ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

TRex: Practical Type Reconstruction for Binary Code

Jay Bosamiya[1*], Maverick Woo[2], and Bryan Parno[2]

[1]*Microsoft Research*
[2]*Carnegie Mellon University*

Abstract

A lack of high-quality source-level types plagues decompiled code despite decades of advancement in the science and art of decompilation. Accurate type information is crucial in understanding program behavior, and existing decompilers rely heavily on manual input from human experts to improve decompiled output. We propose TRex, a tool that performs automated deductive type reconstruction, using a new perspective that accounts for the inherent impossibility of recovering lost source types. Compared with Ghidra, a state-of-the-art decompiler used by practitioners, TRex shows a noticeable improvement in the quality of output types on 124 of 125 binaries. By shifting focus away from recovering lost source types and towards constructing accurate behavior-capturing types, TRex broadens the possibilities for simpler and more elegant decompilation tools, and ultimately reduces the manual effort needed to analyze and understand binary code.

1 Introduction

Reverse engineering of machine code is useful in many contexts: binary hardening [38], malware analysis [40], program comprehension [7], vulnerability discovery [20], etc. Many tools have been built to assist reverse engineers, including disassemblers, debuggers, and decompilers. Decompilers, such as Ghidra [26], Binary Ninja [36], and Hex-Rays [30], are popular amongst reverse engineers since they provide a high-level source-code-like view of low-level machine code.

Unfortunately, despite decades of advances in the science and art of decompilation, the quality of decompiled output leaves much to be desired. A particularly persistent, unsolved problem is providing high-quality source-level type information for decompiled code. These types benefit both human understanding and the decompiler itself in producing improved output [24]. However, decompilers often struggle to recover any meaningful types from binaries (§5.1). Instead, they heavily rely on human experts to manually provide better type information. This is perplexing given the many years of academic research devoted specifically to producing high-quality source-level types. We speculate that this gap between practice and academia may be attributable to factors including: the unavailability of academic tools, the use of inconsistent benchmarks, and the inherent complexities of decompilation not fully captured in research papers.[1]

To explore this gap in decompilation performance, we decided to develop a new tool to perform better automated deductive type inference. We focus on deductive inference, rather than on approaches based on machine learning [6,7,21], since the latter can be hard to depend on in scenarios that differ drastically from their training corpus (§2.1). As a step towards reversing the unfortunate tendency of closed-source tools and non-reproducible benchmarks, we open-source our tool and release reproducible scripts for the benchmarks.[2]

As a starting point, we analyzed the requirements for automated initial type inference from the perspective of practitioners in the field. Here, we discovered a crucial mismatch in expectations—while prior techniques were attempting to recover source-level types, reverse engineers "merely" desire output that captures the behavior of the program. Even worse, *type recovery is often impossible* due to uncircumventable reasons (§2.2). Rather than attempting the impossible, we believe that the goal must shift to the construction of accurate behavior-capturing types. We call this new perspective Type Reconstruction[3] (§2.3).

We adopt this new perspective in constructing TRex,[4] a tool for automating the inference of source-level types for binary code. Building from the perspective of Type Reconstruction has fundamental consequences for the design of type inference tools, and in TRex's case, results in a noticeable quality improvement in the C-like types it produces when

[*] Work done while at Carnegie Mellon University.

[1] Indeed for one paper [25], other employees at the same company attempting to reproduce the work stated that "[i]t is a powerful system but difficult to understand" and the "presentation is very dense and subtle" [12].

[2] `https://github.com/secure-foundations/{trex,trex-usenix25}` (archived at `https://doi.org/10.5281/zenodo.15611994`)

[3] Capitalized to distinguish from prior informal usage (§2)

[4] TRex: **T**ype **R**econstruction for **ex**ecutables

compared to popular state-of-the-art decompilers.

Focusing on Type Reconstruction aids TRex in various ways. For example, since C is the de facto output language for decompilers, many existing tools succumb to the temptation to use C-like types internally during analysis. However, Type Reconstruction tells us that C-like nominal types are ill-suited to the task, and instead we must use structural types (meaning that a type is defined by observable behaviors/features; not to be confused with `struct` types) for the analysis even if the output types are in C, thereby better capturing what reverse engineers expect to see from their tools (§3.1).

Thus, TRex internally uses and can produce types that are far more expressive than C; these types can be used for further downstream analyses. It also produces human-readable C-like types projected from the more precise internal machine-readable types through a separate analysis phase. Phase separation is not limited to just this distinction between machine-readable and human-readable types, but extends further, both in the internal design of our approach, and also how it integrates with over-arching binary analysis frameworks. TRex makes only a small number of assumptions on its input and output, making it easy to adapt to any binary analysis framework without tight coupling to any of them. For our own reverse engineering projects, and also to improve the broader open-source binary analysis ecosystem, we build in support for Ghidra, thereby supporting all architectures supported by it. That said, our Ghidra-specific code is small, consisting only of ∼1000 lines of primarily boilerplate code.

In the process of constructing TRex, we have encountered challenges and discovered insights about binary code type inference that, to the best of our knowledge, are either novel or are tacit "oral tradition" in the community and remain unpublished. These insights help simplify the design and implementation of the tool while still providing expressive, high quality output. For example, binary analyses must actively manage conservativeness—a fully conservative tool would be a surprisingly bad idea (§3.2). Additionally, in exploring the difficulties of key phases of Type Reconstruction, we have discovered an algorithmic hardness result for a phase we call *type rounding*, which we show to be NP-Hard (§3.3.5).

In §5, we evaluate TRex. Given our insight regarding the lack of a singular valid ground truth, we evaluate TRex both qualitatively and quantitatively to better understand its benefits in comparison to other tools available to practitioners. Our quantitative evaluation uses a new metric that attempts to capture the expectations of reverse engineers, while working around the inherent difficulty of objectively evaluating type reconstruction. TRex outperforms the open-source state-of-the-art on a collection of benchmark binaries picked by prior works (that we create reproducible variants of, to facilitate future comparisons), achieving an average score that is 24.83% higher than Ghidra, which itself achieves a 17.68% higher score than a trivial baseline.

In summary, we make the following contributions.

Type ...	Multi-Language	No GT Assump.	Technique
Inference		N/A	Deductive
Prediction	✓		Learning
Recovery	✓		Deductive
Reconstruction	✓	✓	Deductive

Table 1: Our Categorization of Automated Type Inference for Binary Code. A key metric is whether a technique assumes the existence of ground truth (GT). See §2.1 and §2.3.

1. We propose Type Reconstruction, a new perspective on automated type inference from binary code, which accounts for the impossibility of recovering lost source-level types.
2. We build a new open-source tool, TRex, which takes arbitrary disassembly, from any architecture liftable to TRex's intermediate representation, and produces C-like types (covering C primitives, structs, arrays, unions, and recursive types) for human analysts, and more-detailed machine-readable output for downstream analyses.
3. We discover and document useful insights, such as the (im)possibility of traditional ground truth for binary code type inference, algorithmic hardness results, and more.
4. We demonstrate an improvement on 124 of 125 benchmark binaries when compared to an existing state-of-the-art decompiler. We also propose a new metric that better captures output quality from a reverse engineer's perspective. Additionally, we demonstrate that TRex outperforms an existing state-of-the-art learning-based approach.

2 Automated Type Inference for Binary Code

Automated source-like type inference from binary code has been explored for decades. Caballero and Lin [5] provide a detailed survey of techniques leading up to 2016, categorizing approaches along multiple axes: static vs. dynamic, value vs. flow-based, primitive vs. nested types, etc. In this section, we propose an alternate categorization, which we summarize in Table 1. We explore prior approaches and their core underlying flawed assumption of ground truth, finally leading us to our proposed new perspective—Type Reconstruction.

Note: Prior work's informal usage of the terms in Table 1 causes unintentional merging of distinct concepts. For instance, the abstract of one paper [25] states that "[t]he problem of recovering high-level types by performing type inference over stripped machine code is called type reconstruction", which conflates what we distinguish as three separate concepts. To avoid confusion with prior informal usage, we use the capitalized names for our more precise definitions.

2.1 Prior Approaches

Type Inference, when not qualified with any modifiers (such as "from binary code"), is well studied for programming languages, and commonly refers to the single-language task of automatically inferring types for a source program. While incredibly useful in practice for programming, due to its assumption of a single language, techniques from it (e.g., Hindley-Milner type inference [14, 23]) are not (directly) applicable to decompilation, which focuses on two-language situations.

Some recent techniques [6, 7, 21, 39] for automatically recovering source-like types from compiled code utilize learning-based approaches. We call such techniques *Type Prediction*. Despite showing increasingly impressive success at predicting source-level types, it can be hard to depend on the correctness of predicted output, especially in scenarios that might differ drastically from their training corpus. Fundamentally, such techniques come with no guarantee about usage on out-of-distribution binaries. If the usage scenario is similar to the ones they are trained on, then there is some expectation of good results. Unfortunately, most reverse engineering happens on source-unavailable code where it would be difficult to make the in-distribution assumption. This problem perhaps reaches its most extreme for malware, where malicious actors might be incentivized to create binaries that are far away from training data.

Thus, we focus on reasoning-based (*deductive*) techniques going forward. These techniques derive types through a series of deductions, often by understanding the semantics of the program in question. Although deductive type inference approaches may have differences in sensitivity of analysis, constraint solving techniques, and even choice of static- or dynamic-analysis, a common theme is a focus on a balance between the accuracy and conservativeness of the types produced. While rarely (if ever) explicitly concretized as such, the results of these approaches can, in theory, be traced through a series of deduction steps.

We use *Type Recovery* to specifically denote deductive approaches that attempt to recover the source-level types in the program that was compiled to the target binary. This is the primary focus of almost all the techniques in the aforementioned survey [5]. A key observation of our work is that all existing approaches for Type Prediction and Type Recovery attempt to recover *the* source-level types from the original program —we discuss the feasibility of such recovery below.

2.2 On the Impossibility of Type Recovery

Type Prediction and Type Recovery both assume that there is an objectively correct ground truth. Indeed, it seems obvious that if some initial source code is compiled to a binary, then that original source code should provide the ground truth for its compiled machine code. More precisely, the (implicit) assumption being made is that for an arbitrary source program S compiled to the binary program $[S]$, the ground-truth ζ for each variable can be modeled as a perfect recovery tool, achieving equivalence to source-level types γ modulo naming (i.e., $\forall S, v \in \text{vars}([S]).\ \rho(\zeta([S], v)) = \rho(\gamma(S, v))$, where $\rho(\tau)$ is the α-normalized representation of source-level type τ).

Yet, this view on type inference (or decompilation, more generally), is fundamentally flawed. More precisely, even a single counter-example of two source-programs $S_1 \neq S_2$ such that $[S_1] = [S_2]$ and $\exists v.\ \rho(\gamma(S_1, v)) \neq \rho(\gamma(S_2, v))$ would imply the impossibility of binary-level recovery ground truth, as a trivial consequence of universal quantification, reflexivity, and substitution of equality.

We demonstrate multiple such counterexamples through a collection of hand-crafted examples that show how severe the lossiness of the compilation process can get.[5] We emphasize that we list multiple *simple* examples to demonstrate that such scenarios are not a mere one-off pathological case (which would still be sufficient to prove impossibility), but instead are caused by pervasive common patterns found in source code. Furthermore, this lack of a singular ground truth implies that source *recovery* (even modulo comments and naming) is impossible. We believe that attempting to achieve the impossible leads one down a rabbit hole of increasingly sophisticated and complex techniques to recover the specific kinds of types that appear to be common in specific evaluations. Indeed, recent work [22, 25] on Type Recovery has continued to show increasing sophistication, both in techniques and the expressivity of types produced. At its extreme, this could effectively become a collection of idioms hyper-specialized to a particular (version of a) compiler. Nonetheless, even this recent work suffers from the same common (implicit) assumption.

Figure 1a demonstrates a pair of functions that differ in sizes of arguments and return types, yet compile to the exact same assembly code. Similarly, Figure 1b shows a pair of functions that differ on their usage of pointers, yet compile to the same machine code. Colocation of variables on the stack can be intentional (using a `struct`) or incidental, as shown in Figure 1c. Clearly, memory usage of "the" ground-truth cannot be relied upon. Worse, optimizations can confound operations performed on variables too—arithmetic operations, or even entire loops, can be switched out, as shown in Figure 1d.

An obvious caveat is that introducing additional context *outside* the analyzed code can aid in disambiguating the types. For example, since our examples are individual functions (due to space constraints), an inter-procedural or dynamic analysis might be able to disambiguate them. However, note that such analysis introduces external context that was not mentioned in the counterexample (callers, all program inputs known, etc.). If we expand the set of potential inputs to the hypothetical perfect recovery tool (even if such inputs are more than just the compiled binary code, but as long as they do not trivialize the analysis), then similar counterexamples

[5] Tested with GCC 13.2 with `-O2` on x86-64, as the compiler.

```
uint8_t foo(uint8_t x) {
  return ~x;
}
int32_t bar(int32_t x) {
  return ~x;
}
```

(a) Confounding sizes

```
char* foo(
    char* x,
    size_t i
) {
  return &x[i];
}
int64_t bar(
    int64_t a,
    int64_t b
) {
  return a + b;
}
```

(b) Confounding pointerness

```
int foo(int x) {
  volatile int y = x;
  volatile int z = x;
  return y + z;
}
struct S{int y; int z;};
int bar(int x) {
  volatile struct S s;
  s.y = x;
  s.z = x;
  return s.y + s.z;
}
```

(c) Confounding stack shape

```
uint32_t foo(uint32_t n)
{
  uint32_t r = n;
  while (n-- > 0) {
    r++;
  }
  return r;
}
uint32_t bar(uint32_t n)
{
  return n * 2;
}
```

(d) Confounding operations

Figure 1: Each pair of functions compile to the same assembly code, demonstrating the flaw of assuming a single ground truth

could still be designed at that level.[6]

With all these examples, it can initially feel like all hope of type inference is lost. However, reverse engineers often do not require perfect type recovery.

2.3 Type Reconstruction: Capturing Behavior

Recognizing that recovering *the* ground truth is impossible, we turn to understanding what reverse engineers really want from decompilation. We argue that rather than perfect recovery of source, in practice, reverse engineers desire output that captures the binary's observable behavior, i.e., a summary of observed/allowed operations on each variable. Types are "merely" an encoding of this information into an easily digestible form. Type inference, even if the produced types are wildly different from the original types, is still useful when they are compatible with the observed behavior of the program, since this allows the reverse engineer to understand what is *actually* being executed, or how to interface with it.

Hence, we propose a new perspective—*Type Reconstruction*. Like Prediction and Recovery, Reconstruction takes in low-level code and produces high-level types. However, in contrast to the prior approaches, it does not attempt to infer or recover *the* ground truth; instead, it focuses on *(re)constructing* types that are compatible with the observed behavior.

Specifically, the goal of Type Reconstruction is to construct the "nearest" source-level types that capture observable behaviors, using deductive techniques and thereby producing types that can, at least in theory, be reasoned about. We believe that adopting this more realistic goal is essential to improving type inference in practice. This goal naturally accounts for different compilers' interpretation of source code and does not attempt to codify any particular compiler's behavior(s). Instead, it focuses on providing useful types.

Recognizing the lack of a singular ground truth does come with a non-trivial downside—we must redefine what it means for a Type Reconstruction tool to produce "correct" or "accurate" types, i.e., how should one evaluate the effectiveness of Type Reconstruction? We detail our specific evaluation further in §5, but morally, we aim to capture features of types that reverse engineers would prefer the tool to get correct.

[6]Notice a similarity with Rice's theorem [29].

3 TRex: System Design

TRex is the first tool explicitly designed with the goal of Type Reconstruction. We describe some of our high-level design decisions, followed by TRex's architecture below.

3.1 Structural Types Capture Behavior

Since the de facto output language of decompilers is C, it is quite tempting to use C-like nominal types during Type Reconstruction. However, we believe that this would be a mistake, leading to the need for complicated mechanisms, such as those used in prior works on Type Recovery. As an example, we might discover that a certain location supports 64-bit integer addition; while this location could be thought of as an `int64_t`, `uint64_t`, or `undefined64`, if the analysis were to internally represent this location as any of these, then it is necessarily introducing inaccuracies; in particular, both the `int_t` types imply that the type (without casts) does not support pointer dereferencing, while the `undefined64` (even though it captures the size correctly) does not capture the observation that the type supports integer addition. This means that later analyses that use these types cannot rely upon them and must either rely on side information, or re-analyze the code that led to the observation of the 64-bit addition.

We argue instead that the types most natural to Type Reconstruction are behavior-capturing types, i.e., *structural types*.[7] In particular, rather than attempting to match behaviors in the executable to C-like types, the types themselves need to capture behaviors precisely, even if the behaviors cannot be represented as C-like types. Structural types are freed from

[7]In this paper, we are drawing a contrast with nominal types, where structural types mean a static version of duck-typing [28]. This usage of structural types should not be confused with the alternative definition used in some PL circles to contrast with sub-structural type systems.

```
StructuralType {
  COPY_SIZES  {8, 16, 32}
  INTEGER_OPS   {
    Add32, Sub32, Mult32, UDiv32, SDiv32, URem32, SRem32,
    And32, Or32, Xor32, Eq32, Neq32, ULt32, SLt32, UCarry32,
    ... }
  POINTER_TO None
  COLOCATED_STRUCT_FIELDS None
  ... }
```

Figure 2: Example Structural Type. Simplified for presentation purposes. Equivalent to `int` in C.

the constraints of human-readability, and working with structural types as far as possible during type reconstruction allows us to maintain high precision and conservativeness (§3.2).

Figure 2 shows an example structural type, equivalent to C's `int` type. Clearly, exposing the full precision of structural types to users is untenable and would detract from understanding. For example, if a type supports 64-bit addition, subtraction, multiplication, right-shift, and more, then for practical purposes, it is reasonable to believe that it would support other operations such as division too, and that it is best shown to humans as an `int64_t`. Thus, Type Reconstruction, despite best conducted with structural types, must still deal with nominal C-like types. Since the structural types are more precise, we capture this through a phase that performs "type rounding" (introducing the division operation, in the above example), followed by projection to C-like types (§3.3.5).

Overall, we make the design decision to stick to structural types *as far as possible*, only switching to nominal types in the last stages of reconstruction. This maintains a high degree of fidelity with the actual observable behavior of the machine code as far as possible. The output of our tool can be consumed either by a downstream tool (which can use the more accurate structural types) or by a human (who can use the easier-to-read nominal types).

In addition to using structural types, decisions must still be made regarding the expressivity of such structural types. We choose to support both aggregate and recursive types, since their expressivity better captures machine-code behavior, compared to only supporting primitive types. However, we do not support polymorphism[8] (neither parametric nor ad-hoc). This is because we have observed outputting polymorphic types to be useful only in a small number of toy examples—in practice, compilers monomorphizing and optimizing code leads to sufficiently different code and types, warranting separate attention rather than collapsing via polymorphism.

[8]Polymorphism here has the standard definition of the same function supporting multiple types. For example, `malloc` would have the more precise type of $\forall \tau : \texttt{size_t} \rightarrow \tau\texttt{*}$. C's type system is insufficiently expressive to represent this type and instead uses `size_t` → `void*` and casting. Languages that support polymorphism (e.g., C++) tend to monomorphize.

3.2 Conditional Conservativeness

Binary analysis is difficult, even undecidable in many cases [17], and Type Reconstruction is no exception. Tools thus must make tradeoffs between soundness and completeness—put differently, there must be a balance between the conservativeness and utility of the output from a tool. Naïvely, one might wish for a tool that never makes any non-conservative leaps, so that it cannot mislead (except by omission). However, a hypothetical fully-conservative analysis would quickly find itself unable to provide anything useful.

For example, even a single `call` instruction would lead to the halting problem via "does the callee return?"; thus, a fully-conservative analysis might not be able to progress beyond a `call`. Nonetheless, we would like at least some guarantees from our tools. Thinking about the composability of guarantees from analyses, we realize that *conditional conservativeness* appears to be a good tradeoff (especially in the long run, as various analyses are built and improved). We define conditional conservativeness to mean conservativeness only under the condition that the "upstream" analysis provides true facts. Said differently, a conditionally conservative analysis will output only true facts within the axiomatic system set up by the upstream analysis. Returning to the example situation of the `call` instruction, an upstream analysis is in charge of reachability, and the downstream analysis (such as Type Reconstruction) only needs to be conservative if the upstream analysis produced true facts.

Unfortunately, even conditionally conservative analysis is untenable in practice, since it still completely cuts off "obvious" inferences that are technically non-conservative, e.g., if a type is seen to support division, it is reasonable (but non-conservative) to assume it supports the modulus operation.

In light of these observations, while it might seem tempting to give up on conservativeness (even conditional), we believe that there is a better approach—managing the loss of conservativeness. In particular, tracking (and supporting the toggling of) the introduction of non-conservativeness introduced by different analyses becomes essential to building larger analyses and tools. Specifically, tools may make opportunistic inferences, but they must also support disabling those inferences. We believe that this design forms a useful blueprint for various binary analyses to adopt, in order for us to improve each component without compromising the guarantees provided by the overall composition.

For TRex, we choose a largely conservative stance for decisions made by the tool, with some opportunistic inferences when purely conservative inference would be incredibly unhelpful. This means it can take advantage of common patterns, improving the default utility of its output, while supporting a more conservative analysis when an analyst (or another analysis) discovers that a particular opportunistic inference is being unhelpful. For example, after a `call` instruction, without a global analysis confirming

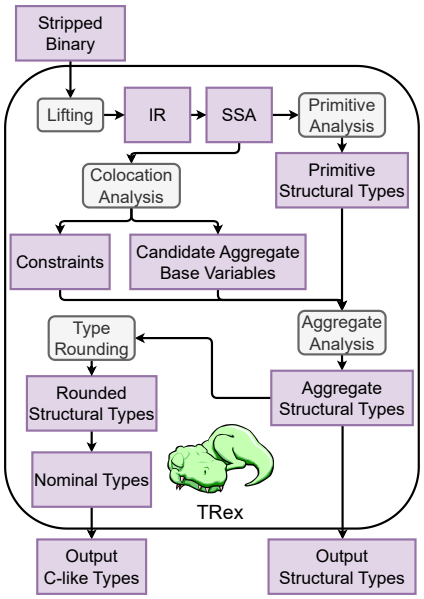


Figure 3: Phases in TRex

that no memory corruption vulnerabilities exist, any *purely* conservative analysis cannot assume that the stack pointer is returned back to where it was prior to the call. This clearly would not be helpful for most executables. Thus, in TRex, we opportunistically assume that the stack pointer is returned to the expected value. However, we also recognize that this introduces non-conservativeness, and this opportunistic assumption (like all ∼15 other such similar introductions of non-conservativeness) can be disabled with a CLI flag.

3.3 TRex Architecture and Analysis Phases

TRex, like many prior deductive tools (§6), can be thought of as a constraint generator and solver, and thus the interesting aspects of its design derive from the specific phased architecture, choice of constraints in each phase, and techniques used to solve them. TRex consists of multiple phases that we summarize in Figure 3 and describe below. Analysis begins by lifting from disassembly. It is flow-sensitive, but path- and context-insensitive. We additionally choose to perform type reconstruction intra-procedurally—we do not yet propagate types inter-procedurally (although this could be added with straightforward, albeit non-trivial, engineering work[9]). Similar to prior tools, TRex is not designed for obfuscated code—handling this is an orthogonal challenge in deobfuscation research.[10] The output from TRex is either C-like types (useful for human analysts; specifically, all C primitives, arrays, unions, and structs, including recursive types, as well as common decompilation types like `unknown`*N* and `code`) or precise-but-verbose structural types (useful for downstream analyses that can aid in better decompilation [24]).

3.3.1 Input, Lifting, and SSA

An important decision is the choice of input to a Type Reconstruction tool. A plausible input could be from an existing decompiler (potentially with type information removed if the decompiler already does some type inference), since that allows one to take maximum advantage of other analyses that already exist in the decompiler. Another plausible input is the disassembly (or somewhat equivalently, an architecture-independent lifting of it), since that gives the highest level of detail about what is actually happening at the machine level. Of the two, we opted for the latter, so as to obtain a higher degree of assurance in our tool's output. In practice, TRex takes disassembly-equivalent lifted code from a binary analysis framework as input, and lifts it to its own internal intermediate representation (IR) that we discuss further in §4.1. From this IR, to consistently track variables without worrying about mutation or (immediate) aliasing, we compute the static single-assignment (SSA) form of the code. Future stages reconstruct types for all variables produced in this stage.

3.3.2 Primitive Analysis

We begin the type-related analyses in TRex with primitive analysis, which collects and solves low-level constraints (Appendix A) required by instruction semantics. The solved constraints provide a partial map from SSA variables to members in a graph of primitive structural types. In practice, integrating both constraint collection and solving works ergonomically for primitives. Performing them in lock-step removes the requirement for explicit reification of a bag of constraints, and thus we describe the imperative view to ease understanding. During a walk through the SSA IR, we begin assigning types to each variable (initialized with the empty type at first) by iteratively adding observed operations to relevant types (such as "this variable supports 32-bit addition"). This provides us with the primitive size and operation information about the variables that will be useful for future phases. Additionally, certain IR instructions cause the introduction of equality constraints, causing types to merge together, either always (for SSA ϕ-nodes), or opportunistically (depending on the operation performed, e.g., comparison operations). An additional global-value-numbering pass identifies further opportunities for equality constraints to merge types. Since structural types are stored as a graph (§4.3), any equality constraints (added at any stage in the overall pipeline, including this one) might suddenly cause a recursive type to appear—they do not require a special "look for recursion" pass.

[9] Note that extending *decompilation* from intra- to inter-procedural might be conceptually challenging, but extending Type Reconstruction is not. Direct calls propagate types; indirect calls introduce controllable non-conservativeness (manageable by, say, value-set analysis). Multiple paths through a single `void*` (e.g., `malloc`) in source produce precise unions.

[10] An interesting question for future research: "what should 'successful type reconstruction' even mean for a Church-numeral-obfuscated binary?"

Naïvely, one might expect a lot of merging, but we note that most operations do not cause merging. For example, an instruction like $a < b$ requires both a and b to be the same type; but $a \leftarrow b + c$ does not necessarily mean that a, b, and c are the same type—the addition operation is as valid for pointers and offsets as it is for integers.

3.3.3 Colocation Analysis

Having recognized primitive structural types, we begin collecting constraints (Appendix B) that will help us discover aggregate types. These constraints recognize a dereference of a variable at an offset from another variable. Unlike the prior phase, representing the constraints produced by this phase explicitly, rather than implicitly (combining this phase with the next), simplifies both implementation and understanding. The constraints store both a static (constant) or dynamic (symbolic variable) offset, along with the base and the accessed variables. Dynamic offsets indicate scanning across memory, and help identify arrays. Each constraint may be derived either from a single instruction, multiple instructions, or even a collection of constraints and instructions. To aid this phase, we also implement an on-demand constant-folding analysis to identify static offsets that might otherwise be imprecisely considered dynamic. Alongside the constraints, from this phase we also obtain the base variables for candidate aggregate types.

3.3.4 Aggregate Analysis

Using the constraints and types from prior analyses, the aggregate analysis recognizes when different types are consistently colocated, and thus can be grouped into the same `struct` or array. It finalizes the offsets and sizes of structural types, including non-aggregate types.[11] The structural types it identifies are analogous to C `structs` (including support for flexible array members—unsized arrays at the end of a `struct` [15]) and C arrays.

Unfortunately, we find that in the general case, this phase is impossible to determine precisely, at least with static analysis. For example, if a function takes a pointer to a `struct` of two integers and only touches the first, then it is impossible to even detect that there was any colocation. Worse, simply because the compiler places two variables beside one another consistently, this may not reflect programmer intent. Instead, colocation of variables is influenced greatly by the compiler's interpretation of memory. To accurately tease apart such variables, one needs to find a contradiction to the colocation hypothesis. A naïve fix is to instead only consider colocation when there is an obvious offset operation in play, recognizing that compilers rarely perform such offsets unless they are from a single `struct`. However, this cannot work in practice, since compilers often over-read bytes, or even cross across fields in `memcpy`-like operations, causing spurious fields of invalid sizes to appear. Recognizing this inherent difficulty in aggregate analysis, and that a purely conservative analysis would be forced to assume non-aggregate always, we instead opt for a more pragmatic approach that combines careful speculative-but-safe non-conservativeness and the ability to turn it off (§3.2).

[11] While TRex does not *require* variable boundary information, if provided, it is used during this phase when finalizing sizes.

3.3.5 Type Rounding

Next, we begin the process of connecting the extremely detailed structural types to more human-readable nominal types. We call this the *type rounding* phase, to evoke the similarity to rounding a real number to an integer. For types, this phase rounds *up* structural types to their nearest (union of) primitive types. For example, if a type has been seen to support integer multiplication, it is relatively natural to assume it would also support division, since there is no C primitive type that supports multiplication but not division.

Type rounding can round types up to any collection of primitive types. However, since C is the lingua franca of current decompilers, we build in support for C. Note that our implementation is agnostic to this choice, and other sets of primitive types could easily be supported. Indeed, one primitive that we include by default that does not directly exist in C is `code`, which allows representing pointers to executable memory (such as function pointers, or the return address slot on the stack) as `code*`, rather than `void*`. Note that despite rounding up to C-like types, the output of this phase is still structural types.

Clearly, this phase necessarily makes opportunistic non-conservative inferences with respect to the claims of observable behaviors that structural types capture. Nonetheless, we would like the smallest number of non-conservative inferences possible. More precisely, we define type rounding as an optimization problem to find the smallest subset of primitives that, when unioned, form a supertype of the input type. Unfortunately, this problem of type rounding is NP-hard. We show this by a reduction from Set Cover [16], an NP-hard problem to find the smallest collection of sets that cover all elements in the union of those sets. Specifically, if we transform each set into a primitive type, with a structural type representation that has the relevant elements represented as observable behaviors, then finding the Set Cover is equivalent to performing a type rounding of the structural type that contains all behaviors, and using the resultant union type to recover the relevant primitives, and thus sets. This inherent theoretical complexity of type rounding implies that, in the general case, this phase is difficult to perform.

Nonetheless, for the purposes of decompiling to C-like types, we observe that our greedy approximation algorithm (Appendix C) is both performant and produces sufficiently good results. Our algorithm for type rounding starts by representing the types using vectors and matrices, reducing the

problem of rounding to that of finding the smallest x such that $Ax \geq b$, where b is the vector of indicator variables for each component of a structural type (e.g., $\texttt{Add}_{32}$, $\texttt{Add}_{64}$, ..., would have separate rows), A is the matrix consisting of similar vectors for the allowed primitive types, and x selects which primitives are to be `unioned`. Starting x at all 1s (representing a union of all primitives), we repeatedly flip the most *expensive*-but-*unnecessary* primitive to 0, until we cannot anymore, returning the union of all remaining primitives. A primitive is considered unnecessary iff its removal from the union does not prevent the union from being a supertype (i.e., flipping its element in x to 0 does not negate the inequality $Ax \geq b$); cost is computed as the number of enabled indicator variables in A for that type.

3.3.6 Nominal-Type Reconstruction

This phase converts the large and complex structural types to human-readable C-like types. This both identifies and provides names to all `structs` and `unions`. A recursive walk through the graph of all structural types helps identify all C primitives, `structs`, `unions`, pointers, and arrays. An interesting case that requires careful management during the graph walk is that of `structs`, since they can be easily confused with their 0th offset. We handle this by maintaining two names for each type, which are identical for all types other than `structs`, where one refers to the 0th field and the other refers to the `struct` itself. This graph walk approach works well even for recursive types, including `struct`-of-`structs`, with the minor caveat that generally speaking, a `struct` that contains a `struct` as its 0th field is indistinguishable from the flattened `struct`; thus the flattened version is picked during reconstruction. However, all non-zero offset fields *can* be distinguished and are handled as expected.

4 Implementation

TRex is implemented in Rust, consists of ∼9,600 source lines of code, and took approximately 2 person-years of effort to build. It is agnostic to the choice of binary analysis framework to plug into; for this paper, we use Ghidra. This involves a tiny script in the binary analysis framework (∼100 SLoC of Java for Ghidra) that outputs lifted code (P-Code for Ghidra) into a file that can then be ingested by TRex's lifter (∼900 SLoC of boilerplate Rust) into its own internal intermediate representation. Following this, the analysis proceeds in a series of phases (§3.3) that work on a graph representation of structural types, finally outputting either human-readable C-like types for reverse engineers, or machine-readable representation for downstream analyses. We describe important components in more detail below.

4.1 Intermediate Representation in TRex

Choosing an intermediate representation (IR) for code is critical for expressivity and ease of building any static analysis tool. For TRex, we design a custom IR that is inspired by, but distinct from, Ghidra's P-Code IR. We pick this custom IR since P-Code has idiosyncrasies specific to Ghidra, and we have somewhat different goals that are better served by a custom IR. For example, Ghidra conflates various kinds of `call` and `branch` instructions into a fairly small set of instructions (some with implicit overloaded semantics), which we break up into separate instructions. Also, Ghidra mixes constants, addresses, and registers into a single kind of type called a `varnode` (distinguished by magic constants picking an address space for each); since we implement TRex in Rust, we use algebraic types to handle these more cleanly. Also, Ghidra has (what we believe to be) shortcomings in its lifting that we fix up during the lifting to our IR, such as using an explicit `NOP` instruction in the IR (which Ghidra does not have), which is useful for maintaining alignment between the real machine instruction-pointer (e.g., `rip`) and the IR-internal program counter, rather than relying upon implicit fallthroughs at a jump target if there is a lack of instructions. We also explicitly support (a small number of) vector instructions, which Ghidra instead handles as architecture-specific magic constants in its catch-all `CALLOTHER` instruction. Additionally, we support an explicit `HAVOC` instruction that denotes a conservatively-under-specified clobbering of its output (used, for example, when lifting rare vector operations when we have not yet found the need for more precise semantics).

The in-memory representation of our IR uses a Rust vector (`Vec`) of the (lifted) program instructions and properties such as maps between IR addresses and machine addresses (since a single machine instruction might expand to multiple IR instructions), and maps to maintain basic-block and function information. A special `FunctionStart` and `FunctionEnd` instruction denotes the *singular* start and end of a function, rather than allowing multiple entries and exits from functions.

Our static single-static assignment (SSA) IR is implemented as a view on the core IR that assigns SSA variable names to it. Unlike textbook SSA, we maintain ϕ-nodes in a separate list, thereby allowing a convenient map between instructions in both forms.

4.2 Representing Types in TRex

Each structural type is a precise representation of observable behaviors for a particular variable. The structural type consists of information about size, colocation, dereferencing, and operations observed upon it, represented as sets, booleans, and indices. Figure 2 shows an example structural type.

The set of operations within a type precisely captures observable behavior; each is roughly analogous to a machine operation—for example, each of the following is an entirely

distinct operation that could exist in a structural type: $\texttt{Add}_{32}$, $\texttt{Add}_{64}$, $\texttt{Sub}_{32}$, $\texttt{Copy}_{32}$. Note how the existence of 32-bit addition ($\texttt{Add}_{32}$) does not (immediately) indicate that the type supports 32-bit subtraction ($\texttt{Sub}_{32}$). We also note that while it might seem superfluous to maintain size information with the operations (instead of maintaining size information about the whole type), this is crucial to handle situations such as the `union(u32,f64)` being distinct from `union(u64,f32)`.

We represent dereferencing via a points-to relation on the type by maintaining an `Index` into the graph of all structural types (§4.3). This graph of all types supports encoding recursive types that may be introduced by solving constraints in any of TRex's stages. Aggregate types (e.g., `structs`) maintain colocation information as a map from non-zero offsets to `Indexes` into the graph. The offsets are non-zero, since the operations for the first field of a struct are the same as the struct itself, modulo colocation; thus offset 0 always refers to the "current" type. Finally, we do not represent `struct` padding in the structural type explicitly; this allows us to distinguish between a `struct` padding byte (unused) and an `undefined1` (used single byte, but nothing more known; cannot be a `char`, `uint8_t`, etc.).

4.3 Joinable Containers

Since the design of TRex requires directly dealing with possibly-recursive types, we must represent the structural (and later, C-like) types in some graph structure. Rust's type system is famously known to be graph-unfriendly (although good graph libraries exist). Additionally, we require the ability to merge separate, partially-specified structural types together when we recognize them as equal (for example, two different instructions might help us learn that some location X's type supports 64-bit integer addition, and Y's type supports dereferencing, and then a third operation might show that X and Y have the same type; here, we need to merge the types of X and Y). Furthermore, while in the process of merging two types, we might further discover other types that require merging (for example, p_X and p_Y might point to X and Y; once we find that p_X and p_Y must have the same type, we must also merge the types of X and Y).

Hence, we not only need a graph structure, it must also support this (potentially recursive, and deep) join/merge operation. To solve this, we implemented a custom graph data structure that supports safe access to graph members by storing members into an arena with strongly-typed `Indexes`. This structure is parametric in its objects, only requiring that the objects support *some* join operation. Inspired by the Disjoint-Set (aka Union-Find) data structure, we maintain a canonical internal index for each external `Index`, so that merge operations do not require a global scan to update all indices.

During any join operation, we maintain a queue to schedule further join operations during the process of each join, repeatedly processing each join until it terminates. We guarantee termination by explicitly checking for repeats (which can happen when joining two recursive structural types), and noticing that, modulo loops, each join operation decreases the total number of distinct structural types available.

The arena itself behaves (as expected) as an arena allocator, handing out new strongly-typed `Indexes` upon allocation request; freeing is performed en masse, deallocating the entire arena at once. However, this is not the only possible deallocation operation, since the arena also supports garbage-collection and compaction, which can be explicitly invoked by providing it with a series of roots to keep alive. By maintaining a globally-unique arena identifier, indices into one arena are checked to not be accidentally used with another arena, and sentinels confirm that no use-after-free can occur with these `Indexes` after invoking a garbage-collection pass.

5 Evaluation

We answer the following research questions, through both a qualitative and a quantitative evaluation of TRex.

- **RQ 1.** Are there certain aspects of types where the state of the art fails, while TRex succeeds?
- **RQ 2.** Does TRex significantly improve upon the state of the practice on prior real-world benchmarks?
- **RQ 3.** Do compiler optimizations affect reconstruction?
- **RQ 4.** How does TRex compare to state-of-the-art Type Prediction (i.e., machine-learning based techniques)?

Comparison to Prior Work. As described in §1, most tools from prior work on deductive type inference are unavailable. Thus, the majority of our comparisons are done against Ghidra [26], a popular open-source decompiler. Where feasible, we include other comparisons as well, such as comparison against other popular decompilers [30, 36] used in practice by reverse engineers. Unfortunately, despite reaching out to the authors, we were unable to obtain the deductive tools in prior published work (§6). Thus in the following sections, we do our best to include comparative details, within the constraints of what we could get access to. TIE [18] is unavailable for comparison (as also discovered by other prior work [42]), but it is superseded by improvements made in more recent work. Retypd [25] appears to be a highly capable system on paper, but it is unavailable. The first author did point us at a public open-source reproduction (BTIGhidra [35]); unfortunately, the reproduction suffers major flaws (failing entirely even on small trivial examples; we have filed issues on their issue tracker detailing shortcomings) that prevent a valid comparison against the theoretical performance of the system described in the paper. The authors of the more recent OSPREY [42] were unable to share their code "due to some commercial and patent concerns", but graciously shared their output on a version of GNU Coreutils (detailed in §5.2),

which we base our comparison to OSPREY upon. Finally, as we explain in §2.1, our work focuses on reverse-engineering scenarios where we cannot assume the binaries are in distribution; nonetheless, we include a comparison to the most recent Type Prediction work (ReSym [39]). The publicly available artifact for ReSym is unfortunately incomplete, and the authors of ReSym were unable to share parts of their code "due to some property concerns"; thus we implement missing components and fix issues in the available portion (detailed in §5.3), and base our comparison to ReSym upon this fixed version.

5.1 Qualitative Evaluation

To help answer RQ 1, we use a simple example of a singly-linked list. Using the Decompiler Explorer [37], we compare TRex against the outputs of popular decompilers, namely (i) Binary Ninja 3.5.4526 [36], (ii) Ghidra 11.0 [26], and (iii) Hex-Rays (IDA Pro) 8.3.0.230608 [30]. We compile the C code in Figure 4 to an x86-64 ELF object file with GCC 11.4.0, strip away debugging information, and pass it to all of the evaluated tools, including TRex.

```
struct Node { int data; struct Node* next; };
int getlast(struct Node* n) {
  struct Node* nxt = n->next;
  while(nxt != 0) { n = nxt; nxt = n->next; }
  return n->data;
}
```

Figure 4: Source Code for the Singly-Linked List Example

Focusing on the type of the parameter, all three decompilers successfully recognize that it is a pointer but fail to detect the `struct`. Instead, they identify it as a primitive type—`int32_t*` for Binary Ninja, `undefined4*` for Ghidra, and `unsigned int*` for Hex-Rays. A reverse engineer *can* request further automated analysis for particular variables in Ghidra and Hex-Rays if they manually decide it could be a `struct`. For example, if they invoke Ghidra's "Auto Fill in Structure", then Ghidra updates the type to be a `struct` with six fields: an `undefined4`, four `undefineds` (indicating padding, distinct from `undefined1`), and an `undefined4*`. Notice that despite explicitly requesting structure analysis, Ghidra is unable to detect the recursive nature of the type.

As discussed earlier in §5, we were unable to obtain the deductive tools in prior published work for a fair comparison, but here we include our best interpretation on how they would perform on the same simple example. TIE [18] explicitly states that it does not support recursive types. In contrast, at least on paper, Retypd [25] should be able to handle the simple singly-linked list example; in practice, the public reproduction pointed to by the authors (BTIGhidra [35]) is unable to improve upon Ghidra's types (above) even on this small example. Based on the output provided by the authors of OSPREY [42], its output schema supports only size information for struct fields; so even in an ideal case, OSPREY can only output `Struct<4,4,8>` and cannot express the recursion or the unused padding. We defer comparison against non-deductive Type Prediction to §5.3.

```
// TRex's structural type for n : t31
t31 { UPPER_BOUND_SIZE  8
      COPY_SIZES  {8}
      POINTER_TO  t33
      INTEGER_OPS   {Add_8, Sub_8, ULt_8, SBorrow_8} }
t33 { COPY_SIZES  {4}
      INTEGER_OPS   {ZeroExtendSrc_4}
      COLOCATED_STRUCT_FIELDS   8   t31 }

// TRex's C type for n : t1*
struct t1 { uint32_t field_0; t1* field_8; };
```

Figure 5: TRex output on Singly-Linked List Example

In contrast, TRex produces the output in Figure 5. The structural type for the parameter is `t31`. For human consumption, TRex rounds this to the C type `t1*`, which successfully captures the correct `struct` shape, including the recursion and padding (`field_N` denotes the field at byte offset `N`). The only part of the type where TRex does not perfectly match the source is the first field, where the signedness differs. However, notice that the code itself does not interact with this value, and thus either signedness would be consistent. TRex picks `uint32_t` since that is the most precise primitive available that is consistent with the only observed operation on the variable from the disassembly, namely the zero-extension. Note that C programmers often use `int` (which means `signed` rather than `unsigned`) when the signedness might not matter; thus, TRex also supports a CLI flag (§3.2) that makes it prefer the signed variant in such ambiguous-signedness situations.

The specific order of the `Node` fields in Figure 4 makes this example challenging, since recursion and colocation influence one another. Nonetheless, when we try the easier ordering (with recursive pointer being at offset 0), none of the three existing decompilers does any better at recognizing either the `struct` or the recursion (producing `int64_t*`, `undefined8*`, and `_QWORD**`, for Binary Ninja, Ghidra, and Hex-Rays, respectively). TRex produces the expected type.

To understand the impact of colocation and aggregate analysis in TRex, we perform an ablation study by disabling these particular phases—this produces the parameter type `uint32_t*` for `n` in Figure 4, similar to the evaluated decompilers. However, in the easier case where recursion and colocation are independent, TRex successfully detects the recursion, warns about the infinitely dereferenceable pointer, and outputs the closest C-like type that captures this, namely `void*`.

5.2 Quantitative Evaluation

To explore how successful TRex is on benchmarks commonly used by prior work and help answer RQ 2, we perform a quantitative evaluation against 125 binaries from prior work.

As discussed in §1, prior work rarely makes reproducible benchmarks available for comparison. Thus, we develop a set of reproducible benchmarks that can be used to compare against state-of-the-practice and future tools.

The benchmarks themselves consist of GNU Coreutils 9.3 and SPEC CPU® 2006 [34] (hereafter, COREUTILS and SPEC, respectively). We compile each benchmark program with debug symbols to obtain ground truth variables and types. Then, by stripping the debug symbols, we obtain a stripped executable. The tools have access to only this stripped executable and the ground-truth variable information (location and size of each variable). They are expected to output type information, which is then scored against the ground truth types. We use all 108 binaries from COREUTILS. For SPEC, of the 29, we skip all 10 Fortran binaries because their DWARF debug symbols appear to be invalid, producing either empty or nonsensical sets of ground truth variables and types. In addition, we are also forced to skip two of the C++ binaries because Ghidra could not analyze them correctly.

As discussed earlier in §5, we were unable to obtain any of the deductive tools in prior published work for a fair comparison. However, the authors of OSPREY [42], despite being unable to share their code, were kind enough to share their output on their pre-compiled version of GNU Coreutils. Unfortunately, this output covers only approximately 31.8% of the variables in the ground truth, and it is missing one of the binaries (the authors could not locate their build scripts). Due to the missing 68.2% of variables, their output underperforms even against a trivial baseline (described below). This means a direct comparison is infeasible, and any measurement taken on the subset would be skewed [11] (in addition to other variables, all register-based variables, and thus all function parameters, are omitted in their output). Thus, we are forced to limit ourselves to our earlier qualitative comparison against OSPREY.

Amongst practically available tools, we compare against Ghidra, since both IDA and Binary Ninja require commercial licenses to perform batch analysis,[12] and our practical anecdotal evidence have neither doing particularly better than Ghidra at type inference. We additionally compare against a trivial baseline (that outputs an `undefinedN` of the correct size on each variable) to provide context on the quality of existing state-of-the-art tools. Again, we defer quantitative comparison against non-deductive Type Prediction to §5.3.

As discussed in §2.2, it is impossible to define any single ground truth for binary type inference, assuming no additional external context. Any quantitative evaluation is thus necessarily an approximation of the quality of the output types. Indeed, we hypothesize that this underlying phenomenon is the cause of the largely mutually-incompatible evaluation methodologies appearing in prior work. Nonetheless, one commonality in prior evaluation methodologies is a notion of true positives, and the use of precision and recall (sometimes summarized via an F1 score).

[12] Additionally, IDA's EULA explicitly disallows using IDA for "publishing data or analysis relating to the performance of the Software" [13].

Tool	COREUTILS Prec.	COREUTILS Recall	COREUTILS F1	SPEC Prec.	SPEC Recall	SPEC F1
Trivial Baseline	0.00	0.00	0.00	0.00	0.00	0.00
Ghidra [26]	0.09	0.98	0.16	0.13	0.48	0.21
ReSym [39] (§5.3)	0.42	0.20	0.27	0.29	0.19	0.23
TRex	0.32	0.94	0.47	0.28	0.99	0.44

Table 2: Standard summary metrics. A type is considered "correct" if its C representation (with names normalized) is identical to the source type. See §5.2 and §5.3.

Unfortunately, prior work rarely specifies what it means to have a correct type, and sometimes chooses a generous definition of correctness. As an example, OSPREY [42] reports using an otherwise underspecified homomorphism to determine true positives, implying that a projection of the C types was taken before being compared; from the output data provided to us by the authors, we can surmise that their projection considers a type to be correctly recovered if the tool reports a $\texttt{Primitive}_4$, independent of whether original type was a `float`, `int`, or `unsigned int`, whereas a typical reverse engineer would hope for more precision from a "correct" type. Since their output is not more precise than this, we cannot explore their output itself, but note that using the more typical notion of correctness (matching the C type) expected from a reverse engineer, we find that the F1 scores for Ghidra on their binaries are approximately an order of magnitude lower than that reported in their paper. Worse, some other prior work rarely mentions *that* a projection is taken, let alone what the projection is. Without access to the specific projection taken, comparison against reported numbers in various papers via precision/recall/F1 scores is impossible, and uninstructive. Nonetheless, as a brief comparison, if we compute F1 scores (Table 2), TRex appears to do quite well—we are however skeptical of its predictive value since it is trying to measure performance on a goal that we know may be impossible (§2.2).

Instead, we propose a new metric that we believe to be a more accurate measure of output quality, even though it reduces our perceived gain. Importantly, rather than the blunt hammer of "standard" metrics in prior works (which consider, for example, both `int*` and `char` to be *identically* "incorrect" with respect to a `float*`), our proposed metric introduces much-needed nuance. Relying on our own experience as reverse engineers (and anecdotes from others), we define a best-effort prioritized ordering of frustrations (each represented by an indicator variable p_i) with incorrectly inferred aspects of types. Using this ordering, we define our scoring function to be: $f(p_1, p_2, \ldots) = (p_1 \times (1 + p_2 \times (1 + \ldots)))$ for measuring the agreement between the computed and the original type. Figure 6 illustrates our preferred instantiation of this function. For example, we find it incredibly frus-

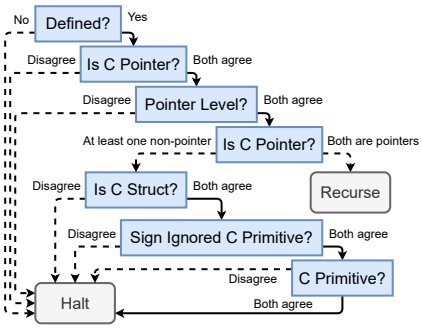


Figure 6: Illustrating Our Scoring Function for Evaluating Type Reconstruction. Traversing solid lines increments the score by 1, while dotted lines do not update the score.

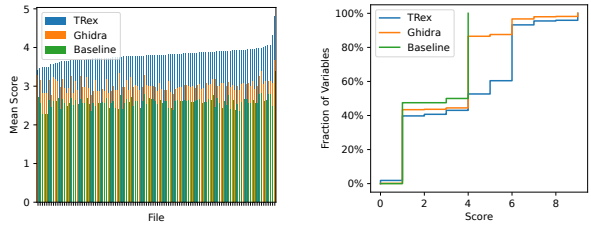
Scoring / Tool	Regular COREUTILS	Regular SPEC	Generous (§5.3) COREUTILS	Generous (§5.3) SPEC
Trivial Baseline	2.583	2.710	2.583	2.710
Ghidra [26]	3.019	3.321	3.023	3.600
ReSym [39] (§5.3)	2.064	2.475	3.675	3.985
TRex	3.802	3.933	3.880	3.944

Table 3: Mean scores achieved by each tool. "Regular" uses the exact scores computed by our scoring function (Figure 6). "Generous" (see §5.3) assigns any unassigned variables to types equivalent to Trivial Baseline's. Higher is better.

trating if a pointer is not detected as such (or vice versa), since it can lead to incorrect reasoning about memory, and thus getting this wrong leads to a large score penalty; the pointer level (e.g., `int*` vs. `int**`) is slightly less crucial, but still highly important; etc. We explicitly distinguish between sign-ignored and regular C primitive agreement, since we deem it more important to get size and behavior (e.g., `double` vs. `int32_t`) correct, compared to sign. To make this concrete, compared to an expected (i.e., original source) type of `uint64_t`, the types `char*`, `struct{...}`, `double`, `int64_t` and `uint64_t` would achieve the scores of 1, 3, 4, 5, and 6 respectively.

We do not claim that our metric is ideal, but merely that our evaluation attempts to capture frustrations from our own experience as reverse engineers (and anecdotes from others). Nevertheless, we also note that our tool consistently outperforms its baselines even when we simulate alternate metrics. Specifically, the relative order of performance of the tools remained unchanged when we evaluated five other random permutations of the p_i's (to simulate reverse engineers with potentially different orderings of priorities). Additionally, to help with future research, we release our evaluation scripts (in addition to our tool, and benchmark reproduction scripts), so that experiments can be re-run with newly proposed metrics.

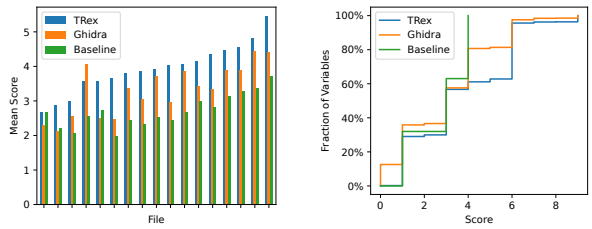
We describe more detail in the upcoming paragraphs (and in §5.3 for the ML-based ReSym, along with the "Generous" evaluation necessary for ReSym), but as seen in Table 3, on average TRex outperforms the others.



(a) Mean scores, sorted by score on TRex. Higher is better.

(b) Cumulative distribution function (CDF) of scores. Lower is better.

Figure 7: Comparing tools across executables in COREUTILS.



(a) Mean scores.

(b) CDF of scores.

Figure 8: Comparing tools across executables in SPEC.

Figure 7a shows the mean score achieved by each tool across all variables in each of the 108 COREUTILS executables (compiled on x86-64), sorted by the scores for TRex. On all executables, TRex consistently achieves better scores than Ghidra, which itself beats the trivial baseline. TRex is able to outperform Ghidra *despite* TRex not yet implementing interprocedural type propagation, which Ghidra uses (in addition to external functions it knows, such as those in libc) to obtain better types. Interestingly, on the three executables where TRex and Ghidra are closest (`od`, `sha384sum`, and `sha512sum`), some preliminary analysis of the root causes point towards conservatively underspecified vector instructions (whose semantics one could improve with straightforward-if-non-trivial engineering effort).

To better understand the scores across the entirety of the dataset, in Figure 7b we also plot the cumulative distribution function (CDF) of scores across all 104,953 variables. We notice that TRex moves into higher scores sooner than Ghidra, which in turn moves into higher scores sooner than the baseline. Note that the graph appears step-like since the scoring function of any particular variable evaluates to an integer.

We plot similar graphs for SPEC, which covers a larger variety of executables (although it is compiled on x86 32-bit, which is not what we have optimized TRex for yet). Figure 8a shows a less stark difference between the tools, compared to the situation in COREUTILS. Nonetheless, TRex outperforms Ghidra on 16 of the 17 executables. The CDF of scores across all 63,600 variables in Figure 8b shows a similar behavior

as with COREUTILS, where TRex moves into higher scores sooner than Ghidra, which in turn moves into higher scores sooner than the baseline.

To help answer RQ 3 (impact of compiler optimizations), we perform a cross-optimization evaluation by compiling COREUTILS with GCC's `-O0` (default, no optimizations), `-O1`, `-O2` and `-O3` optimization levels. The mean scores achieved by TRex on each are 3.499, 3.292, 3.374, and 3.530, respectively, showing that the optimization level has relatively little impact on TRex's performance. In contrast, Ghidra's mean scores are 2.992, 2.236, 2.174, and 2.190, respectively, showing a sharp drop in performance when optimizations are enabled. Note that TRex has no special support for higher optimization levels, other than basic semantics in our IR for the small handful of vector instructions that are introduced at higher optimization levels—many of which we model as the conservative `HAVOC` in our IR (§4.1). We hypothesize that the relative consistency of TRex's score across optimization levels derives from the fact that compiler optimizations preserve semantics, and that structural types successfully capture higher-level semantic behaviors in the code.

Finally, we end this subsection with a brief discussion on performance. For our use cases, exact performance numbers are not particularly relevant, since we can afford to let an analysis run overnight or longer if it produces better output than something that finishes instantly. Nonetheless, we note that TRex takes only a mean time of 56.32s (min=1.60s, max=1385.15s) across the entirety of COREUTILS and SPEC; in contrast, Ghidra takes a mean time of 77.16s (min=7.91s, max=1606.45s) on the same hardware. Some caveats to these numbers include the following: (i) there is no easy way to separate type-reconstruction time from total decompilation time in Ghidra, thus the numbers for Ghidra are over-estimates, and (ii) TRex spends the majority ($\sim$ 85%) of its time on SSA construction; optimizing the current naïve algorithm may significantly improve TRex's performance [9].

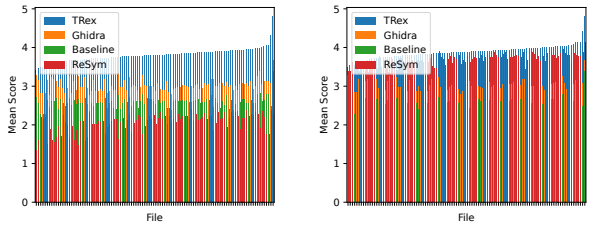
5.3 Comparison to ML-based Type Prediction

While our work focuses on scenarios where we cannot assume the binaries are in distribution (§2.1), to add context and answer RQ 4, we compare against ReSym [39], a learning-based Type Prediction technique. ReSym is the most recent work in the area (Distinguished Paper Award at CCS'24), and its evaluation demonstrates that it dominates prior approaches.

Challenges in reproducibility. We appreciate that the ReSym authors have released a public artifact, including their model weights. Unfortunately, this artifact is incomplete for reproducing their results. When contacted, the authors were unable to share the code for their post-processing pipeline, which is a significant component of their approach. Furthermore, we discovered multiple bugs in the pre-processing code as well as the code that runs the trained model to perform inference. For example, the inference code assumes access to not only the code, but also the expected output types; we patched this to only take reasonable inputs that are accessible from knowing the decompilation. Overall, our changes to get ReSym working account for $\sim$400 LoC added for pre-processing, single-digit LoC modified for inference, and $\sim$500 LoC added to implement basic post-processing (that takes the possibly-conflicting free-form output and produces C-like types). These were not simple updates, and required $\sim$2 person-weeks just to get ReSym running. We also note that despite the claim [39, §5.2] of 3.4s per-binary execution time (on COREUTILS), even on equivalent hardware (four A100 GPUs), the mean inference execution time we observed was 656.81s (min=85.61s, max=1978.29s). We note all this not to criticize ReSym, but to highlight the context within which the comparison to their technique is possible.

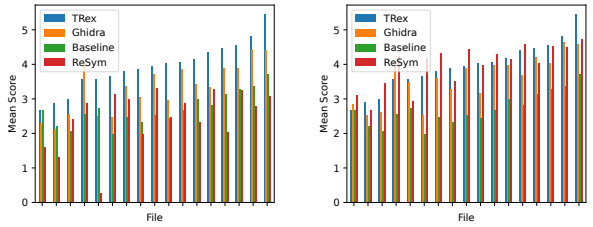
Qualitative comparison. ReSym, unlike many deductive approaches which take machine code (or equivalent low-level code) as input, takes decompilation output (i.e., C-like high-level code) as its input. This means that its output can be impacted by other decisions made by the decompiler (which itself includes a deductive type inference pipeline), for better or worse. For example, if the decompiler has a large function and type-signature database, which it applies to preemptively match types (e.g., knowing that `fopen` returns a `FILE*`), or if the user already marks some names or types from having recognized them through their experience and intuition, then ReSym is able to successfully exploit such data, since it reinforces its own learned beliefs. Unfortunately, using high-level decompilation as input can impact output negatively too, where seemingly innocuous changes can lead it astray.

Focusing on the singly-linked list example from §5.1, ReSym's outputs range all the way from perfect to problematic, simply by varying the input parameter name. As context, it is not uncommon in the reverse engineering process to rename variables (away from the automatically-generated `param_1`/`var_a2c4`, to more human-readable, but out-of-domain, names—`apple`/`banana`/etc.); such α-conversion is semantics-preserving and aids in improved readability until finalized names are picked. To explore this, we consider 100 randomly generated renamings of the identifiers in the exact same code, for each of the field orderings (recursion in the first/second field is *easier*/*harder* respectively, see §5.1), and look at ReSym's outputs. On 32% (*easier*) and 2% (*harder*) of these renamings, ReSym did produce output equivalent to the expected type—`struct t{t*;int}`/`struct t{int;t*}`. However, 30% (*easier*) and 62% (*harder*) of outputs do not have *any* reasonable equivalent C type (for example, nearest equivalent would be the clearly-broken `struct t{t;t}`—note the absence of any pointers or differentiation between the fields). The remaining (38% and 36%, respectively) were valid C types, but incorrect in differing ways. Importantly, by design, ReSym additionally takes the field



(a) Regular scoring (b) Generous scoring

Figure 9: Mean scores on executables in COREUTILS, sorted by score on TRex. Higher is better.



(a) Regular scoring (b) Generous scoring

Figure 10: Mean scores on executables in SPEC, sorted by score on TRex. Higher is better.

offsets of the `struct` as input (for which we provide correct values), thus it cannot get those incorrect under any situation; if field offsets are not provided, ReSym is unable to provide details of such a `struct`. In contrast, TRex, by design, cannot be impacted by semantics-preserving changes that reverse engineers commonly perform (again, for better or worse), and does not require prior knowledge of correct field offsets. We also note that TRex *deterministically* produces the correct results for both orderings of fields.

Quantitative comparison. For the quantitative evaluation, in the interest of fairness, we leave all variables with the names as auto-generated by the decompiler. On standard metrics, ReSym's F1 scores are included in Table 2, where it does indeed perform better than Ghidra; however, TRex outperforms it. Further discussion below focuses on the more nuanced metric we have introduced in §5.2, and the results of which are summarized in Table 3. Figure 9a shows the initial results on COREUTILS, where ReSym appears to perform worse than even the baseline. We did some investigating and determined that this was due to a failure in the "alignment" process between low-level variable references (which talks about actual memory regions) and decompilation variables (pseudo memory regions, which may or may not correspond to a real memory region). To account for this, we additionally include Figure 9b (corresponding to "Generous" in Table 3), where we mark any variable without an assigned type as equivalent to the baseline's output. We stress that while this is optimistic and generous to all tools, this is *especially* generous to ReSym, since it affects multiple orders of magnitude more variables for ReSym. In this generous comparison, we note that TRex and ReSym are put into a similar range, where each beats the other on some binaries (although TRex still achieves a higher average score). We additionally note that, as seen with our qualitative evaluation above, it is hard to know when ReSym's output can be trusted. For completeness, we include a similar quantitative comparison on SPEC, in Figure 10, briefly noting that the results do not change significantly (TRex beats ReSym on Regular scoring, while each beat the other on some binaries on Generous scoring).

Takeaway. The main takeaway from this subsection's evaluation is not that any one approach is strictly superior to the other, but that there are pros and cons of each. Some combination of the (largely orthogonal) techniques of Type Prediction and Type Reconstruction might be interesting for future exploration.

6 Related Work

Research on decompilation is vast and decades long [8], focusing on different aspects like disassembly [4,10,19], control flow recovery [3, 31, 41], function identification [1, 2], etc. Here, we focus in particular on the sizable literature related to the problem of computing high-level types from binaries.

Caballero and Lin [5] conducted an authoritative survey of 38 prior papers in this area up to 2015, observing that existing tools vary widely both in their approaches and the set of types that they can compute. Due to space, we discuss only more recent and/or highly relevant projects and refer interested readers to the survey for a more comprehensive overview.

On the deductive side, TIE (2011) by Lee et al. [18] was an early tool that popularized the constraint-solving approach, which TRex also uses. At a high level, TIE collects usage constraints about identified variables from a binary (static TIE) or a trace (dynamic TIE) and outputs a solution that satisfies all collected constraints. Compared to TIE, TRex is designed to support a more expressive type system, notably including recursive types due to their prevalence in real-world programs.

Retypd (2016) by Noonan et al. [25] is a recent prior work that also uses the constraint-solving approach. As far as we know, the type system supported by Retypd is the most expressive among the tools in the literature, and it even exceeds that of TRex in one aspect: Retypd supports polymorphic types, e.g., $\forall \tau : \texttt{size_t} \rightarrow \tau*$ (§3.1). Regrettably, it is infeasible to know how Retypd actually performs in practice (§5).

OSPREY (2021) by Zhang et al. [42] is one of the latest tools based on constraint solving. Compared to prior tools, OSPREY introduced an interesting and powerful extension—the use of probabilistic constraints. While this enables OSPREY to potentially cope with the inherent uncertainty in variables and types due to information lost during compila-

tion, the design of OSPREY was tuned for Type Recovery. In particular, while OSPREY uses probabilities to model uncertainty and emits nominal types deemed most likely in the final phase, TRex uses structural types to preserve complete information about program behavior until the final phases.

Unfortunately, we cannot obtain any of the above tools (§5) and so we cannot directly evaluate them against TRex. On the other hand, during the final preparation of this manuscript, a new and concurrent work appeared in the literature. Specifically, BinSub (2024) by Smith [33] focuses on replicating Retypd in a simpler, more efficient manner by leveraging the framework of algebraic subtyping. We are delighted to learn that the author of BinSub has made both the system and the dataset publicly available. Naturally, we look forward to seeing future evaluations with both BinSub and TRex.

While our work focuses on scenarios where dependability of output is crucial, there are definitely scenarios where one either has binaries "in the distribution" or can forgo explainability. Thus, on the learning side, there has been a surge of new proposals in the area, leveraging the latest advances in machine learning. Remarkably, StateFormer (2021) by Pei et al. [27] has demonstrated the power of a clever use of neural networks for this domain. It starts with a fully self-supervised pre-training step where the model is taught to statistically approximate the operational semantics of instructions in both forward and backward directions using a very small number of execution states. Then, it fine-tunes the model to use the learned operational semantics to infer types. Compared to TRex, StateFormer is able to output only a fixed set of 35 types. However, a key strength of StateFormer's approach is its explicit learning of instruction semantics, which can be extremely valuable when new instructions are introduced by CPU vendors or when dealing with binaries for a new ISA.

DIRTY (2022) by Chen et al. [7] features a transformer neural network (trained on a large corpus of open-source C projects mined from GitHub) that recommends variable names and types as a post-processor of decompilation outputs. In total, DIRTY supports the 48,888 types that were encountered in its corpus. While the set of types is much larger than that of StateFormer, it is still limited to previously-seen types when compared to TRex. As such, Chen et al. suggest the use of Byte Pair Encoding [32] to extend DIRTY to support computing previously-unseen `struct` types as future work.

ReSym (2024) by Xie et al. [39] uses fine-tuned large language models with a Prolog-based reasoning system for variable and type recovery. While experiments show it can outperform the prior work in name, type, and structure recovery, the authors also reported an approximate halving of accuracy when it is run on functions outside their training set. In addition, our evaluation (§5.3) also shows that the output of ReSym is hard to depend upon.

7 Concluding Remarks

Lack of high-quality source-level types has plagued decompiled code for decades despite advances in the art and science of decompilation. In this paper, we presented TRex, a new tool that performs automated deductive type inference for binaries using a new perspective, *Type Reconstruction*. This perspective accounts for the inherent impossibility of recovering lost source types, and guides us to perform analyses using *structural types* that capture the behavior of the program. Since human reverse engineers are more familiar with C-like output from decompilers, we then round these precise-but-verbose structural types to C-like nominal types. Overall, TRex shows a noticeable improvement in the quality of output types compared with Ghidra, an actively developed state-of-the-art decompiler used by practitioners. We also document insights derived while building TRex, such as showing that type rounding is an NP-hard problem and yet our greedy algorithm works reasonably well in practice.

More broadly, we hypothesize that the field of decompilation's focus on the impossible goal of *recovering* lost source code has contributed to the increasing complexity of tools and techniques in the academic literature. To the extent that this complexity "overfits" to a particular compilation technique or even a particular compiler, this may also explain why these techniques are not used in practical decompilers. In contrast, we show that by shifting our focus away from the impossible and instead towards what reverse engineers desire—output that captures behavior, we can design and implement simpler and more elegant decompilation tools that help reverse engineers with their actual requirements.

Our work does lead to some natural questions for future work. Inter-procedural type propagation appears to be a significant contributor to Ghidra's output; at present, TRex does not implement this, but it would be interesting to understand how much this factors into quality of output in a Type Reconstruction setting in contrast to the older Type Recovery setting. Furthermore, TRex focuses on high-quality automated *initial* types; and it does not currently take types from users as input. It would be interesting to explore Type Reconstruction with additional input types. Along a different direction, TRex could be augmented to propose better names for the rounded C-like aggregate types, rather than the current auto-generated identifiers, possibly using recent advances in machine learning; indeed, this might be a fertile ground for learning-based Type Prediction approaches to connect with Type Reconstruction, with the former contributing its strength of human readability on in-distribution binaries and the latter its guarantees of reasonable output for all binaries. Finally, we speculate that our shift in perspective from Type Recovery to Type Reconstruction might more broadly be applicable to other components of the decompilation pipeline, providing more avenues for exploration.

Acknowledgments

We thank the anonymous shepherd and reviewers for their helpful feedback. This work was supported in part by the Office of Naval Research (under ONR Award No. N00014-17-1-2892), and a gift from VMware. While at Carnegie Mellon University, Jay Bosamiya was also supported by the CyLab Presidential Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the organizations that supported this work.

Ethical Considerations

To the best of our knowledge, the research presented here does not present any significant negative ethical concerns. Our work does not involve human subjects or personal data, focusing solely on automating the reconstruction of type information from binary code. This work does not introduce any new risks of harm to individuals or systems, nor does it facilitate new attacks or privacy violations. We mention the Menlo Report here only to note that it has no direct relation to the work presented in this paper: there are no human subjects, this work introduces no additional harms to human subjects, selection of subjects is vacuously fair because there are no human subjects, and all methods and results as part of this research are legal to the best of our knowledge.

Open-Science

As described in §1, as a step towards reversing the unfortunate tendency of closed-source tools and non-reproducible benchmarks, we open-source our tool, evaluation framework, and reproducible scripts for the benchmarks. These can be found at `https://github.com/secure-foundations/{trex,trex-usenix25}` (archived at `https://doi.org/10.5281/zenodo.15611994`).

References

[1] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189, 2017.

[2] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, 2014.

[3] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! there is no need to DREAM of c: A Compiler-Aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, Philadelphia, PA, August 2024. USENIX Association.

[4] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[5] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, May 2016.

[6] Ligeng Chen, Zhongling He, and Bing Mao. CATI: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020.

[7] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA, August 2022. USENIX Association.

[8] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35. ACM Press, January 1989.

[10] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[11] Gernot Heiser. Systems Benchmarking Crimes. `https://gernot-heiser.org/benchmarking-crimes.html#subset`. Archived: `https://archive.is/BgUCl#subset`.

[12] GrammaTech. Type inference (in the style of retypd). `https://github.com/GrammaTech/retypd/blob/f8dd231478c3e1722d0d160c3cf99c628a25/reference/type-recovery.rst`. Archived: `https://archive.is/GbsUB`, July 2021.

[13] Hex-Rays. Hex-Rays End User License Agreement. https://hex-rays.com/eula.pdf. Archived: https://web.archive.org/web/20240810083425/https://hex-rays.com/eula.pdf.

[14] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[15] ISO. ISO C standard 1999. https://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, 1999. Section 6.7.2.1, Item 16, Page 103.

[16] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[17] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[18] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2011.

[19] Kaiyuan Li, Maverick Woo, and Limin Jia. On the generation of disassembly ground truth and the evaluation of disassemblers. In Kevin W. Hamlen and Long Lu, editors, *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation, FEAST2020, Virtual Event, USA, 13 November 2020*, pages 9–14. ACM, 2020.

[20] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the Network and Distributed System Security Symposium 2010*, pages 1–18. The Internet Society, 2010.

[21] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. TypeMiner: Recovering types in binary programs using machine learning. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 288–308, Cham, 2019. Springer International Publishing.

[22] Matthew Maurer. *Holmes: Binary Analysis Integration Through Datalog*. PhD thesis, Carnegie Mellon University, Oct 2018.

[23] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[24] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, page 208–223, Berlin, Heidelberg, 1999. Springer-Verlag.

[25] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 27–41. Association for Computing Machinery, Jun 2016.

[26] National Security Agency (NSA). Ghidra. https://www.nsa.gov/ghidra.

[27] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–702. ACM, 8 2021.

[28] Python 3. Glossary: Duck typing. https://docs.python.org/3/glossary.html#term-duck-typing. Archived: https://archive.is/qvZGc.

[29] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[30] Hex-Rays SA. Hex-Rays Decompiler. https://hex-rays.com/decompiler/.

[31] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native ×86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Conference on Security*, pages 353–368, USA, 2013. USENIX Association.

[32] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725. Association for Computational Linguistics, 2016.

[33] Ian Smith. Binsub: The simple essence of polymorphic type inference for machine code. In *Proceedings of the 31st International Static Analysis Symposium*, page 425–450, Berlin, Heidelberg, 2024. Springer-Verlag.

[34] SPEC. SPEC CPU2006. https://www.spec.org/cpu2006/.

[35] Trail of Bits. Binary type inference in Ghidra. https://blog.trailofbits.com/2024/02/07/binary-type-inference-in-ghidra/. Archived: https://archive.is/VPwgD, February 2024.

[36] Vector 35. Binary Ninja. https://binary.ninja/.

[37] Vector 35. Decompiler explorer. https://dogbolt.org/.

[38] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147. Association for Computing Machinery, 3 2020.

[39] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to recover variable and data structure symbols from stripped binaries. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 4554–4568, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pages 158–177. IEEE, 5 2016.

[41] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[42] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021.

Appendix A Primitive Analysis Constraints

Here, we list the constraints from TRex's primitive analysis phase (§3.3.2) in the form of inference rules $\frac{P_1 \quad P_2 \quad \dots}{Q}$, which means that the presence of the antecedents P_i's leads one to learn the consequent Q. We use the trivial consequent ($\top$) to imply that nothing new is learnt from the antecedents. An equality constraint ($\tau_u = \tau_v$) in the consequent refers to a full recursive merge (§4.3), which implicitly allows the introduction of recursive types. Any italicized symbol (such as o, $i_1, \dots$) is a meta-variable, universally quantified over its natural domain. The type of an SSA variable v is represented as τ_v. Most instructions are written in the form $o \leftarrow \text{Foo}(i_1, \dots)$ to mean that the SSA-variable o is assigned the result of executing the instruction `Foo`. Where an instruction has no output (e.g., `Nop`), the left-arrow symbol is elided. The shorthand $|v|$ is used to refer to the size (in bytes) of the variable v.

$$\frac{o \leftarrow \phi(i_1, i_2, \dots)}{\tau_o = \tau_{i_1} \wedge \tau_o = \tau_{i_2} \wedge \cdots} \qquad \frac{\text{GVN}(v_1) = \text{GVN}(v_2)}{\tau_{v_1} = \tau_{v_2}}$$

$$\frac{\text{Nop}}{\top} \qquad \frac{\text{ProcException}}{\top} \qquad \frac{o \leftarrow \text{Havoc}(i_1, i_2, \dots)}{\top}$$

$$\frac{\text{FunctionStart}}{\top} \qquad \frac{\text{FunctionEnd}}{\top}$$

$$\frac{\text{Call}(f)}{\top} \qquad \frac{\text{Return}(p)}{\text{Pointer}(\tau_p) \wedge \text{Code}(\text{Pointee}(\tau_p))}$$

$$\frac{\text{Branch}(t) \quad t : \text{variable}}{\text{Pointer}(\tau_t) \wedge \text{Code}(\text{Pointee}(\tau_t))} \qquad \frac{\text{Branch}(t) \quad t : \text{constant}}{\top}$$

$$\frac{\text{CondBranch}(c, t)}{\text{ZeroComparable}(c) \wedge \text{Supports}(\tau_c, \text{IntEq}_{|c|})}$$

$$\frac{o \leftarrow \text{Copy}(i)}{\tau_o = \tau_i \wedge \text{Supports}(\tau_o, \text{Copy}_{|o|})} \qquad \frac{o \leftarrow \text{Piece}(i_1, i_2)}{\top}$$

$$\frac{o \leftarrow \text{SubPiece}(i, s) \quad s \neq 0}{\top} \qquad \frac{o \leftarrow \text{SubPiece}(i, 0)}{\tau_o = \tau_i}$$

$$\frac{o \leftarrow \text{Load}(i)}{\text{Supports}(\tau_o, \text{Copy}_{|o|}) \wedge \text{Pointer}(\tau_i) \wedge \tau_o = \text{Pointee}(\tau_i)}$$

$$\frac{\text{Store}(p, v)}{\text{Supports}(\tau_v, \text{Copy}_{|v|}) \wedge \text{Pointer}(\tau_p) \wedge \tau_v = \text{Pointee}(\tau_p)}$$

$$\frac{o \leftarrow F(i) \quad F \in \{\text{BoolNeg}, \text{IntNeg}, \text{FloatNeg}, \text{OnesComp}, \dots\}}{\text{Supports}(\tau_o, F_{|o|}) \wedge \text{Supports}(\tau_i, F_{|o|})}$$

$$\frac{o \leftarrow F(i) \quad F \in \{\text{Int2Float}, \text{Float2Int}, \dots\}}{\text{Supports}(\tau_o, \text{Out}(F)_{|o|}) \wedge \text{Supports}(\tau_i, \text{In}(F)_{|i|})}$$

$$\frac{o \leftarrow F(i_1, i_2) \quad F \in \{\text{BoolOr}, \text{IntMul}, \text{FloatDiv}, \dots\}}{\forall v \in \{o, i_1, i_2\}.\ \text{Supports}(\tau_v, F_{|o|})}$$

$$\frac{o \leftarrow F(i_1, i_2) \quad F \in \{\text{IntEq}, \text{IntLt}, \text{FloatEq}, \text{IntCarry}, \dots\}}{\text{Bool}(o) \wedge \tau_{i_1} = \tau_{i_2} \wedge \text{Supports}(\tau_{i_1}, F_{|i_1|})}$$

$$\frac{o \leftarrow F(i_1, i_2) \quad F \in \{\text{IntShl}, \text{IntUShR}, \text{IntLShR}\}}{\text{Supports}(\tau_o, F_{|o|}) \wedge \text{Supports}(\tau_{i_1}, F_{|i_1|}) \wedge \text{Supports}(\tau_{i_2}, \text{ShAmt})}$$

$$\frac{o \leftarrow F(i) \quad F \in \{IntZExt, IntSExt\}}{\text{Supports}(\tau_o, \text{Tgt}(F)_{|o|}) \wedge \text{Supports}(\tau_i, \text{Src}(F)_{|i|})}$$

$$\frac{o \leftarrow F(i_1, i_2) \quad F : \text{packed/lower vector op } f \text{ over size } s}{\forall v \in \{o, i_1, i_2\}.\ \text{Supports}(\tau_v, f_s)}$$

$$\frac{o \leftarrow F(i_1, i_2) \quad F : \text{upper vector op } f \text{ over size } s}{\top}$$

Appendix B Colocation Analysis Constraints

Here, we list the constraints from TRex's colocation analysis phase (§3.3.3) in the form of inference rules (described in Appendix A).

$$\frac{o \leftarrow \text{Load}(i)}{\zeta(i, o) \wedge \text{ConstOffDeref}_0(o, i)}$$

$$\frac{\text{Store}(p, v)}{\zeta(p, v) \wedge \text{ConstOffDeref}_0(v, p)}$$

$$\frac{o \leftarrow F(i_1, i_2, \ldots)}{o \leftarrow F(i_1, i_2, \ldots)}$$

$$\frac{o \leftarrow \phi(i_1, i_2, \ldots) \quad i_1 \leftarrow F_1(I_{11}, I_{12}, \ldots) \quad i_2 \leftarrow F_2(I_{21}, I_{22}, \ldots) \quad \ldots}{o \leftarrow F_1(I_{11}, I_{12}, \ldots) \wedge o \leftarrow F_2(I_{21}, I_{22}, \ldots) \wedge \ldots}$$

$$\frac{\text{IsConst}(v)}{\xi(v)}$$

$$\frac{o \leftarrow F(i_1, i_2, \ldots) \quad \xi(i_1) \quad \xi(i_2) \quad \ldots}{\xi(o)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_1, i_2) \quad \xi(i_1) \quad \text{Pointer}(i_2)}{\text{ConstOffDeref}_{\text{ConstVal}(i_1)}(v, i_2) \wedge \zeta(i_2, v)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_1, i_2) \quad \xi(i_2) \quad \text{Pointer}(i_1)}{\text{ConstOffDeref}_{\text{ConstVal}(i_2)}(v, i_1) \wedge \zeta(i_1, v)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntSub}(i_1, i_2) \quad \xi(i_2) \quad \text{Pointer}(i_1)}{\text{ConstOffDeref}_{-\text{ConstVal}(i_2)}(v, i_1) \wedge \zeta(i_1, v)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_1, i_2) \quad \xi(i_1) \quad \neg\text{Pointer}(i_2)}{\text{NonConstOffDeref}_{i_2}(v, \text{ConstVal}(i_1))}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_1, i_2) \quad \xi(i_2) \quad \neg\text{Pointer}(i_1)}{\text{NonConstOffDeref}_{i_1}(v, \text{ConstVal}(i_2))}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_1, i_2) \quad \neg\xi(i_1) \quad \neg\xi(i_2) \quad \text{Pointer}(i_1)}{\text{NonConstOffDeref}_{i_2}(v, i_1) \wedge \zeta(i_1, v)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntAdd}(i_2, i_1) \quad \neg\xi(i_2) \quad \neg\xi(i_1) \quad \text{Pointer}(i_2)}{\text{NonConstOffDeref}_{i_1}(v, i_2) \wedge \zeta(i_2, v)}$$

$$\frac{\zeta(p, v) \quad p \leftarrow \text{IntSub}(i_1, i_2) \quad \neg\xi(i_1) \quad \neg\xi(i_2) \quad \text{Pointer}(i_1)}{\text{NonConstOffDeref}_{-i_2}(v, i_1) \wedge \zeta(i_1, v)}$$

$$\frac{\text{ConstOffDeref}_o(v, p) \quad o \neq 0}{\text{BaseVar}(p)}$$

$$\frac{\text{NonConstOffDeref}_o(v, p)}{\text{BaseVar}(p)}$$

Appendix C Algorithm for Type Rounding

Algorithm 1 Greedy algorithm for Type Rounding (§3.3.5).

```
function RoundUp(type, primitives)
    for all p_i ∈ primitives do
        A_i ← Components(p_i)
                ▷ indicator vector for p_i's components
    end for
    A ← [A_1; A_2; ...]
    b ← Components(type)
    x ← column vector of all 1s
    repeat
        for all i ∈ indices of x where x_i = 1 do
            x' ← x; x'_i ← 0
            if Ax' ≥ b then
                cost_i ← number of 1s in row A_i
            else
                cost_i ← ∞
            end if
        end for
        if all cost_i = ∞ then
            break
        end if
        j ← arg max_i cost_i
                ▷ skipping any cost_i = ∞
        x_j ← 0
    until no further reduction is possible
    return ⋃{p_i | x_i = 1}
end function
```