performs a recursive traversal starting at the binary's entry point while validating all reached instructions. DSV over-approximates the semantics of the binary under investigation in two ways. First, the semantics of various instructions are over-approximated by treating their effects on certain state parts as unknown. Second, the jumps and paths that can be traversed at runtime are statically over-approximated. DSV needs to deal with three key problems: unbounded loops, pointer aliasing, and indirect-branch instructions. In order to deal with loops, we employ *bounded model checking* (BMC) [11]. To handle the pointer aliasing problem and indirect branches, we use *concolic execution* [12].

We apply DSV to all the binaries of Coreutils library for eight different disassemblers. Soundness issues are found in each of them. Some examples include:

1. Incorrectly recovering instructions, e.g., Ghidra [3] disassembles `49 0f a3 c8` to `bt rax,rcx` while the correct result should be `bt r8,rcx`;
2. Incorrectly recovering immediate values in operands, e.g., Dyninst [13] translates `48 b8 ff ff ff ff ff` to `mov rax, 0x4611686018427387903`, however, the valid instruction is `movabs rax,0x3fffffffffffffff`;
3. Missing instructions due to under-approximating indirect control flow transfers.

The contribution of this paper consists of:

1. A formal definition for the soundness of disassembly.

2. An automated methodology called DSV (for: Disassembly Soundness Validation) for validating whether the output of a black-box disassembler is sound w.r.t. a binary.
3. The application of this methodology to 102 binaries of Coreutils, each for eight different disassemblers: angr 8.19.7.25 [2], BAP 1.6.0 [4], Ghidra 9.0.4 [3], objdump 2.30, radare2 3.7.1 [1], Dyninst 10.2.1 [13], IDA Pro 7.6, and Hopper 4.7.3.

Paper Organization We discuss past and related work in Section 2. In Section 3, we introduce a soundness definition for the disassembly process and discuss the definition's validity. Section 4 illustrates DSV's implementation details. We discuss soundness issues in existing disassemblers detected by DSV in Section 5. Section 6 reports experimental results obtained by applying DSV to the Coreutils library. We conclude in Section 7.

2 Past and Related Work

We first discuss the main approaches to disassembly. Then, the approaches for the validation of disassembly are discussed.

2.1 Disassembly Techniques

Linear sweep and recursive traversal are the major techniques behind the binary disassembly process. PSI [14] and objdump are typical linear-sweep disassemblers. These disassemblers handled the byte sequences in the binaries sequentially. Linear-sweep disassemblers have superior performance under certain circumstances. For example, some linear sweep disassemblers fulfilled a 100% correctness on SPEC CPU2006 benchmarks generated by gcc and clang [10]. However, linear sweep disassemblers have poor performance in handling special situations such as overlapping instructions, inline data, and jump tables.

On the other hand, disassemblers such as IDA pro, Dyninst [13], Ghidra [3], and Hopper were implemented using recursive traversal. These disassemblers decoded the instructions following the execution path of the sequential and branching instructions and tried to resolve the indirect jump addresses. Essentially, they reconstructed the *control flow* on-the-fly in order to perform disassembly. Recursive traversal handles overlapping instructions and inline data in a more reliable way than linear sweep disassemblers.

2.2 Soundness Validation

Andriesse et al. [10] checked the false positive and false negative rates for nine mainstream disassemblers using SPEC CPU2006 and Glibc-2.22 as the benchmarks. The researchers gave a comprehensive comparison between different disassemblers on five critical criteria, including instruction recovery, function starting address relocation, function signature restoration, control flow graph (CFG),

and callgraph reconstruction. They used the ground truth information derived from LLVM analysis, DWARF debugging information, and some manual ancillary work. These ground truths provided critical information for the five criteria.

Paleari et al. [15] developed a methodology called n-version disassembly to apply differential analysis to verify the correctness of different x86 disassemblers. The writers employed various disassemblers to recover the instruction from the same string of bytes and compared the results to find out the divergences. This paper validates the correctness of single-instruction disassembly, whereas our paper focuses on a complete disassembly process.

Pang et al. [16] manually evaluated the code base of various disassemblers and discussed the algorithm and heuristics used by these disassemblers. They also studied 3,788 binaries from different sources on nine main-stream disassemblers to evaluate the instruction recovery, cross-reference accuracy, function starting point, and CFG construction. They reported incorrectly disassembled cases existing in these disassemblers. The ground truths were automatically collected in the compiling and linking procedures when generating binaries with a method similar to the technique used by Andriesse et al. [10].

3 Definition of Disassembly Soundness

In this section, we provide a definition of the soundness of a disassembly process. Moreover, we discuss a crucial assumption required to ensure that this definition reflects the correctness of a disassembly process without ground truth.

3.1 Soundness Definition

To formulate a formal notion of disassembly soundness, we first introduce the types and notations used in the definition. An element of type `Nword` is a bit vector with size `N`. Given a bit vector w, notation $|w|$ provides the size of the bit vector. The type `Instruction` indicates the type of valid x86-64 instructions. In our soundness definition, an instruction is represented by, among other things, an opcode mnemonic, its operands with size directives, and possibly certain prefixes.

The definition of soundness is based on three components: a function $\mathsf{read_bytes}$ that reads bytes from a binary file, a function $\mathsf{bytes_of}$ that assembles a single instruction into bytes, and an abstract transition relation $\rightarrow_A$.

The first function $\mathsf{read_bytes}$ reads, given an address and a size, a byte sequence from the binary file. In all the following definitions, the type of the address is expressed as `64word`, and the type of byte is `8word`. Then the type annotation of $\mathsf{read_bytes}$ is represented as:

$$\mathsf{read_bytes} : \mathtt{64word} \mapsto \mathbb{N} \mapsto [\mathtt{8word}]$$

Function $\mathsf{bytes_of}$ maps a single instruction to the corresponding byte sequence representation, which is the essential work of any assembler. Although

the $\mathsf{bytes_of}$ function represents an assembly process, our soundness definition does not consider any specific implementation of an assembler. Function $\mathsf{bytes_of}$ is type-annotated as:

$$\mathsf{bytes_of} : \mathtt{Instruction} \mapsto [\mathtt{8word}]$$

Let $\rightarrow_C$ denote a deterministic concrete transition relation over concrete addresses, and $\rightarrow_C^*$ represents the transitive closure of this transition relation. Modeling this concrete transition relation is impossible: the relation depends on the current state of registers, memory, and flags, but also on the state of peripherals, the OS, etc. Let a_0 be a binary's entry address. An instruction address a is *reachable* at run-time, if and only if:

$$a_0 \rightarrow_C^* a$$

The soundness definition is based on an over-approximative abstraction of this concrete transition relation, which is defined as $\rightarrow_A$. This is a non-deterministic transition relation over addresses: $\rightarrow_A$ is of type $\mathtt{64word} \mapsto \{\mathtt{64word}\}$. This transition relation solely concerns the 64-bit value of the instruction pointer `rip` of the concrete state and produces a set of next instruction addresses.

Definition 1. *Transition relation* $\rightarrow_A$ is a proper abstraction of *concrete transition relation* $\rightarrow_C$*, if and only if, for any reachable concrete states* s *and* s'*:*

$$s \rightarrow_C s' \implies \mathtt{rip}(s) \rightarrow_A \mathtt{rip}(s')$$

We use $\rightarrow_A^*$ to indicate the transitive closure of $\rightarrow_A$.

Finally, the input of our soundness definition is the output of a disassembler. This output essentially is a partial mapping from byte sequence to instructions. It is denoted as disasm. We also define an auxiliary function $\mathsf{disasm_n}$. Function $\mathsf{disasm_n}$ returns, given the current address, the size of bytes that are to be disassembled for the next single instruction. The two functions are of type:

$$\mathsf{disasm} : [\mathtt{8word}] \mapsto \mathtt{Instruction}$$

$$\mathsf{disasm_n} : \mathtt{64word} \mapsto \mathbb{N}$$

Definition 2. *Let* a_0 *be a binary's entry address and let* disasm *be some disassemblers' output. Output* disasm *is* sound*, if and only if:*

$$\begin{gathered}\forall a \cdot a_0 \rightarrow_A^* a \implies \mathsf{bytes_of}(\mathsf{disasm}(\beta)) = \beta \\ \textbf{\textit{where}} \quad \beta = \mathsf{read_bytes}(a, \mathsf{disasm_n}(a))\end{gathered}$$

Definition 2 indicates that for all reachable addresses a inside a binary file, the bytes β of the disassembled instruction $\mathsf{disasm}(\beta)$ located at address a are equal to the actual bytes that are read from the binary. If there exist some reachable instructions whose bytes are not equal to those in the binary, the disassembler is unsound.

This definition is independent of the inner mechanism of a disassembler. Whether a disassembler is implemented using recursive traversal, linear sweep, or machine-learning is irrelevant since we only try to validate the consistency between a binary file and the output of the disassembler.

3.2 Loose Comparison of Instruction Bytes

For each reachable instruction address, Definition 2 compares the bytes produced by reassembling a disassembled instruction with the original bytes from the binary. However, a strict byte-by-byte comparison may incorrectly classify a disassembler as unsound. Consider Figure 1. The original assembly process is modeled as a asm function, which maps an instruction to the corresponding bytes. This function is part of the trust base, but it is not available.

$$\mathsf{asm} : \texttt{Instruction} \mapsto [\texttt{8word}]$$

The ground truth is the original instruction i_0, assembled by the original assembler asm to b_0. Both i_0 and asm are assumed to be unavailable. The black-box disassembler disasm produces an instruction i_1 from b_0. Definition 2 suggests that it suffices to reassemble instruction i_1 into bytes b_1 and then strictly compare b_0 and b_1 to validate the soundness.

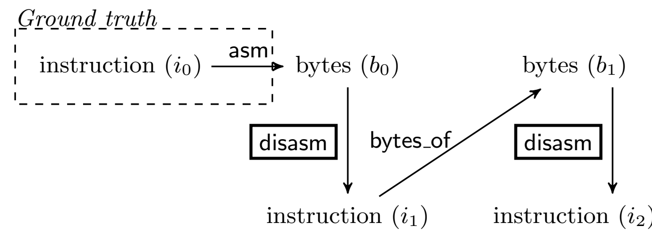

Fig. 1. Comparison per instruction. The dashed box indicates that the ground truth, i.e., the original instruction and original assembler, are unavailable. The disassembler under investigation (disasm) is black-box.

This, however, is not necessarily correct for two reasons. First, the function disasm may produce an instruction different from i_0 but with the same semantics. In such a case, reassembling may not reproduce the same bytes. Second, function bytes_of may be different from the original assembler asm (since that function is unavailable). Thus, even if the disassembler under investigation disasm was able to reproduce the exact instruction i_0, a strict comparison between b_0 and b_1 may still fail in the soundness validation.

Listing 1.1. An example that does not satisfy the soundness definition.

```
objdump(0f1f440000) = nop DWORD PTR [rax+rax*1+0x0]
gcc(nop DWORD PTR [rax+rax*1+0x0]) = 0f 1f 04 00
objdump(0f1f0400) = nop DWORD PTR [rax+rax*1]
```

For example, we employ gcc as the assembler and objdump as the disassembler and get the example in Listing 1.1. In this example, b_0 is `0f 1f 44 00 00`, b_1 is

`0f 1f 04 00`. They are not equivalent. If we solely compare b_0 and b_1, we will make the wrong declaration that the disassembly process carried out by objdump is not sound. However, the disassembled result is sound since `nop DWORD PTR [rax+rax*1+0x0]` and `nop DWORD PTR [rax+rax*1]` are semantically equivalent. The reason behind this situation is that gcc would automatically apply optimization when it encounters certain types of instructions.

Thus, instead of a strict comparison, we will use a loose comparison of bytes. The bytes b_1 produced by reassembling are again disassembled. This produces instruction i_2. We will consider b_0 and b_1 loosely equal if these instructions are equal *after normalization*. The *normalization* is executed by a normalize function, which rewrites an instruction to a normalized format following rules such as reformatting assembly code from AT&T format to Intel, removing `*1` and `+0`, and normalizing the representation of memory accesses. The normalized instruction is ensured to be semantically equivalent to the original instruction.

Definition 3. *Let β_0 and β_1 be two byte-sequences. They are* loosely equivalent*, notation $\beta_0 \simeq \beta_1$, if and only if:*

$$\begin{aligned} &\beta_0 = \beta_1 \vee \mathsf{normalize}(i_0) = \mathsf{normalize}(i_1) \\ &\textbf{\textit{where}} \quad i_0 \coloneqq \mathsf{disasm}(\beta_0), \\ &\qquad\qquad i_1 \coloneqq \mathsf{disasm}(\beta_1) \end{aligned}$$

We can now summarise a fundamental part of the TCB of our approach. Since there is no ground truth, this must be assumed and cannot be proven.

Assumption 1. *For any instruction i_0:*

$$\mathsf{asm}(i_0) \simeq \mathsf{bytes_of}(\mathsf{disasm}(\mathsf{asm}(i_0)))$$

implies that instruction i_0 has been correctly disassembled by function disasm*.*

4 Validation Algorithm

In Section 3, we define the soundness of the output of a disassembler w.r.t. the original binary file. According to that definition, there are three components that must be implemented: read_bytes, bytes_of, and the abstract step function $\rightarrow_A$.

The first two are straightforward. For read_bytes, we employ the `readelf` utility to get the binary segment information and implement a Python program to read a byte sequence from a binary file directly. To implement function bytes_of, we need to translate *a single instruction* to its byte-sequence representation. The choice of the assembler, whether gcc, clang, or some other, is independent of the disassembler under investigation and of the type of the source binary file.

The third component, an abstract transition relation $\rightarrow_A$, is more involved. A perfect and exact implementation of this component does not exist since it is undecidable which addresses are reachable from the entry point [7]. It is also undecidable to distinguish instructions from raw data [8]. Implementation of $\rightarrow_A$ requires, among other things, dealing with indirect jumps and calls, jump tables, data inlined in code, and overlapping instructions. Specifically, predicting where an indirect branch jumps to is a major challenge for all existing disassemblers.

4.1 Consequences of An Inexact Abstract Transition Relation

We thus, necessarily, implement an *inexact* abstract transition relation. We will use $\rightsquigarrow_A$ to denote this inexact implementation of the hypothetical exact abstract transition relation $\rightarrow_A$. We introduce the following terminology (here a_0 denotes the binaries' entry point):

White An instruction address a is white if it is deemed reachable by the implementation $\rightsquigarrow_A$, i.e.:

$$a_0 \rightsquigarrow_A^* a$$

We can now rephrase the notions of false positive and false negative w.r.t. this terminology. A *false positive* occurs when disassembler-output is deemed sound by DSV, whereas it is incorrect. We define a false positive as the existence of an incorrectly disassembled reachable instruction that is not white. It is thus reachable at runtime and deemed unreachable (and therefore missed) by the implementation $\rightsquigarrow_A$. A *false negative*, then, is an incorrectly disassembled unreachable instruction that is white. In other words, it is deemed reachable by the implementation $\rightsquigarrow_A$, but unreachable at runtime.

A false positive can happen if the implementation $\rightsquigarrow_A$ *under-approximates* the concrete transition relation $\rightarrow_C$. In other words, it can happen if it is possible that a reachable instruction is not white. We aim for an implementation that does not suffer from false positives, and therefore require the implementation to be proper (see Definition 1): any reachable instruction is visited. In the case of proper over-approximation, a false negative can happen, i.e., an unreachable instruction may be white.

Finally, we would like to note that there is no decidable way to determine whether an instruction address is reachable or not. There is no ground truth and no reliable way of establishing reachability without source code. In practice, however, it is possible to establish the unreachability of certain parts of the binary. For example, in the current implementation, functions called inside an external `__cxa_atexit` function are not considered to be reachable (e.g., deconstructors). We thus use the following terminology:

Black An instruction address is black if it is not white and *it can be established* (e.g., with conservative manual inspection) that it is unreachable.

Grey An instruction address is grey if it is not white and it is not black, i.e., if it cannot be established whether it is reachable or not.

Given an over-approximative implementation $\rightsquigarrow_A$, all instruction addresses reported by some disassembler are either white, black, or grey. The aim is to construct an implementation $\rightsquigarrow_A$ that minimizes the number of grey instructions. Only the case where DSV finds an issue in a grey instruction constitutes a false negative.

4.2 DSV Overview

In essence, DSV employs a standard forward BMC exploration loop. At all times, three parameters are maintained:

s**: the current state.** A symbolic state is maintained that contains symbolic expressions for registers, flags, and memory. The initial state solely consists of an assignment of some concrete values to the stack pointer `rsp` and the instruction pointer `rip`.

π**: the current path constraint.** A symbolic predicate is maintained that contains the branching conditions of the current path. Its purpose is to prune inconsistent paths (we check the consistency using the Z3 SMT Solver [17]). Initially, this constraint is `true`.

Σ**: the stored states.** A key-value mapping with as keys instruction addresses and as values symbolic states. This mapping allows DSV to keep track of which addresses have been visited and to reduce the traversed state space. Initially, this mapping is empty.

DSV first fetches the instruction i as disassembled by the disassembler under investigation and validates that instruction (see Section 3.2). It then updates Σ by adding the current state σ. It may be the case that the current instruction address was already visited. In that case, a *merge* must happen between the current state s and the stored state. If the current state s and the merged state *agree* (intuitively: they contain the same information), then no further exploration is necessary. If the instruction address was unvisited, the current state is simply inserted into Σ. DSV then concolically executes instruction i to the merged state s_m, given the current path constraint π. This provides a set of pairs of symbolic states and path constraints; one instruction may induce multiple paths. Each of these pairs is explored.

4.3 State and Memory Model

The state consists of assignments of symbolic expressions to flags, registers, and memory. Symbolic expressions consist of expressions with a standard set of operators (e.g., $+$, $-$, $\ldots$) and as base operands either immediate values, registers, or flags. Most notably, a symbolic dereference operator is supported that reads data from memory. An operand may also be an unconstrained universally quantified variable. We will use v_f to denote a fresh variable. The symbolic expressions used by DSV are close to that used in existing literature [18].

Since the bit length of all registers is fixed, we model general-purpose registers as a 64-bit Z3 bit-vector and deal with register aliasing accordingly. We set the initial values of all the registers, except for `rip` and `rsp`, to symbolic values and modify the values of registers according to the semantics of instructions. The value of each register can be either symbolic or concrete.

There are different techniques to model memory. To design a space-efficient memory model that simulates the memory changes during the execution of a binary, we model memory as a function `mem` of type `64word` $\mapsto$ ([`8word`], $\mathbb{N}$). This function maps memory addresses to byte sequences and the size of the region starting at the given address. Function `mem` is partial, which means that not all addresses at the memory have explicit content. At all times, all regions in the range of `mem` are separate.

Since we keep the stack pointer concrete, all local variables correspond to memory regions with concrete addresses. The same holds for global variables. Moreover, the Glibc functions malloc and calloc are modeled in such a way that they return a *concrete* address that does not overlap with any existing region in the memory. This concretizes the majority of addresses. Theoretically, this approach may lead to unsoundness issues: for example, if a program successfully allocates memory using malloc, then branches are taken based on whether that (non-null) pointer is greater than some immediate value. To the best of our knowledge, such behavior is undefined according to the C standard.

Assumption 2. *We assume that the control flow of a binary does not depend on the concrete values returned by memory allocation functions or on the concrete value of the stack pointer.*

However, not all memory addresses are concrete: symbolic addresses occur when pointers are returned by external functions that are not linked statically. In these cases, reading from a symbolic memory region returns a fresh symbol. Writing to such a memory region will remove all heap-related regions from the memory but will keep the local stack frame intact.

4.4 Merging and Agreeing

If the address of the current state s was already visited, the current state s and the visited state s_{old} are merged (see Algorithm 1). If the current value v at a key k in s is symbolic, then v is possible to represent any value, and we do not need to change it. However, if the current value v is concrete, we need to compare v with v_{old} at the same key k in s_{old} to decide how to merge v and v_{old} to get the new result.

Algorithm 1 Merging algorithm.

```
 1: function MERGE(s_old, s)
 2:     s_new ← copy(s)
 3:     for all (k, v) ∈ s do
 4:         v_old ← s_old[k]
 5:         if v is a concrete value then
 6:             if v_old is a concrete value then
 7:                 if v ≠ v_old then
 8:                     s_new[k] ← fresh variable
 9:                 end if
10:             else
11:                 s_new[k] ← fresh variable
12:             end if
13:         end if
14:     end for
15:     return s_new
16: end function
```

The current state s is not explored if state s and merged state s_m contain the same information, i.e., if the two state *agree*. Two states agree if they have the same keys and for any key-value pair (k, e) in s and (k, e_m) in s_m the expression e and e_m agree.

Definition 4. *Let* $\mathsf{fresh}(e)$ *denote the set of fresh variables in symbolic expression* e*. Two expressions* e_0 *and* e_1 agree *if and only if there exists a bijection* β *between* $\mathsf{fresh}(e_0)$ *and* $\mathsf{fresh}(e_1)$*, such that* e_0 *and* e_1 *are syntactically equal if all fresh variables* v_f *in* e_0 *are replaced with* $\beta(v_f)$*.*

Example 1. Consider a loop in which register `rax` is incremented with 4 every iteration. Let the visited state $s_{old} = \{\texttt{rax} := v_{f_0}, \texttt{rdi} := v_{f_0} + 100\}$. After one loop iteration, the current state $s = \{\texttt{rax} := v_{f_0} + 4, \texttt{rdi} := v_{f_0} + 100\}$. The merged state will be $s_m = \{\texttt{rax} := v_{f_1}, \texttt{rdi} := v_{f_0} + 100\}$ and will be stored. States s_m and s do not agree and exploration will continue. However, after one more iteration, we will obtain state $s' = \{\texttt{rax} := v_{f_1} + 4, \texttt{rdi} := v_{f_0} + 100\}$. States s' and state s_m will be merged, resulting in $s'_m = \{\texttt{rax} := v_{f_2}, \texttt{rdi} := v_{f_0} + 100\}$. States s_m and s'_m do agree, and therefore the loop is not unrolled further.

4.5 Instruction Semantics

There is no need to set up complete semantics for *all* instructions. In our implementation, instruction semantics is constructed to change the value of the `rip` register to guide the symbolic execution. We only need to build up semantics for instructions that – be it directly or indirectly – influence the `rip` register. We will call this the set of *relevant* instructions.

The set of *relevant* instructions include `push`, `pop`, `mov`, `lea`, `call`, `ret`, simple arithmetic instructions, logical instructions, bitwise instructions, jump instructions, etc. According to the statistics taken in some literature [19], these instructions would make up over 96% of instructions in multiple C/C++ applications and web browsers. Advanced instructions such as floating-point instructions and SIMD extensions typically do not impact register `rip`. It is not necessary to construct specific semantics for these instructions.

For all the irrelevant instructions, we use *unknown* semantics by assigning fresh variables any time an irrelevant instruction is executed. In most cases, an instruction has an opcode and different operands, and the content of the destination operand is modified by the instruction. For irrelevant instructions, the semantics assigns some fresh variable v_f to the destination operand, representing that the current status of the corresponding register, flag, or memory is undefined or undetermined. The fresh variables are handled using the symbolic execution rules in our DSV SE engine.

4.6 Concolic Execution

As discussed in Section 4.3, we make use of concolic execution that concretizes memory addresses as much as possible while leaving the remainder as symbolic as

possible. As such, the branching conditions that are taken are generally symbolic. In the case of a conditional jump based on a symbolic flag value, both paths are taken (sequential execute and jump). This over-approximates reachability.

A key challenge is to resolve indirect-branch addresses. An indirect branch is a control flow transfer (jump or call) where the target is computed instead of an immediate. Indirect branches happen, e.g., in the case of compiled switch statements, function callbacks, or virtual tables. Three cases may arise:

1. The current state is sufficiently concrete that the computation can be resolved. In this case, exploration continues.
2. The expression that computes the next value of `rip` is symbolic, but the current state and the path constraint contain sufficient information to both bind and over-approximate the set of next addresses. In this case, exploration continues to all next addresses.
3. The current state does not contain sufficient information to bind the set of next addresses; the expression that computes `rip` contains unbounded symbolic values. An error message is produced, and we manually investigate how to resolve the issue. Generally, we need to trace back and see which irrelevant instructions need to be considered relevant. This situation is infrequent since we have modeled the semantics of the most common instructions based on their usage rate.

With the state model for registers, flags, and memory, we carry out the concolic execution to construct a CFG for the machine code. Concolic execution is over-approximative. The vast majority of branches are taken due to symbolic conditions. Meanwhile, `rsp` is always concrete, and therefore local variables in the stack frame can be read/written. Besides, addresses are concrete in the memory allocation functions. The concrete addresses prevent memory aliasing issues.

In the construction of CFG, indirect jump, indirect call, and return instructions pose a challenge in how to resolve the indirect-branch addresses. The path constraint provides a bound on the set of next addresses. Besides, we introduce a trace-back model to fix the problem of unimplemented instruction semantics. We also implement an algorithm [20] to solve the challenge of jump table without determined upperbound. However, there still exist unresolved indirect-branch addresses in the concolic execution since it is an undecidable problem.

5 Soundness Issues Exposed by DSV

This section summarises some of the soundness issues found by DSV. We mainly focus on instructions that are erroneously recovered by different disassemblers.

In Section 6.1, we use DSV to evaluate the disassembly results generated by eight disassemblers on the Coreutils library. Even though most of the reachable instructions for these disassemblers are correctly recovered, there are few exceptions where the disassembled instruction is incorrect w.r.t. the byte sequence. We report on some cases found by DSV that are inappropriately disassembled by certain disassemblers. Table 1 summarises the found results, which are disagreed

for different disassemblers. Some of the disagreements (row 1, 2 of the table) are trivial and can be argued not to impact soundness. Row 3, 4, 5, and 6 of the table consist of actual soundness issues.

Table 1. Examples of instruction recovery results for different disassemblers. All the results are normalized to Intel format.

<table>
<tr><th>bytes</th><th>objdump</th><th>radare2</th><th>angr</th><th>Hopper</th><th>BAP</th><th>IDA Pro</th><th>Ghidra</th><th>Dyninst</th></tr>
<tr><td>f3c3</td><td>repz ret</td><td colspan="3">ret</td><td>rep ret</td><td>rep retn</td><td>ret</td><td>rep ret</td></tr>
<tr><td>4881a4249000
0000fffbffff</td><td colspan="6">and qword ptr [rsp+0x90],
0xfffffffffffffbff</td><td colspan="2">and qword ptr [rsp+
0x90],0xfffffbff</td></tr>
<tr><td>4899</td><td colspan="7">cqo</td><td>cdq rax</td></tr>
<tr><td>4d0fa3f7</td><td colspan="6">bt r15,r14</td><td>bt rdi
,r14</td><td>bt r15,r14</td></tr>
<tr><td>48be00000000
00f0ffff</td><td colspan="5">movabs rsi,
0xfffff00000000000</td><td colspan="2">mov rsi,0xffff
f00000000000</td><td>mov rsi,0x-17
592186044416</td></tr>
<tr><td>64488b042528
000000</td><td colspan="7">mov rax,qword ptr fs:[0x28]</td><td>mov rax,0x28</td></tr>
</table>

Row 1 and 2 of Table 1 mainly concern different representations of the same semantical intent. There are cases where the operands of an instruction are not represented since default behavior is assumed. For instance, both Ghidra and Dyninst (correctly) assume that immediates are sign-extended to fit the destination operand, if necessary. However, minor differences may be relevant. For example, the instructions `repz ret` and `ret` have the same semantical intent but their execution time may differ for certain architectures.

Row 3, 4, 5, and 6 concern semantically different recovered instructions. For instance, Dyninst disassembles 4899 to `cdq rax`, which is not a valid instruction in x86-64 ISA (note that `cdq` performs sign-extension to 64 bits, whereas `cqo` performs sign-extension to 128 bits). An example is shown where Ghidra misrepresents a register (`rdi` instead of `r15`). Besides, a 64-bit immediate is wrongly disassembled by Dyninst. Finally, Dyninst sometimes seems to omit representations of segment registers such as `ds` and `fs`.

Except for the examples listed in Table 1, there are some ambiguous cases for different disassemblers. The outputs generated by Dyninst do not have any `ptr` operator to indicate the operand size of a memory operand, which leads to ambiguous semantical behavior. For example, 49837c242800 is translated to `cmp [r12 + 0x28],0x0` by Dyninst while the other disassemblers' result is `cmp qword ptr [r12+0x28],0x0`. Without the `qword ptr` specifying the size of the operand as 64-bit, we cannot determine what the exact value reading from the memory is. Thus the result of the `cmp` instruction is undetermined.

6 Experimental Results

In Section 6.1, we apply DSV on eight different disassemblers: objdump 2.30, radare2 3.7.1, angr 8.19.7.25, BAP 1.6.0, Hopper 4.7.3, IDA Pro 7.6, Ghidra 9.0.4,

and Dyninst 10.2.1, using 102 test cases from Coreutils-8.31. Here, we evaluate the performance of DSV.

All these experiments are carried out on a machine with Intel Core i7-7500U CPU @ 2.70GHz × 4 and 16GB RAM. The OS is Ubuntu 20.04.2 LTS, and the Coreutils-8.31 library is compiled using gcc 7.5.0 through the standard build process.

6.1 Coreutils Library

We apply DSV on 102 test cases in the Coreutils library, which are disassembled using eight disassemblers. For each test case, we report the number of instructions: total, white, gray, and black. The definition of *white*, *black*, or *grey* instructions are given in Section 4.1. Roughly speaking, white indicates instructions that are proven to be reachable by DSV, and black illustrates unreachable instructions. The grey instructions are those that are reported by the disassembler but are not visited by DSV; the reachability of these instructions is unknown. Table 2 shows the results of `basename`, `expand`, `mknod`, `realpath`, and `dir` test cases in the Coreutils library for different disassemblers. These 5 test cases are selected based on the number of total instructions and the diversity of various instruction types.

Instruction Recovery Most disassemblers are capable to correctly disassemble all the reachable instructions. As shown in Figure 2, for most of test cases in Coreutils library, objdump, angr, BAP, and IDA Pro achieve an accuracy rate of 100% for single-instruction recovery. Meanwhile, Ghidra and Dyninst make some errors in the disassembly process for some test cases, and the accuracy would decrease to around 97.5%.

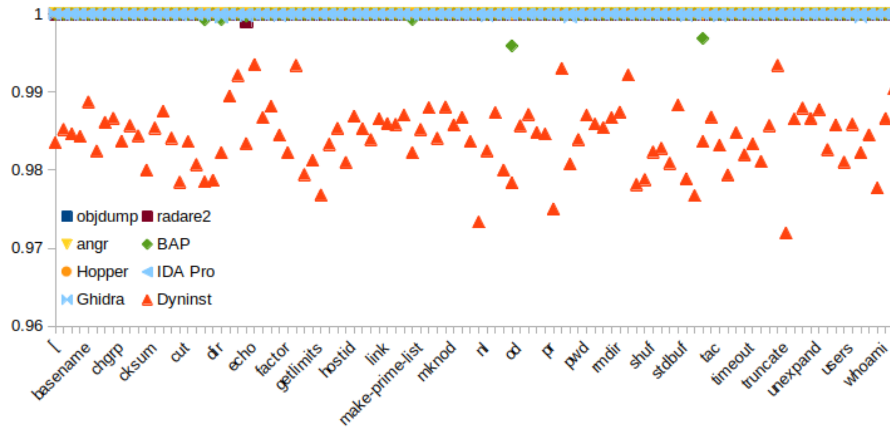

Fig. 2. Ratio of correctly disassembled vs. the white disassembled instructions.

Control Flow Recovery For all test cases, there exists a gap between the number of white instructions, which are reachable instructions detected by DSV, and the number of total instructions; in other words, the number of black instructions can be relatively high. This can be accounted for two reasons.

The first reason is that different disassemblers consider different parts of the binary. For example, BAP generates the instructions from sections .symtab, .debug_line, .debug_ranges, and so on, while some disassemblers may solely generate instructions from .text, .plt, and .plt.got sections.

The second reason lies in the technique that DSV employs to handle external functions. DSV treats external functions as black boxes and does not go inside the external functions to execute them. Internal functions that are called by external functions may be considered black. For example, the internal function `close_stdout` is called by the external function `__cxa_atexit` (it calls the close function after program exit). Thus, the `close_stdout` function is considered black. Some exceptions include `__libc_start_main` and `pthread_create`. These two external functions execute the function pointer passed through the `rdi` register, and the internal functions pointed to are not executed by DSV. Broader coverage, i.e., fewer black instructions, can be reached by providing semantics to external functions that call internal ones.

The ratio of grey vs. white instruction is an indication of how accurate control flow has been recovered. If the ratio is low (zero), then the disassembler highly accurately decides which instructions are reachable and which are not. If it becomes higher, this may indicate either that the disassembler coarsely over-approximated which instructions are reachable (many grey instructions), or that the disassembler missed instructions. The ratio is on average about 4%. As shown in Figure 3, BAP usually has the highest ratio since the instructions whose addresses are stored in indirect jump tables are missed by BAP due to lack of support for indirect branching. Meanwhile, objdump and angr have similar ratio for most of test cases , as we use angr to statically generate a CFG (CFGFast) and to disassemble a binary file, which have similar outputs as objdump.

The amount of white instructions per disassembler is an indication of how many instructions have been reached. objdump, radare2, angr, and Ghidra have similar numbers of white instructions. Meanwhile, BAP has smaller results in all these test cases since it does not employ any heuristics to solve the indirect branch problem caused by the jump table. The results for Dyninst are unstable because there are some instruction-recovery errors in the disassembly results.

Soundness Results Most disassemblers are sound for most of the test cases. We find that Ghidra sometimes incorrectly recovers instructions. There are three other major exceptions.

First, BAP does not resolve indirect branches. Since BAP essentially reports an empty set of next addresses for indirect jump tables – whereas DSV wants to continue exploration – DSV reports a soundness issue. We marked these as missing instructions: the issue is not that BAP incorrectly recovers instructions, but that it misses instructions by “under-approximating” control flow.

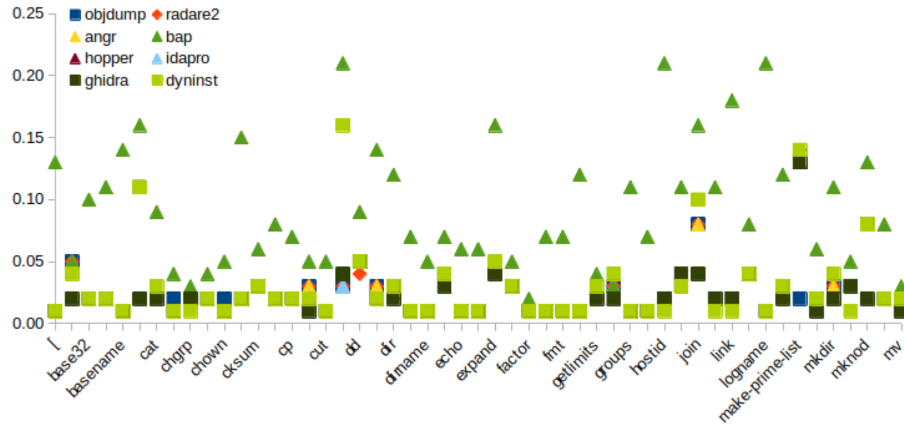

Fig. 3. Ratio of grey instructions to white for different disassemblers.

Additionally, radare2 sometimes translates instructions to data. For example, in `dir` test case, radare2 disassembles the bytes `ff2552c72100` at address `3888` to data `.qword 0x90660021c75225ff`, which should be translated to a `call` instruction to `malloc`. This kind of mistranslation leads to missing instructions.

In some situations, Hopper is not capable to correctly determining the instruction boundaries. For example, in `dir` test case, at address `0xf2a8`, the disassembler should generate an instruction `sub r12d,0x1`. However, Hopper classifies it as data and continues the disassembly process from address `0xf2a9`.

Another exception is Dyninst. There are various examples showing that Dyninst involves errors in instruction recovery. These errors may cascade since incorrectly recovering instructions may also lead to incorrectly assessing which instruction addresses are to be disassembled. For instance, Dyninst cannot recover control flow for the `seq` test case from the Coreutils library since incorrectly recovered instructions lead to unrealistic paths.

7 Conclusion

Disassembly is a challenging and undecidable problem that lies at the base of various research in reverse engineering, formal verification, binary hardening, and security analysis. Even state-of-the-art disassemblers that have been elaborately designed and tested have soundness issues, such as whether a disassembly accurately reflects the semantical behavior of the binary under investigation. We propose a definition for soundness of the output of a disassembler w.r.t. the original binary. Moreover, we propose DSV, a tool for validating whether a binary has been correctly disassembled. DSV finds incorrectly disassembled instructions and assesses whether the disassembler under investigation could determine at which addresses instructions need to be recovered correctly.

DSV does not assume the existence of ground truth in the form of source code, an LLVM representation, or debugging information. We, therefore, necessarily make assumptions and aim to provide an explicit insight into the trusted codebase. The trusted codebase of DSV contains two key assumptions. First, we assume that the proposed way of loosely comparing byte sequences allows DSV to decide whether a single byte sequence correctly corresponds to a single instruction. Second, DSV employs concolic execution leaving certain parts, such as the stack pointer, concrete. It is assumed that leaving these parts concrete does not influence the reachability of instruction addresses.

DSV has been applied to validate the output of eight state-of-the-art disassembler tools on 102 binaries of Coreutils library. Soundness issues were exposed, ranging from incorrect instruction recovery to incorrectly recovered control flow of the binary (leading to missing instructions).

Future Work: DSV essentially is a binary exploration tool. We argue that DSV demonstrates that the combination of bounded model checking and concolic execution is very applicable in the context of stripped binaries as it mitigates the complexity of some fundamental issues. Even though its current version solely focuses on the validation of disassembly, we aim to use the core algorithm and concepts of DSV for other binary exploration efforts. For example, We aim to use DSV for validating the correctness of generated control flow and call graphs, and generally for exposing "weird" edges [21] and security vulnerabilities in binaries. Currently, DSV is restricted to binaries with the x86-64 format. Since our formal definition is general, we intend to extend our implementation and validation efforts to other ISAs, such as ARM.

Source Code Availability

The complete source code, benchmarks, and experimental results are open-sourced and available at the project website: https://ssrg-vt.github.io/DSV. The source code artifact is archived with a DOI link at:
https://doi.org/10.5281/zenodo.6380975.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which greatly improved the paper. This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090028 and contract N6600121C4028, and the US Office of Naval Research under grants N00014-17-1-2297 and N00014-18-1-2665.

References

1. Radare2: Unix-like reverse engineering framework. https://github.com/radareorg/radare2 (2021)

2. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157. IEEE (2016)
3. Rohleder, R.: Hands-on ghidra-a tutorial about the software reverse engineering framework. In: Proceedings of the 3rd ACM Workshop on Software Protection. pp. 77–78 (2019)
4. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: International Conference on Computer Aided Verification. pp. 463–469. Springer (2011)
5. Park, J., Xu, X., Jin, Y., Forte, D., Tehranipoor, M.: Power-based side-channel instruction-level disassembler. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2018)
6. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making reassembly great again. In: NDSS (2017)
7. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953)
8. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M.: Shingled graph disassembly: Finding the undecideable path. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 273–285. Springer (2014)
9. Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 24–35 (2016)
10. Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 583–600 (2016)
11. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking (2003)
12. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)
13. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools. pp. 9–16 (2011)
14. Zhang, M., Qiao, R., Hasabnis, N., Sekar, R.: A platform for secure static binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 129–140 (2014)
15. Paleari, R., Martignoni, L., Fresi Roglia, G., Bruschi, D.: N-version disassembly: differential testing of x86 disassemblers. In: Proceedings of the 19th international symposium on Software testing and analysis. pp. 265–274 (2010)
16. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. arXiv preprint arXiv:2007.14266 (2020)
17. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
18. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
19. Akshintala, A., Jain, B., Tsai, C.C., Ferdman, M., Porter, D.E.: X86-64 instruction usage among c/c++ applications. In: Proceedings of the 12th ACM International Conference on Systems and Storage. pp. 68–79 (2019)

20. Cifuentes, C., Van Emmerik, M.: Recovery of jump table case statements from binary code. Science of Computer Programming **40**(2-3), 171–188 (2001)
21. Shapiro, R., Bratus, S., Smith, S.W.: "weird machines" in ELF: A spotlight on the underappreciated metadata. In: 7th USENIX Workshop on Offensive Technologies (WOOT 13) (2013)

Table 2. Execution results for Coreutils library on different disassemblers. Only 5 of 102 binaries are shown.

		Number of total	Number of white	Number of grey	Number of black	Ratio of grey vs. white	Number of indirects	Missing instr	Sound
objdump	basename	3310	2217	18	1075	0.01	59		
	expand	3928	2742	112	1074	0.04	79		
	mknod	4101	2775	216	1110	0.08	65		
	realpath	5828	2644	89	3095	0.03	72		
	dir	19029	12751	417	5861	0.03	230		
radare2	basename	3409	2217	18	1174	0.01	59		
	expand	4027	2742	111	1174	0.04	79		
	mknod	4200	2775	214	1211	0.08	65		
	realpath	5927	2644	86	3197	0.03	72		
	dir	19124	12900	320	5904	0.02	231	×	×
angr	basename	3415	2217	18	1180	0.01	59		
	expand	4033	2742	111	1180	0.04	79		
	mknod	4206	2775	214	1217	0.08	65		
	realpath	5933	2644	86	3203	0.03	72		
	dir	19134	12751	413	5970	0.03	230		
BAP	basename	5894	826	114	4954	0.14	37	×	
	expand	7373	1320	205	5848	0.16	56	×	
	mknod	7022	1282	162	5578	0.13	43	×	
	realpath	11368	1251	108	10009	0.09	46	×	
	dir	28906	5718	667	22521	0.12	150	×	×
Hopper	basename	3250	2217	18	1015	0.01	59		
	expand	3845	2742	111	992	0.04	79		
	mknod	4022	2775	68	1179	0.02	65		
	realpath	5636	2644	86	2906	0.03	72		
	dir	18292	12607	350	5335	0.03	230	×	×
IDA Pro	basename	3221	2217	18	986	0.01	59		
	expand	3820	2742	111	967	0.04	79		
	mknod	3995	2775	68	1152	0.02	65		
	realpath	5607	2644	87	2876	0.03	72		
	dir	18220	12751	268	5201	0.02	230		
Ghidra	basename	3256	2217	18	1021	0.01	59		
	expand	3826	2742	99	985	0.04	79		
	mknod	4029	2775	68	1186	0.02	65		
	realpath	5658	2644	86	2928	0.03	72		
	dir	18303	12751	267	5285	0.02	230		×
Dyninst	basename	3269	2222	16	1031	0.01	60		×
	expand	3874	2707	123	1044	0.05	79		×
	mknod	4058	2747	214	1097	0.08	64		×
	realpath	5724	2609	85	3030	0.03	71		×
	dir	18694	12845	329	5520	0.03	230		×