On The Decidability Of Disassembling Binaries*

Daniel Engel [1], Freek Verbeek [1,2], and Binoy Ravindran [2]

[1] Open University, 6419 AT Heerlen, The Netherlands
{daniel.engel,freek.verbeek}@ou.nl
[2] Virginia Tech, VA 24061 Blacksburg, United States
binoy@vt.edu

Abstract. The general consensus is that disassembly of binaries is undecidable. The cause lies in distinguishing instructions from data, and resolving indirections. Furthermore, binaries can behave in "weird" ways which have no counterpart in assembly languages, e.g., instructions may overlap, or use other instructions as data. Yet, the general consensus is that, for a large part of production binaries, disassembly works sufficiently well for the use cases at hand. This paper aims to address the question: for which binaries is disassembly decidable? For which binaries can disassembly become decidable if an external oracle, e.g., provides the set of instruction addresses, or resolves indirections? We present a set of five theorems on decidability of disassembly; each theorem corresponding to a use case. All five theorems are accompanied by a proof of correctness based on bisimilarity between the input binary and the output assembly program, and have been formalized in the Isabelle/HOL theorem prover.

Keywords: Disassembly · Formal Methods · Decidability

1 Introduction

Disassembling is the process of retrieving an *assembly program* for a given *binary*. It is a well-known fact in the decompilation community that it is not possible to create an algorithm which, for every binary, performs disassembling in such a way that the semantics of the assembly program is equivalent to the semantics of the original binary [7, 15, 18, 23, 28, 31]. The bytes in a binary do not indicate whether they encode instructions (i.e., the program text) or raw data. In order to decide this, one must know which bytes are reachable as the start of an instruction and which are unreachable by any execution of the program, which is an *undecidable* problem [24].

Matters are complicated by the ability of binaries to behave "weirdly" [6]. They may read their program text as raw data, modify its instructions to dynamically change the control flow, execute raw data as instructions or jump into

the middle of an instruction. Most assembly languages do not allow such behavior: there is a clear distinction between the program text and data, instructions may not overlap and self-modification is usually not allowed. This may even make it impossible to disassemble weird binaries to well-formed assembly code to begin with.

This paper addresses the question *in which scenarios is disassembling decidable?* A scenario is characterized by *assumptions* about the input binary, *oracles* that provide information over the binary and *requirements* for the output assembly program. The assumptions may allow or exclude binaries that modify themselves, use text as data and vice-versa, jump into the middle of instructions, etc. These assumptions might be statically or dynamically verified, or be enforced at compile time. The oracles may solve hard problems such as reachability analysis or indirection resolution. They too could be created by a static or dynamic analysis tool, or be created as an additional compilation artifact. The requirements enable different *use cases* in which disassembling is needed. For example, in order to symbolically execute an assembly program, one needs a Control Flow Graph (CFG), while recompiling into a binary requires a byte-consistent assembly output.

In this work, we identify five scenarios for which we prove sound disassembling to be decidable: 1. *Debugging*, for which the distinction between text and data is unimportant, 2. *Verification*, for which an overapproximation of all reachable addresses is needed, 3. *Traversing*, for which an overapproximative CFG is needed, 4. *In Situ Patching*, for which the encoding of instructions may not influence the behavior of other instructions, and 5. *Decompilation* for which entire program sections need to be relocatable.

The contribution of this paper is a set of five theorems on decidability of disassembly; one for each scenario. The theorems have been formally proven correct in the Isabelle/HOL theorem prover [20]. The motivation behind these theorems is twofold:

- Binaries are known to be opaque. Current research focuses on extending the binary format with metadata that makes the binary analysis more tractable. The Debugging With Arbitrary Record Formats (DWARF) format[3] provides debugging information, but is, in itself, not enough to recover, e.g., a CFG. This paper aims to provide a formal answer to the question: what meta data (i.e., oracles) should an extended binary format at least contain to make it amenable for the aforementioned use cases?
- We argue that our effort effectively formalizes several properties that capture the intuition of "normal" behavior for binaries (in contrast to their possible "weird" behaviors). Lifting a binary to a semantically equivalent Intermediate Representation (IR) requires that the binary behavior is expressible in that IR. Self-modification, as a single example, generally is not. We thus formalize negations of "weird" behaviors such as the aforementioned ones, and prove that these properties allow lifting to IRs.

[3] `https://dwarfstd.org`

In the following section, we present an overview of these five scenarios together with the proofs that their respective disassembling is decidable. We also contribute a meta theorem that allows combination of different scenarios. Section 2 discusses the related work on decidability of disassembly. The overview of Section 3 is presented in more mathematical rigor in Section 4. In Section 5, we discuss the implications of the assumptions and oracles over the binaries, and the limitations of our model of assembly programs. Section 6 provides a conclusion and possible future work.

2 Related Work

The general consensus in related work is that disassembly is undecidable [1, 7, 9, 10, 15, 27, 31]. Cifuentes phrases the intuition as follows: «Given a binary program, the separation of data from code, even in programs that do not allow such practices as self-modifying code, is equivalent to the halting problem, since it is unknown in general whether a particular instruction will be executed or not (e.g. consider the code following a loop).» [9]. This observation has triggered research into incorporating heuristics into disassembly algorithms [21], or in applying machine learning to disassemble binaries [14, 22, 32]. These techniques aim to be precise but acknowledge they can never be provably sound. Wartell et al. explicitly state that «fully correct x86 disassembly is provably undecidable: Bytes are code if and only if they are reachable at runtime - a decision that reduces to the halting problem.» [32].

Yet, one can take a binary, overapproximate the set of reachable instruction addresses by taking the entire address space of the binary, and run a per-instruction disassembler on each address. Assuming trust in a per-single-instruction disassembler, the result is sound and recompilable to a byte-equivalent binary [26], and the process is clearly decidable. Thus: disassembly *is* decidable. This observation has triggered research into disrupting a static disassembler through obfuscation [11, 17] to provide protection against intellectual property theft in case of closed-source software.

The reconciliation between these two contradictory statements is that both are true, depending on the definition of "disassembly". If "disassembly" is defined as lifting a binary to a representation with solely the requirement that the IR is executable, then the above argument for decidability holds. This is exactly what we formalize as Theorem 1 in Figure 1 (which will be discussed in the next section). However, if "disassembly" is defined as lifting to an IR with more requirements (which is often implicitly done in literature), it becomes undecidable and oracles are needed. For example, disassembly to a representation that can be recompiled without fixing the location of the text- and datasections in memory, is undecidable as it requires resolving indirections (this is the argument by Cifuentes quoted above). This is exactly the statement made in Theorem 5 in Figure 1.

To the best of our knowledge, there is no existing work that aims to formalize how decidability of disassembly depends on the definition of "disassembly", on its requirements and use cases, and on the assumptions made over the binaries.

3 Overview

This section provides a semi-formal overview over the theorems proven in this paper. We formally define the meaning of a *decidability theorem* for disassembling, and informally present and discuss different properties for both binary and assembly programs, which are used as the assumptions and requirements. This combines into a set of decidability theorems as shown in Figure 1.

We model both binaries and assembly programs to abstract over all kinds of real-world architectures. Binaries are partial maps from addresses to bytes. Assembly programs consist of two partial maps, one from addresses to bytes (the *program data*) and one from addresses to instructions (the *program text*). Both mark sections of the program to be executable, readable or writable.

The semantics for both are given as unlabeled transitions systems. Programs are executed in an environment Γ which contains the offset at which the program is loaded into memory, and the addresses and semantics of external symbols. The semantics for a binary β in environment Γ are denoted $TS(\Gamma;\beta)$, and are defined as straightforwardly as possible. The initial states have their instruction pointer (`RIP`) set to the entry point of the binary β and all of its bytes are loaded into memory. Transitions are given by fetching the bytes at the current `RIP`, decoding them as an instruction and executing it. The semantics for an assembly program, denoted by $TS(\Gamma;\alpha)$, are defined similarly, with the difference that the instructions are fetched directly from α's program text, not from memory. Note that the binary semantics allow self-modification while the assembly semantics do not.

3.1 Theorems

A *disassembler* is a computable function that takes as input an binary and a (possibly empty) set of oracles, and produces an assembly program as output. *Assumptions* are predicates over binary programs, *requirements* are predicates over assembly programs. *Oracles* can be used by the disassembler function to retrieve information about the binary program. We use the standard notion of *bisimilarity* [3] (notation $\simeq$) as the correctness relation between the assembly and binary semantics.

Definition 1 (Sound Disassembler). *Given assumptions $\mathcal{A}$, requirements $\mathcal{R}$ and oracles $\mathcal{O}$, a disassembler d is* sound *iff, for every binary β for which the assumptions hold, the resulting assembly program α fulfills the requirements and the semantics are bisimilar.*

$$sound(d,\mathcal{A},\mathcal{O},\mathcal{R}) \coloneqq \forall \beta,\Gamma\cdot \mathcal{A}(\beta) \Longrightarrow (\mathcal{R}(\alpha) \wedge TS(\beta;\Gamma) \simeq TS(\alpha;\Gamma))$$
$$\textbf{\textit{where}}\ \alpha = d(\beta,\mathcal{O})$$

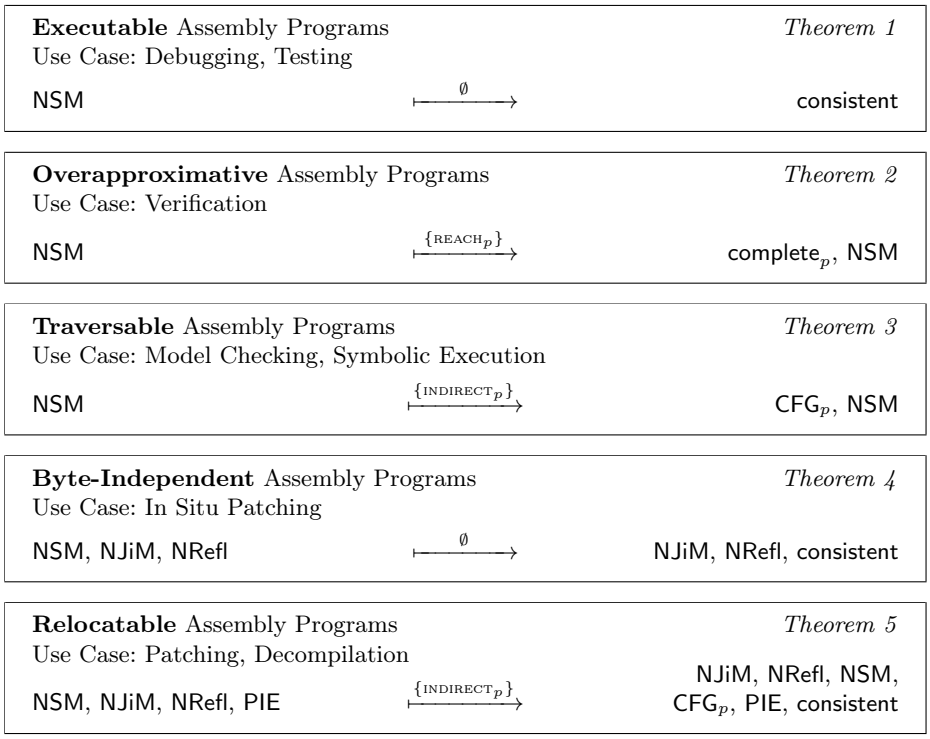

Fig. 1. Overview of the five decidability theorems. The set of properties on the left are the assumptions made over the input binary, the set of properties on the right are the requirements over the output assembly program guaranteed by the disassembler, the set of oracles on the arrow are the oracles needed for the scenario.

Definition 2 (Decidability Theorem). *Disassembling from a set of binaries fulfilling assumptions $\mathcal{A}$ to a set of assembly programs fulfilling requirements $\mathcal{R}$ using oracles $\mathcal{O}$ is* decidable *iff a sound disassembler exists. We use the notation $\mathcal{A} \overset{\mathcal{O}}{\mapsto} \mathcal{R}$ to denote this decidability.*

$$\mathcal{A} \overset{\mathcal{O}}{\mapsto} \mathcal{R} \coloneqq \exists d \cdot sound(d, \mathcal{A}, \mathcal{O}, \mathcal{R})$$

Executable Assembly Programs can be used to *interactively debug* the binary. In a debugging scenario, the user is only interested in the instructions that are executed and not in semantic properties such as the distinction between instructions and data, the overlapping of instructions with each other and with data, or the entire CFG of the program. For this, the assembly program is only required to be *byte-consistent*, meaning that any overlapping instructions or any instructions overlapping with data must agree on their byte encoding. The only assumption on the binary is that it is *non self-modifying (*NSM*)* since our assembly semantics do not capture self-modifying behavior. Byte-consistency does not

need to be assumed on binaries since all binaries are byte-consistent. Intuitively, the disassembler can decode the bytes at every address in the binary, regardless of whether or not they are reachable, as both text and as data, as this distinction is not important for debugging.

Overapproximative Assembly Programs can be used to verify safety properties such as absence of buffer overflows or probing functions return normally. In order to do so soundly, a verification tool needs a *complete* set of all instructions that may be reachable. A disassembler thus requires some oracle that provides a set containing at least all reachable addresses of the binary. The precision of the verification increases with a more precise (smaller) reachability oracle. In order for the oracle to correctly model all reachable instructions, the assembly program and thus the binary are assumed to be NSM.

Traversable Assembly Programs extend overapproximative assembly programs by enabling static traversal of the program in order to perform model checking techniques [5, 16], abstract interpretation [4, 19] or symbolic execution [12, 25]. In order to traverse the program, it is not enough to know which instructions are reachable, but rather which instructions can be directly reached from which other instructions. In the realm of low-level programs, the construction of CFGs is even more difficult than for high-level programs since indirect jumps are ubiquitous. Examples for indirect jumps include jump instructions to dynamically computed addresses such as `jmp RAX`, calls to callbacks such as `call RDI` or even simple return statements `ret`, as they need to load the return address from the stack to indirectly jump to it. A disassembler creating traversable programs may use an *indirection oracle* to build the CFG.

Byte-Independent Assembly Programs have semantics that do not depend on the encoding of instructions. This allows for *in situ patching* where individual instructions are changed without affecting the semantics of unchanged instructions [13]. The three properties that guarantee that the encoding does not influence the behavior are: 1. NSM so that the program cannot change individual bytes in an instruction, 2. *non reflexive (*NRefl*)* so that the program cannot read the encoding of instructions as raw bytes, and 3. *no jumps in the middle (*NJiM*)* so that the partial encoding of instructions cannot be interpreted as new instructions. In addition to these properties, the assembly program needs to be able to be assembled into a binary again, requiring it to be byte-consistent.

Relocatable Assembly Programs extend the use case of in situ patching by making entire program sections relocatable. On the one hand, this is needed for *full scale patching* for which new instructions may be introduced, requiring later instructions to be moved to other addresses. On the other hand, this is needed by further *decompilation* steps such as symbolization, for which sections need to be moved from constant offsets to arbitrary, more abstract program points. In

addition to the byte-independence properties consistent, NSM, NRefl and NJiM, relocatable programs also need to be position independent executables (PIEs). Position independence means that the offset at which the program is loaded into memory does not affect the program behavior, thus parts of the program can be relocated to arbitrary offsets.

The properties used in Figure 1 are *semantic* properties, i.e., they are formulated as properties over the transition systems provided by $TS(\Gamma;\beta)$ and $TS(\Gamma;\alpha)$. As example, NSM is defined as "if an address is a reachable instruction address, then there exists no execution that writes to that address". Such a property can be hard to verify. However, several properties have a *syntactic* counterpart that enforces the semantical property by construction. For example, NSM can be enforced if the executable address space does not overlap with the writable address space (denoted with $X \overline{\wedge} W$). For binaries that enforce the desired semantical properties syntactically, there is no need to verify whether the binary satisfies the assumptions. The syntactic version implies its semantic counterpart.

Table 1 presents an overview of the properties used in Figure 1, all of which will be formally defined in Section 4.

Semantic	**Syntactic**	**Description**	**Definitions**
	consistent†	The overlapping bytes of different instructions and data match	Definition 5
NSM‡	$X \overline{\wedge} W$	The program does not change its text at runtime	Definitions 10,6
NRefl	$X \overline{\wedge} R$	The program does not read its instructions as data	Definitions 11,7
NJiM	mosaic	The control flow does not lead into the middle of an instruction	Definitions 12,8
PIE		The semantics do not depend on the offset where the program is loaded into memory	Definition 13
complete$_p$		All reachable addresses are known, there are at most p addresses	Definition 14
CFG$_p$		The complete control flow is known, there are at most p edges	Definition 15

†: Always true for binaries; ‡: Always true for assembly programs.

Table 1. Overview of the properties used as assumptions and requirements. The first two columns are the semantic and syntactic version of the property, most syntactic properties are only defined for assembly programs.

3.2 Meta Theorem

The theorems presented in Section 3.1 represent individual scenarios in which disassembling a binary can be used to enable certain use cases while guaranteeing preservation of the semantics. In addition to these five theorems, we show that the requirements and assumptions used in these theorems can be combined to create new scenarios.

In order to show that the combination of the five theorems is possible, we first need to define the *merge* function for disassemblers. Merging is done by applying both disassemblers to the input binary and oracles, and keeping the parts both resulting assembly program agree on. In particular, for the text and data, at every address, the instruction or byte both assembly programs have at that address is kept in the result. The different program sections are joined, meaning each address of the merged assembly program is only marked as executable/writable/readable iff it is in both original assembly programs. The entry address can be arbitrarily chosen from the original assembly programs. If the disassemblers are sound then the entry addresses of both original assembly programs must be equal.

Definition 3 (Merging of disassemblers). *The* merge *function is a higher order function taking two disassemblers d_1, d_2 as an input and returning a disassembler that combines the outputs of d_1 and d_2.*

$$
\begin{aligned}
\mathit{merge}(d_1, d_2)(\beta, \mathcal{O}) \coloneqq & \langle \mathit{text}', \mathit{data}', X_1 \cap X_2, W_1 \cap W_2, R_1 \cap R_2, E_1 \rangle \\
& \textbf{\textit{where}}\ \langle \mathit{text}_1, \mathit{data}_1, X_1, W_1, R_1, E_1 \rangle \coloneqq d_1(\beta, \mathcal{O}) \\
& \quad \langle \mathit{text}_2, \mathit{data}_2, X_2, W_2, R_2, E_2 \rangle \coloneqq d_2(\beta, \mathcal{O}) \\
& \quad \mathit{text}'(a) \coloneqq \begin{cases} i & \text{if } \mathit{text}_1(a) = \mathit{text}_2(a) = i \\ \emptyset & \text{otherwise} \end{cases} \\
& \quad \mathit{data}'(a) \coloneqq \begin{cases} b & \text{if } \mathit{data}_1(a) = \mathit{data}_2(a) = b \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Theorem M (Combination of Scenarios). *For any two sound disassemblers d_1 and d_2 presented in Figure 1, the merged disassembler is sound:*

$$
\begin{aligned}
& \mathit{sound}(d_1, \mathcal{A}_1, \mathcal{O}_1, \mathcal{R}_1) \wedge \mathit{sound}(d_2, \mathcal{A}_2, \mathcal{O}_2, \mathcal{R}_2) \\
& \qquad \Longrightarrow \mathit{sound}(\mathit{merge}(d_1, d_2), \mathcal{A}_1 \wedge \mathcal{A}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \wedge \mathcal{R}_2)
\end{aligned}
$$

In particular, this means:

$$
\mathcal{A}_1 \overset{\mathcal{O}_1}{\longmapsto} \mathcal{R}_1 \wedge \mathcal{A}_2 \overset{\mathcal{O}_2}{\longmapsto} \mathcal{R}_2 \Longrightarrow (\mathcal{A}_1 \wedge \mathcal{A}_2) \overset{(\mathcal{O}_1 \cup \mathcal{O}_2)}{\longmapsto} (\mathcal{R}_1 \wedge \mathcal{R}_2)
$$

Proof. Given the input binaries β_1, β_2, the assumptions, requirements and oracles $\mathcal{A}_1, \mathcal{A}_2, \mathcal{R}_1, \mathcal{R}_2, \mathcal{O}_1, \mathcal{O}_2$, and the sound disassemblers d_1, d_2, let the resulting programs $\alpha_1 = d_1(\beta_1, \mathcal{O}_1), \alpha_2 = d_2(\beta_2, \mathcal{O}_2)$ and $\alpha' = \mathit{merge}(d_1, d_2)(\beta, \mathcal{O}_1 \cup \mathcal{O}_2)$.

By soundness of the the disassemblers, we have for all Γ, $TS(\alpha_1;\Gamma) \simeq TS(\alpha_2;\Gamma)$. This is implies that α_1 and α_2 must agree on all reachable instructions and on all bytes that are loaded or stored. Since merging keeps all instructions and data both programs agree upon and since they are bisimilar, the resulting program α' must also be bisimilar to α_1 and α_2.

We need to show that that all properties are preserved in α' as long as they are fulfilled by either α_1 or α_2. consistent *is preserved as α' only contains the bytes α_1 and α_2 agree upon. The semantic properties* NSM, NRefl *and* NJiM *are preserved by bisimilarity of the transition systems $TS(\alpha_1;\Gamma)$, $TS(\alpha_2;\Gamma)$ and $TS(\alpha';\Gamma)$. Merging does not introduce any new instructions, as only the ones α_1 and α_2 agree on are kept, so no new instruction can read, write or overlap with other instructions. Preservation of* PIE *follows directly from bisimilarity.*

From $\mathsf{complete}_p(\alpha_1)$ *and* $\mathsf{complete}_q(\alpha_2)$*, we have* $\mathsf{complete}_r(\alpha')$*, for some r where $r \leq min(p,q)$. The set of reachable addresses in α' must have at most as many bytes as the smaller one from α_1 and α_2 since only the bytes both agree on are kept. This set contains at least all reachable addresses as α' is bisimilar to α_1 and α_2. The same reasoning holds for* CFG_p*.*

4 Formalization

This section provides more mathematical rigor for the definitions used in Section 3. Our model of the low-level languages runs on an abstract machine with an *infinite address space*. We use α for assembly programs, β for binaries, π for programs that can be either of the two, ω for oracles, ψ for abstract machine states and Γ for execution contexts. We use $\psi[a]$ for a memory read or write in state ψ at address a.

We first present the syntax and syntactic properties of assembly programs and binaries. Both assembly programs and binaries contain partial mappings from addresses to some form of data. Binaries only contain raw bytes as they serve the dual purpose as instructions (summarized in Table 4) to be decoded, and as data. Assembly programs make a distinction between these two forms of information. Additionally, both assembly and binaries contain indications for eXecutable, Readable and Writable sections and an Entry address.

Definition 4 (Programs). *Assembly programs and binaries are tuples containing the program text and data, or the raw bytes representing them, a set for each the executable, readable and writable addresses and an entry address.*

$$
\begin{array}{l|l}
Asm \coloneqq \langle\ text : Addr \rightharpoonup Instr, & Bin \coloneqq \langle \\
\qquad data : Addr \rightharpoonup Byte, & \qquad bytes : Addr \rightharpoonup Byte, \\
\qquad X : \{Addr\},\ W : \{Addr\}, & \qquad X : \{Addr\},\ W : \{Addr\}, \\
\qquad R : \{Addr\},\ E : Addr\ \rangle & \qquad R : \{Addr\},\ E : Addr\ \rangle
\end{array}
$$

Both types of programs are subject to *well-formedness* conditions to ensure their semantics are well-defined. For all assembly programs α and binaries β, we have:

$reachable(\alpha) \subseteq dom(text(\alpha)) \wedge$ $dom(data(\alpha) \subseteq W(\alpha) \cup R(\alpha)) \wedge$ $dom(text(\alpha) \subseteq X(\alpha))$	$reachable(\beta) \subseteq dom(fetch(\beta)) \wedge$ $dom(bytes(\beta)) \subseteq (W \cup R \cup X)(\beta))$

The syntactic properties over both types of programs can be easily checked, and overapproximate the semantic properties used in Figure 1. *Byte-consistency* describes that the bytes of overlapping instructions and data in assembly programs must agree with another, ensuring the assembly program can be assembled into a binary. For binaries, there is no distinction between text and data and there can only ever be at most one byte at an address. Thus, binaries always fulfill this property. A *mosaic* assembly program does not have any overlapping instructions, as such, it does not allow for any jumps into the middle of instructions. The properties $X\overline{\wedge}W$ and $X\overline{\wedge}R$ ensure that the text-sections of a program do not overlap with the data sections. In turn, programs fulfilling these properties cannot, by construction, modify themselves, or read the encoding of their own instructions.

Definition 5 (Byte-Consistency). *An assembly program α is* byte-consistent *iff, for every address a, the bytes of overlapping instructions agree with another and with the data bytes at a. Let $text{-}bytes$ be the function that returns the set of bytes corresponding to the decoding of all instructions that are at a given address.*

$$\begin{aligned} byte{-}consistent(\alpha) \coloneqq\ & (\forall a \cdot data(a) = b \implies b \in text{-}bytes(\alpha, a)) \\ & \wedge\ (\forall a \cdot |text{-}bytes(\alpha, a)| \leq 1) \end{aligned}$$

Definition 6 (No Execute Where Write). *The property $X\overline{\wedge}W$ describes that no address is both in an executable and a writable section.*

$$X\overline{\wedge}W(\pi) \coloneqq X(\pi) \cap W(\pi) = \emptyset$$

Definition 7 (No Execute Where Read). *The property $X\overline{\wedge}R$ describes that no address is both in an executable and a readable section.*

$$X\overline{\wedge}R(\pi) \coloneqq X(\pi) \cap R(\pi) = \emptyset$$

Definition 8 (Mosaic). *An assembly program α is* mosaic *iff no two instruction in the program text overlap.*

$$\mathsf{mosaic}(\alpha) \coloneqq \forall(a_1, a_2 \in dom(text(\alpha))) \cdot \neg(a_1 \leq a_2 < a_1 + |text(\alpha, a_1)|)$$

Programs have no a priori knowledge of where they are loaded into memory or the behavior of dynamically linked (external) symbols. This information is captured in the *execution context* Γ.

The semantics for both binary and assembly programs are given as unlabeled transition systems. We write $TS(\alpha; \Gamma)$ and $TS(\beta; \Gamma)$ for the transition system

Opcode	Arguments	Behavior
`halt`	$\emptyset$	Stops program execution.
`const`	$(\rho : Reg)(v : \mathbb{Z})$	Writes the constant v into register ρ.
`load`	$(\rho_d, \rho_s : Reg)$	Reads address a from register ρ_s, reads value v from memory at a, writes v into register ρ_d.
`store`	$(\rho_d, \rho_s : Reg)$	Reads address a from register ρ_d, reads value v from register ρ_s, write v into memory at a.
`cond`	$(\rho_c, \rho_d, \rho_s : Reg)$	Reads value c from register ρ_c, if $c \neq 0$: Reads value v from register ρ_s, writes v into register ρ_d.
`size`	$(\rho_d, \rho_s : Reg)$	Reads value v from register ρ_s, writes the number of bytes needed to represent v into register ρ_d.
`symb`	$(\rho : Reg)(s : Sym)$	Queries the execution context for the address a of the symbol s, writes a into register ρ_d.
`apply`	$(f)(\rho_d, \rho_1, \rho_2 : Reg)$	Reads values v_1, v_2 from registers ρ_1, ρ_2, writes $f(v_1, v_2)$ into register ρ_d.

Table 2. Generic instruction set used for the binary and assembly languages. Real-world instructions can be built from one or multiple instructions above. For example, `add RAX, 42` can be modeled by `const` ρ_{tmp}, 42 `; apply` (+), `RAX`, `RAX`, ρ_{tmp}.

describing the execution of the assembly program α or the binary β. The initial states have all bytes of the program loaded into memory at the loading offset of Γ. Transitions between states are given 1. for binaries by fetching bytes at `RIP` from memory and decoding them as instructions, 2. for assembly programs by fetching the instruction at `RIP` from the program text, and then executing them according to Table 4. Addresses within the domain of the *bytes* for a binary or *data* for an assembly program can only be read if they are elements of the R set, and only be written if they are elements of the W set. Similarly, addresses can only be fetched as instructions if they are within in X set.

Definition 9 (Execution Context). *The* execution context *is a tuple containing the load address, the location and the semantics of external symbols.*

$$Ctx \coloneqq \langle\ load{-}addr : Addr,\ \ externs : Sym \to Addr,\\ extern{-}sem : Addr \rightharpoonup State \times State\ \rangle$$

The semantic properties define constraints over states reachable from the entry point of a program. All of them use the notations $reachable_n(ts, \psi)$ and $reachable(ts, \psi)$ indicating that state ψ is reachable within n steps or any number of steps in the transition system ts. A binary is *non self-modifying* if no memory write occurs within the boundaries of its executable sections. For assembly programs, the instructions are not fetched from memory, thus they do not have a notion of self-modification. Similarly, a program is *non reflexive* if no memory read occurs within its executable sections. The property *no jump-in-the-middle* captures that no two states are reachable for which the current instructions overlap with another. A program is *position independent* if its semantics do not depend on the execution context's loading offset.

Definition 10 (No Self-Modification). *A binary is* non self-modifying (NSM) *iff no memory writes occurs within the executable sections.*

$$\begin{aligned}\mathsf{NSM}(\beta) \coloneqq \forall \Gamma, \alpha, \psi_1, \psi_2, n \cdot \\ & \mathit{reachable}_n(\mathit{TS}(\beta;\Gamma), \psi_1) \implies \\ & \mathit{reachable}_{(n+1)}(\mathit{TS}(\beta;\Gamma), \psi_2) \implies \\ & \psi_1[a] \neq \psi_2[a] \implies a \notin \mathit{dom}(X(\beta))\end{aligned}$$

Definition 11 (No Reflexivity). *A binary or assembly program is* non reflexive (NRefl) *iff the encoding of any instruction other than the current one does not influence the behavior of the current instruction.*

$$\begin{aligned}\mathsf{NRefl}(\pi) \coloneqq \forall \Gamma, \pi', \psi_1, \psi_2, \psi_2' \cdot \\ & \mathit{reachable}_n(\mathit{TS}(\pi;\Gamma), \psi_1) \wedge \mathit{reachable}_n(\mathit{TS}(\pi';\Gamma), \psi_1) \implies \\ & \mathit{reachable}_{(n+1)}(\mathit{TS}(\pi;\Gamma), \psi_2) \wedge \mathit{reachable}_{(n+1)}(\mathit{TS}(\pi';\Gamma), \psi_2') \implies \\ & \psi_2 = \psi_2'\end{aligned}$$

Definition 12 (No Jump-In-The-Middle). *A binary or assembly program fulfills the* no jumps in the middle (NJiM) *property iff there are no two instructions reachable which overlap with another.*

$$\begin{aligned}\mathsf{NJiM}(\pi) \coloneqq \forall \Gamma, (\psi_1, \psi_2 \in \mathit{reachable}(\mathit{TS}(\pi;\Gamma)) \cdot \\ & \neg(\mathtt{RIP}(\psi_1) \leq \mathtt{RIP}(\psi_2) < \mathtt{RIP}(\psi_1) + |\mathit{fetch}(\pi, \psi_1, \mathtt{RIP}(\psi_1))|)\end{aligned}$$

Definition 13 (Position Independent Executable). *A binary or assembly program is a* position independent executable (PIE) *iff the offset at which it is loaded into memory does not influence the semantics of the program.*

$$\mathsf{PIE}(\pi) \coloneqq \forall \Gamma, a_1, a_2 \cdot \ \mathit{TS}(\pi; \Gamma(\mathit{load{-}addr} \coloneqq a_1)) = \mathit{TS}(\pi; \Gamma(\mathit{load{-}addr} \coloneqq a_2))$$

Finally, a binary or assembly program is $\mathsf{complete}_p$ if all reachable reachable addresses are known in a set of cardinality p. The set containing these addresses is the *reachability oracle* $\textsc{reach}_p$. Here, p serves as a *precision metric*, the smaller p is, the greater the precision of the oracle. Without the precision metric, generating the oracle would become trivial as one can overapproximate the set of reachable addresses by all addresses. Similarly, the program has a known CFG_p if all pairs of successively reachable addresses are known in a set of cardinality p, the corresponding oracle is the *indirection oracle* $\textsc{indirect}_p$. Again, p serves as the precision metric counting the edges in the CFG. One could trivially overapproximate this oracle by considering all addresses reachable from all other addresses.

Definition 14 (Complete). *An assembly program or binary π is* $\mathsf{complete}_p$ *iff there exists a set containing at least all reachable addresses.*

$$\mathsf{complete}_p(\pi) \coloneqq \exists \omega \cdot |\omega| \leq p \wedge \forall \Gamma \cdot \mathit{reachable}(\mathit{TS}(\pi;\Gamma)) \subseteq \omega$$

Definition 15 (Known Control Flow). *A binary or assembly program π has a known control flow (CFG_p) iff for all reachable addresses the immediately next reachable addresses are known and the resulting CFG contains p edges.*

$$\begin{aligned}\mathsf{CFG}_p(\pi) \coloneqq \exists \omega \cdot |\omega| \leq p \wedge \forall \Gamma, \psi_1, \psi_2, n \cdot \\ \mathit{reachable}_n(\mathit{TS}(\pi;\Gamma), \psi_1) \implies \\ \mathit{reachable}_{(n+1)}(\mathit{TS}(\pi;\Gamma), \psi_2) \implies \\ \langle \psi_1, \psi_2 \rangle \in \omega\end{aligned}$$

5 Discussion

The formalization of binary and assembly languages presented in Sections 3 and 4 make certain assumptions over the semantics of assembly languages, the availability of oracles for the translations, and properties of the input binaries.

5.1 Self-Modification

Our model of assembly languages contains *static* program text. Instructions are fetched directly from an unchangeable assembly program instead of being *dynamically* loaded from memory, which stands in contrast to what binaries do. There do exist formalizations of self-modifying assembly programs such as [2,8,29], but for the purposes of this paper, non-self-modifying assembly programs are better suited.

Static Program Text is what one usually wants from a disassembler, as it allows the program text to be analyzed easier. The disadvantage is that the runtime semantics between a binary, which always has a dynamic program text, and the assembly program can differ, even when they *initially* contain the same instructions. Figure 2 shows an example program where a memory write alters the dynamic text of the binary by changing the last instruction while the static text of the assembly program remains unaltered.

```
self_modification_bin:
  ; Overwrites the next instruction
  mov BYTE [RIP], 0xC3

  ; Changes to a ret (0xC3)
  nop
```

```
self_modification_asm:
  ; Does not influence the text
  mov BYTE [RIP], 0xC3
  ; Remains a nop in the text
  ; Changes to a 0xC3 in memory
  nop
```

Fig. 2. Difference in semantics between a binary (left) and an assembly program (right). Both write `0xC3` into memory where the next address will be. For the binary, this changes the program text since instructions are fetched from memory. For the assembly program, the memory does not influence the program text.

Dynamic Program Text models the semantics of both the binary and the assembly program in the same way. Programs such as the one in Figure 2 behave the same for binaries and assembly programs with a dynamic program text. The disadvantage of a dynamic assembly program text is that the lines between text and data are necessarily blurred. Figure 3 shows two example assembly programs with different instructions but the same binary encoding. Under dynamic semantics, where instructions are loaded from memory, these two programs behave the same. For this work, this would make disassembling always trivial as one would never need to resolve the starting addresses of any instructions, as any binary with the correct encoding would be considered bisimilar.

```
push RBP       ; 55
mov  RBP, RSP  ; 48 89 E5
call printf    ; E8 00 00 00 00
add  RAX, 5    ; 48 83 C0 05
```

```
.bytes 0x894855 ; 55 48 89
in  EAX, 0xe8   ; E5 E8
add [RAX], AL   ; 00 00
add [RAX], AL   ; 00 00
add RAX,0x5     ; 48 83 C0 05
```

Fig. 3. Self-Modifying semantics for assembly programs allow even incorrect disassemblers to be considered correct. The original program (left) and the incorrectly disassembled (right) program will have the same behavior under self-modifying semantics. Both contain exactly the same bytes, so fetching instructions from memory will result in the same execution traces. At the same time, both program have instructions with completely different semantics.

5.2 The Oracles

Figure 1 makes use of an oracle to overapproximate all reachable addresses (REACH_p) and an oracle to overapproximate the CFG (INDIRECT_p). Neither of these is available in any current debugging format like DWARF. We discuss whether – and how – these oracles can be obtained either at compile-time (when source code is available), or from a stripped binary (when source is not available).

Both can be implemented trivially by overapproximation of the actual address space and CFG if one considers all addresses to be reachable from all other addresses. However, such an overapproximation would be practically unusable for any real use cases and generating them with a high precision proves to be a difficult problem.

From Source Code. At the source code level, one might expect the information to generate the oracles to be present. For example, the instruction selection phase of a compiler emits all intended instructions and the linker assigns them their final addresses. Based on this information, the compiler might create a reachability oracle by emitting all these addresses as an additional artifact.

However, this only captures the *intended* instruction addresses, reachable from well-behaved source-level code. At runtime, unexpected behaviors like stack

overflows may cause the program to set its `RIP` to an unintended address. In this case, the oracle is no longer a sound overapproximation of all reachable addresses.

For the indirection oracle, the intended addresses are not fully known, not even at compile time. Indirections are ubiquitous in binaries as they are used for the jump tables in control flow constructs like `switch` statements, implement the mechanism used in `return` statements, and model the invocation of callbacks. While jump tables might be fully known at compile time, it is not possible to know all possible targets of `return` statements as this would require a full call graph. Moreover, note that even at the source level unresolved indirections may exist due to, e.g., callbacks.

Still, these artifacts can prove valuable in security analyses and binary hardening, even if they only capture the intention of the compiler. If the set of intended addresses is known, one can analyze the program to see if any address outside this set is reachable, uncovering a vulnerability rooted in low-level behavior not present in the source code. Similarly, one can harden the binary by adding a check before indirect jumps to abort the program if any unintended address is reached.

From Binary Code. At the binary level, the information needed to create the oracles must be recovered directly from the binary itself. In practice, the two oracles conflate, i.e., in order to get the set of reachable instruction addresses one must recover an overapproximative CFG. Deriving an overapproximative CFG requires establishing binary-level invariants that statically bound the values that certain state parts may have at run-time, in order to resolve indirections. This is known hard problem [30].

Real-world disassemblers try to uncover this information by one of two approaches: 1. Linear Sweep, or 2. Recursive Traversal. Tools such as *objdump* start at an entry address and linearly sweep through the section they try to disassemble. The instruction at the current address is decoded and the address is incremented by the instruction's size. For simple binaries, this approach works well, but as soon as text and data are mixed in unexpected ways, a linear sweep will fail to distinguish them. More sophisticated tools such as the disassemblers used in Ghidra, IDA-Pro or Binary Ninja, aim to decode all reachable instructions by following the binary's control flow. In contrast to a linear sweep, this is able to distinguish text from data even when they are mixed but will still fail for indirect jumps for which all possible successor edges in the CFG need to be recognized.

Figure 4 shows an example program where both linear sweep and recursive traversal will fail to produce the full and correct assembly code corresponding to the binary. Both linear sweep and recursive traversal are able to decode the instructions before a jump occurs (address `4000`). The byte following that instruction (address `4002`) is intended to be a raw data byte, but if one linearly decodes all instruction in the snippet, it will be decoded as an incorrect instruction. If one instead follows the CFG to address `4003`, the instructions of the original program can be decoded. However, even recursively traversing the CFG

fails after the last instruction (address 4006) since it is impossible to know all possible values of RAX.

```
4000: jmp $+1
4002: .byte 00
4003: sub RAX, RBX
4006: jmp RAX
```

```
4000: EB FF
4002: 00
4003: 48 29 D8
4006: FF E0
```

```
; Linear sweep decodes the first
; instruction
4000: jmp $+1
; But it will continue to
; disassemble after it
; 4002: 00 48 29
; 4005: D8 FF E0
4002: add [RAX+0x29],CL
4005: fdivr st,st(7)
```

```
; CFG traversal can resolve all 3
; instructions correctly
4000: JMP jmp $+1
4002: .byte 00
4003: sub RAX, RBX
; But it will not be able to continue
; correctly after an indirect jump
; to an unknown location
4006: jmp RAX
```

Fig. 4. An example program which cannot be correctly disassembled without the usage of the indirection oracle. The top left shows the original assembly code, top right shows the assembled result. The bottom left shows the result of linearly sweeping through the program. The first instruction is decoded correctly, but continuing to disassemble directly after it results in incorrect instructions. The bottom right shows the result of traversing the CFG. This results in the correct instructions being decoded on the snipped, but without further information, the indirect jump at the end means that the traversal cannot continue.

6 Conclusion

Generally, it is not possible to disassemble every binary into an equivalent assembly program. We provide a formalization for general binary and assembly languages and show that in certain scenarios, they can be correctly translated to each other. The key in showing that these translations are possible is to lift restrictions on the assembly language which are useful for writing assembly code by hand but are not always needed when doing disassembling.

In order to perform the disassembling correctly, oracles to enumerate the address space or resolve indirections may be needed. Generating these oracles itself can be an undecidable problem. Future work may see these oracles be implemented at compile time by leveraging information that is already known.

In this paper, we only examine the first step of many in a decompilation chain: converting a raw binary to an assembly program in which instructions are referenced by addresses. Further steps of de- and recompilation require the assembly program to be in a *symbolized* format, meaning that instructions can be referenced by abstract program labels. This requires a thorough *pointer analysis*, which could be a third oracle. In contrast to the assembly language presented

in this work, one does not only need to distinguish between instructions and data, but also between addresses pointing to instructions and addresses pointing to data. Similar to the resolution of indirections, a sound and complete pointer analysis is not possible in the general case. But one may ask the question "In which scenarios can data- and text-pointers be told apart?". Future work may see an extension of the Isabelle code-base presented here with a third language: *symbolized assembly*; and theorems describing the conversion between assembly and symbolized assembly.

References

1. Xiaoxin An, Freek Verbeek, and Binoy Ravindran. DSV: Disassembly soundness validation without assuming a ground truth. In *NASA Formal Methods Symposium*, pages 636–655. Springer, 2022.
2. Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. volume 4437, pages 232–248, 07 2006.
3. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
4. Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International conference on compiler construction*, pages 250–254. Springer, 2005.
5. Gogul Balakrishnan, T Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, S Yong, C H Chen, and Tim Teitelbaum. Model checking x86 executables with Codesurfer/x86 and WPDS++. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, pages 158–163. Springer, 2005.
6. Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The {PageFault} weird machine: Lessons in instruction-less computation. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.
7. Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 745–756, New York, NY, USA, 2015. Association for Computing Machinery.
8. Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A computability perspective on self-modifying programs. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 231–239, 2009.
9. Cristina Cifuentes. *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994.
10. Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.
11. C.S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.
12. Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016.

13. Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 151–163, 2020.
14. Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.
15. Ulf Kargén, Ivar Härnqvist, Johannes Wilson, Gustav Eriksson, Evelina Holmgren, and Nahid Shahmehri. desync-cc: A research tool for automatically applying disassembly desynchronization during compilation. *Science of Computer Programming*, 228:102954, 2023.
16. Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment: Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005. Proceedings 2*, pages 174–187. Springer, 2005.
17. Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.
18. Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 475–487, New York, NY, USA, 2020. Association for Computing Machinery.
19. Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings 10*, pages 115–130. Springer, 2012.
20. Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
21. Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP)*, pages 833–851. IEEE, 2021.
22. Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770*, 2020.
23. Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Accurate disassembly of complex binaries without use of compiler metadata. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 1–18, New York, NY, USA, 2024. Association for Computing Machinery.
24. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
25. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236, 2009.
26. Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

27. Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54. IEEE, 2002.
28. Ali Aydın Selçuk, Fatih Orhan, and Berker Batur. Undecidable problems in malware analysis. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 494–497, 2017.
29. Tayssir Touili and Xin Ye. Ltl model checking of self modifying code. *Formal Methods in System Design*, 60(2):195–227, 2022.
30. Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 934–949. ACM, 2022.
31. Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao, editors, *Advances in Knowledge Discovery and Data Mining*, pages 273–285, Cham, 2014. Springer International Publishing.
32. Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.