Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring

Abstract

There are many security tools and techniques for finding bugs, but many of them assume access to source code. We propose leveraging *decompilation*, the study of recovering abstractions from binary code, as a technique for applying existing source-based tools and techniques to binary programs. A decompiler must have two properties to be used for security: it must (1) be correct (is the output functionally equivalent to the binary code?), and (2) recover abstractions (e.g., the output should utilize while loops commands and not `gotos`).

Previous work in control-flow structuring, which is a building block for decompilers, is insufficient for use in security, because it does not provide correctness and effective abstraction recovery. Specifically, we discovered that existing structuring algorithms are not *semantics-preserving*, which means that they cannot safely be used for decompilation without modification. Second, we found that existing structural algorithms miss opportunities for recovering control flow structure on (partially) unstructured code. We propose a new structuring algorithm that addresses these problems.

We evaluate our decompiler, Phoenix, which features our new structuring algorithm, on a set of 107 real world programs from GNU `coreutils`. Our evaluation is an order of magnitude larger than previous studies of end-to-end decompilers. We show that our decompiler outperforms the *de facto* industry standard decompiler Hex-Rays in correctness by 114%, recovers 30× more control-flow structure than existing structuring algorithms in the literature, and 28% more than Hex-Rays.

1 Introduction

Security analyses are faster, easier, and more scalable when performed on source code rather than binary code. For example, source code based taint checkers have a negligible 0.65% runtime overhead [10], whereas the fastest taint checkers for binary code incur a 150% overhead [7]. Many security analyses described in the literature also assume access to source code. For instance, there are numerous static vulnerability finding tools for source code, e.g., KINT [39], RICH [8], and Coverity [5], but equivalent binary-only tools are sparse.

Unfortunately, access to source code is not a reasonable assumption in many security scenarios. Common counter-examples include analyzing commercial software for vulnerabilities and reverse engineering malware. Such scenarios preclude direct use of source-based techniques. The traditional approach in security has been to apply low-level binary analysis that avoids the need for source-level abstractions such as types and functions. Unfortunately, this low-level reasoning is what makes binary analysis more complicated and less scalable than its source-based counterparts [4, 6, 23, 24].

We argue that *decompilation* is an attractive alternative to traditional low-level binary-only techniques since it enables us to leverage the wealth of existing source-based analyses and tools. Decompilation is the study of recovering a program's source code given a program's binary. Decompilation can avoid some of the pitfalls of traditional binary analysis by explicitly recovering high-level abstractions and source, which can then be fed to a source-based analysis. Potential source-based analyses include existing vulnerability scanners [27], taint engines [10], and bug finders [5].

Decompilation is also beneficial for security practitioners [38]. Practitioners often reverse engineer binary code in order to study vulnerabilities fixed in patches, understand proprietary file formats, and determine the exploitability of crashing inputs. Each of these tasks becomes easier when given access to source code.

Unfortunately, current techniques for decompilation are insufficient for security purposes. A decompiler must have two properties to be used for security: it must (1) be correct, and (2) recover abstractions. An incorrect decompiler is harmful because it may introduce errors into an automated analysis, or confound a human analyst. Recovering abstractions is equally important since the difficulty of binary analysis stems from the lack of quality abstractions.

Previous work in decompilation has focused on producing readable summaries of low-level code for human reverse engineering, with little attention paid to whether the decompiler analysis itself was correct. For example, Cifuentes et al.'s pioneering work [11], as well as subsequent work [9, 12, 14, 38, 40], focused on how much smaller the output C code was compared to the input assembly, or on a subjective readability metric.

In this paper, we argue that source can be recovered in a principled fashion. As a result, security analyses can better take advantage of existing source-based techniques and tools both in research and practice. Security practitioners can also recover correct, high-level source code, which is easier to reverse engineer. In particular, we propose techniques for building a correct decompiler that effectively

recovers abstractions. We implement our techniques in a decompiler called *Phoenix*, and measure our results with respect to correctness and high-level abstraction recovery.

Challenges Source code reconstruction requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Previous work such as TIE [28], REWARDS [29], and Howard [37] have addressed recovering data types. In this paper, the main challenges we address are: (1) recovering control flow abstractions, (2) demonstrating accurate end-to-end decompilation and abstraction recovery on a large number of programs, and (3) scientifically measuring our results in terms of correctness and abstraction recovery. No previous work achieves all three goals.

In particular, previous work has proposed mechanisms for recovering control flow based on structural analysis and its predecessors [19, 22, 38], which originate in compiler literature. We show previous techniques based on structural analysis are insufficient for decompilation because (1) they do not feature a semantics-preservation property that is necessary to be safely used for decompilation, and (2) they miss opportunities for recovering control flow structure. Previous decompilers have only tested their algorithms on a few small programs, making it difficult to fairly evaluate their effectiveness. Finally, and perhaps surprisingly, correctness has never been proposed as an end-to-end metric for decompilers, which may explain why decompilers have not been used for binary analysis in the past. Previous work on decompilation has instead focused on readability, or how easy it is for humans to understand the output.

1.1 The Phoenix Structural Analysis Algorithm

These problems have motivated us to create our own control flow structuring algorithm for Phoenix. Our algorithm is based on structural analysis, but avoids the problems we identified with earlier work. In particular, we identify a new property that structural analysis algorithms should have to be safely used for decompilation, called *semantics-preservation*. We also propose *iterative refinement* as a strategy for recovering additional structure.

Semantics Preservation Structural analysis [31, p. 203] is a control flow structuring algorithm that was originally invented to help accelerate dataflow analysis. Later, decompiler researchers adapted these algorithms for use in decompilers to build a control-flow structure or skeleton of the program, for instance:

```
while (...) { if (...) {...} }.
```

Unfortunately, vanilla structural analysis does not always return a structure that accurately reflects the semantics of the program, because it was created for a different purpose. We propose that structuring algorithms must be *semantics-preserving* to be safely used in decompilers. A structuring algorithm is semantics-preserving if it always transforms the input program to a functionally equivalent program representation. Surprisingly, we found that the standard structural analysis algorithm is *not* semantics-preserving. We demonstrate that fixing these problems increases the number of utilities Phoenix correctly decompiles by 30% (see §4).

Iterative Refinement When structural analysis algorithms encounter unstructured code, they stop recovering structure in that part of the program. Our algorithm instead uses iterative refinement. The basic idea is to select an edge from the graph that is preventing the algorithm from making progress, and represent it using a `goto` in the decompiled output. This may seem counter-intuitive, since more `goto`s implies less structure recovered. However, by removing the edge from the graph the algorithm can make more progress, and recover more structure. We also show how refinement enables the recovery of switch structures. In our evaluation, we demonstrate that iterative refinement recovers 30× more structure than structural analysis algorithms that do not employ iterative refinement (see §4). Unfortunately, even the most recent algorithm in the literature [19] does not employ refinement, and thus can stop making progress on problematic program sections.

Contributions:

1. We propose a new structural analysis algorithm that addresses two shortcomings of existing structural analysis algorithms: (1) they can cause incorrect decompilation, and (2) they miss opportunities to recover control flow structure. Our algorithm uses *iterative refinement* to recover additional structure, including switches. We also identify a new property, *semantics-preservation*, that control flow structuring algorithms must have to be safely used in decompilers. We implement and test our algorithm in our new end-to-end binary-to-C decompiler, Phoenix.
2. We demonstrate that our proposed structural analysis algorithm recovers 30× more control-flow structure than existing research in the literature [19, 31, 35], and 28% more than the *de facto* industry standard decompiler Hex-Rays [22]. Our evaluation uses the 107 programs in GNU `coreutils` as test cases, and is an order of magnitude larger than any other end-to-end decompiler evaluation to date.
3. We propose *correctness* (is the output functionally equivalent to the binary code?) as a new metric for evaluating decompilers. Although previous work has measured the correctness of individual decompiler components (e.g., type recovery [28] and structure recovery [19]), surprisingly the correctness of a decompiler as a whole has never been measured. We show in our evaluation that Phoenix successfully

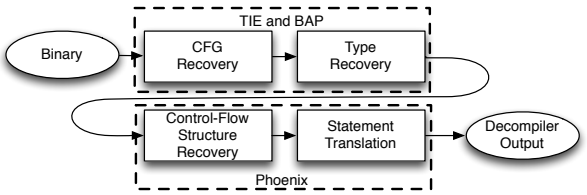


Figure 1: Decompilation flow of Phoenix. Phoenix contains new implementations for control flow recovery and statement translation.

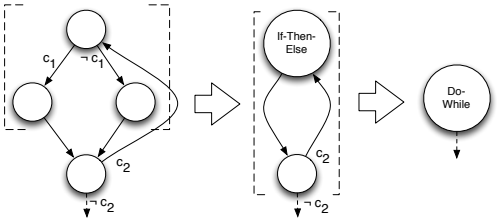


Figure 2: Example of how structural analysis would collapse nodes, from left to right.

decompiled over 2× as many programs that pass the coreutils unit tests as Hex-Rays.

2 Overview

Any end-to-end decompiler such as Phoenix is necessarily a complex project. This section aims to give a high-level description of Phoenix. We will start by reviewing several background concepts and then present an overview of each of the four stages shown in Figure 1. In §3, we will expand on our novel structural analysis algorithm that is the third stage.

2.1 Background

Control Flow Analysis A *control flow graph* (CFG) of a program P is a directed graph $G = (N, E, n_s, n_e)$. The node set N contains basic blocks of program statements in P. Each basic block must have exactly one entrance at the beginning and one exit at the end. Thus, each time the first instruction of a basic block is executed, the remaining instructions must also be executed in order. The nodes $n_s \in N$ and $n_e \in N$ represent the entrance and the exit basic blocks of P respectively. An edge (n_i, n_j) exists in the edge set E if $n_i \in N$ *may* transfer control to $n_j \in N$. Each edge (n_i, n_j) has a label ℓ that specifies the logical predicate that must be satisfied for n_i to transfer control to n_j.

Domination is a key concept in control flow analysis. Let n be any node. A node d dominates n, denoted d **dom** n, iff every path in G from n_s to n includes d. Furthermore, every node dominates itself. A node p post-dominates n, denoted p **pdom** n, iff every path in G from n to n_e includes p. The immediate dominator of n is the unique node d that strictly dominates n (i.e., d **dom** n and $d \neq n$) but does not strictly dominate any other node that strictly dominates n. The immediate post-dominator of n is defined similarly.

Loops are defined through domination. An edge (s, d) is a *back edge* iff d **dom** s. Each back edge (s, d) defines a *natural loop*, whose header is d. The natural loop of a back edge (s, d) is the union of d and the set of nodes that can reach s without going through d.

Structural Analysis *Structural analysis* is a control flow structuring algorithm for recovering high-level control flow structure such as if-then-else constructs and loops. Intriguingly, such an algorithm has uses in both compilation (during optimization) and decompilation (to recover abstractions). At a high level, structural analysis matches a set of region *schemas* over the CFG by repeatedly visiting its nodes in post-order. Each schema describes the shape of a high-level control structure such as if-then-else. When a match is found, all nodes matched by the schema are *collapsed* or *reduced* into a single node that represents the schema matched. For instance, Figure 2 shows the progression of structural analysis on a simple example from left to right, assuming that the topmost node is being visited. In the initial (leftmost) graph, the top three nodes match the shape of an if-then-else. Structural analysis therefore reduces these nodes into a single node that is explicitly labeled as an if-then-else region in the middle graph. This graph is then further reduced into a do-while loop. A decompiler would use this sequence of reductions and infer the control flow structure: `do { if (c1) then {...} else {...} } while (c2)`.

Once no further matches can be found, structural analysis starts reducing acyclic and cyclic subgraphs into *proper regions* and *improper regions*, respectively. Intuitively, both of these regions indicate that no high-level structure can be identified in that subgraph and thus `goto` statements will be emitted to encode the control flow. A key topic of this paper is how to build a modern structural analysis algorithm that can *refine* such regions so that more high-level structure can be recovered.

SESS Analysis and Tail Regions Vanilla structural analysis is limited to identifying regions with a single entrance and single exit. However, common C constructs such as `break` and `continue` can allow a loop to have multiple exits. For instance, the loop

```
while (...) { if (...) { foo; break; } }
```

can exit from the loop guard of the `break` statement. Engel et al. [19] proposed the SESS (single exit single successor) analysis to identify regions such as this that have multiple exits which share a unique successor. Such exits can be converted into a *tail region* that represents the equivalent control flow construct. In the above example, `foo` would be reduced to a `break` tail region. Without tail regions, structural analysis stops making progress when reasoning about loops containing statements such as `break`. We

edge	::	*exp*
vertex	::=	*stmt**
stmt	::=	*var* := *exp* | `assert` *exp* | `addr` *address*
exp	::=	`load`(*exp*, *exp*, *exp*, τ_{reg})
		| `store`(*exp*, *exp*, *exp*, *exp*, τ_{reg})
		| *exp* `op` *exp* | *var* | `lab`(*string*) | *integer*
		| `cast`(*cast_kind*, τ_{reg}, *exp*)

Table 1: An abbreviated syntax of the BAP IL used to label control flow graph vertices and edges.

found that SESS recovers more structure than vanilla structural analysis. However, in our implementation, structural analysis would often stop making progress *before* SESS analysis was able to produce a tail region[1]. Unfortunately, no structure is recovered for these parts of the program. This can occur when regions do not have an unambiguous successor, or when loop bodies are too complex. This problem motivated the iterative refinement technique of our algorithm, which we describe in §3.

2.2 System Overview

Figure 1 shows the high-level overview of the approach that Phoenix takes to decompile a target binary. Like most previous work, Phoenix uses a number of stages, where the output of stage i is the input to stage $i+1$. Phoenix can fail to output decompiled source if any of its four stages fails. For this reason we provide an overview of each stage in this section. The first two stages are based on existing implementations. The last two use novel techniques developed specifically for Phoenix.

2.3 Stages I and II—Existing Work

Control Flow Graph Recovery The first stage parses the input binary's file format, disassembles the binary, and creates a control flow graph (CFG) for each function. At a high level, a control flow graph is a program representation in which vertices represent basic blocks, and edges represent possible control flow transitions between blocks. (See §2.1 for more detail.) While precisely identifying binary code in an executable is known to be hard in the general case, current algorithms have been shown to work well in practice [3, 4, 24, 25].

There are mature platforms that already implement this step. We use the CMU Binary Analysis Platform (BAP) [23], though functionally equivalent platforms, such as BitBlaze [6] or Jakstab [24], should also work. We chose BAP because it is open-source and periodically maintained. BAP lifts sequential x86 assembly instructions in the CFG into an intermediate language called BIL, whose syntax is shown in Table 1 [23]. As we will see, the end goal of Phoenix is to decompile this language into the structural language shown in Table 2.

Variable and Type Recovery The second stage recovers individual variables from the binary code, and assigns them types. Phoenix uses TIE [28] to perform this task. TIE runs Value Set Analysis (VSA) [3] to recover variable locations. TIE then uses a static, constraint-based type inference system similar to the one used in the ML programming language [16]. Roughly speaking, each statement imposes some constraints on the type of variables involved. For example, an argument passed to a function that expects an argument of type T should be of type T, and the denominator in a division must be an integer and not a pointer. The constraints are then solved to assign each variable a type.

2.4 Stage Three—Control-Flow Structure Recovery

The next stage recovers the high-level control flow structure of the program. The input to this stage is an assembly program in CFG form. The goal is to recover high-level, structured control flow constructs such as loops, if-then-else and switch constructs from the graph representation. A program or construct is *structured* if it does not utilize `gotos`. Structured program representations are preferred because they help scale program analysis [31] and make programs easier to understand [18]. The process of recovering a structured representation of the program is sometimes called *control flow structure recovery* or *control flow structuring* in the literature.

Although *control flow structure recovery* is similar in name to *control flow graph recovery* (stage I), the two are very different. Control flow graph recovery starts with a binary program, and produces a control flow graph representation of the program as output. Control flow structure recovery takes a control flow graph representation as input, and outputs the high-level control flow structure of the program, for instance:

```
while (...) { if (...) {...} }.
```

The rest of this paper will only focus on control flow structuring and not control flow graph reconstruction.

Structural analysis is a control flow structuring algorithm that, roughly speaking, matches pre-defined graph schemas or patterns to the control flow constructs that create the patterns [31]. For example, if a structural analysis algorithm identifies a diamond-shape in a CFG, it outputs an if-then-else construct, because if-then-else statements create diamond-shaped subgraphs in the CFG.

However, using structural analysis in a decompiler is not straightforward. We initially tried implementing the most recent algorithm in the literature [19] in Phoenix. We discovered that this algorithm, like previous algorithms, can (1) cause incorrect decompilation, and (2) miss opportunities for recovering structure. These problems motivated us to develop a new structural analysis algorithm for Phoenix which avoids these pitfalls. Our algorithm has two new features. First, our algorithm employs iterative refinement to recover more structure

prog	::=	(*varinfo**, *func**)
func	::=	(string, *varinfo*, *varinfo*, *stmt**)
stmt	::=	*var* := *exp* | `Goto`(*exp*) | `If` *exp* `then` *stmt* `else` *stmt*
		| `While`(*exp*, *stmt*) | `DoWhile`(*stmt*, *exp*)
		| `For`(*stmt*, *exp*, *stmt*)
		| `Sequence`(*stmt**)
		| `Switch`(*exp*,*stmt**)
		| `Case`(*exp*,*stmt*)
		| `Label`(string)
		| `Nop`

Table 2: An abbreviated syntax of the HIL.

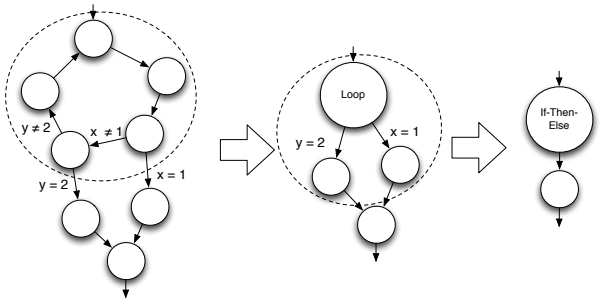


Figure 3: A simple example of how a non-semantics-preserving structural analysis fails.

than previous algorithms. It also features a new property, semantics-preservation. We propose that all structural analysis algorithms should be semantics-preserving to be safely used for decompilation. These topics are a primary focus of this paper, and we discuss them in detail in §3.

2.5 Stage IV—Statement Translation and Outputting C

The input to the next stage of our decompiler is a CFG annotated with structural information, which loosely maps each vertex in the CFG to a position in a control construct. What remains is to translate the BIL statements in each vertex of the CFG to a high-level language representation called HIL. Some of HIL's syntax is shown in Table 2.

Although most statements are straightforward to translate, some require information gathered in prior stages of the decompiler. For instance, to translate function calls, we use VSA to find the offset of the stack pointer at the call site, and then use the type signature of the called function to determine how many arguments should be included. We also perform optimizations to make the final source more readable. There are two types of optimizations. First, similar to previous work, we perform optimizations to remove redundancy such as dead-code elimination [11]. Second, we implement optimizations that improve readability, such as untiling.

During compilation a compiler uses a transformation called *tiling* to reduce high-level program statements into assembly statements. At a high level, tiling takes as input an abstract syntax tree (AST) of the source language and produces an assembly program by covering the AST with semantically equivalent assembly statements. For example, given:

```
x = (y+z)/w
```

tiling would first cover the expression $y+z$ with the `add` instruction, and then the division with the `div` instruction. Tiling will typically produce many assembly instructions for a single high-level statement.

Phoenix uses an *untiling* algorithm. Untiling takes several statements and outputs a single high-level source statement. For instance, at a low-level, $\texttt{High}_1\,[a\&b]$ means to extract the most significant bit from bitwise-anding a with b. This may not seem like a common operation used in C, but it is equivalent to the high-level operation of computing $a <_s 0$ && $b <_s 0$ (i.e., both a and b are less than zero when interpreted as signed integers). Phoenix uses about 20 manually crafted untiling patterns to convert such low-level expressions into an equivalent, higher level form of the computation. The output of this phase is a HIL program.

The final stage in Phoenix is an analysis that takes the HIL representation of the program as input. In this paper, we use an analysis that translates HIL into C, to test Phoenix as a binary-to-C decompiler.

3 Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring

In this section we describe our proposed structural analysis algorithm. Our algorithm builds on existing work by adding iterative refinement and semantics-preserving schemas. Before we discuss the details of our algorithm, we highlight the importance of these additions.

Semantics Preservation Structural analysis was originally invented to scale data flow analysis by summarizing the reachability properties of each region of a program's control flow graph. Later, decompiler researchers adapted structural analysis and its predecessor, interval analysis, to recover the control flow structure of decompiled programs [13, 22]. Unfortunately, vanilla structural analysis can identify control flow that is consistent with a graph's reachability, but is inconsistent with the graph's semantics. This is a problem for decompilation, where the goal is to infer the semantics and abstractions of the binary program and not its reachability.

For instance, structural analysis would identify the loop in the leftmost graph of Figure 3 and reduce it to a single node representing the loop, thus producing the diamond-shaped graph shown in the middle. This graph matches the schema for an if-then-else region, which would also be reduced to a single node. Finally, the two remaining nodes would then be reduced to a sequence node

(not shown), at which point structural analysis is finished. This would be correct for data flow analysis, which only depends on reachability. However, the first node reduction is *not* semantics-preserving. This is easy to see for the case when $x = 1$ and $y = 2$ hold. In the original graph, the first loop exit would be taken, since $x = 1$ matches the first exit edge's condition. However, in the middle graph, both exit edges can be taken. Producing decompiler output from the middle graph would be incorrect.

A structural analysis algorithm that is safe for decompilation must preserve the semantics of a control flow graph during each reduction. Otherwise the recovered control flow structure can be inconsistent with the actual control flow in the binary. Most schemas in standard structural analysis [31, p. 203] preserve semantics, but the natural loop schema is one that does not. A natural loop is a generalized definition of a single-entrance loop that can have multiple exits. The loop in Figure 3 is a natural loop, for example, because it has one entrance and two exits. We demonstrate that fixing non-semantics preserving rules in our structural analysis algorithm increases the number of utilities Phoenix correctly decompiles by 30% (see §4). We describe these fixes in detail in the upcoming sections.

Iterative Refinement At a high level, *refinement* is the process of removing an edge from a CFG by emitting a `goto` in its place, and *iterative refinement* refers to the repeated application of refinement until structuring can progress. This may seem counter-intuitive, since adding a `goto` seems like it would *decrease* the amount of structure recovered. However, the removal of a *carefully-chosen* edge can potentially allow a schema to match the refined CFG, thus enabling the recovery of additional structure. (We describe which edges are removed in the following sections.) The alternative to refinement is to recover no structure for problematic parts of the CFG. We show that algorithms that do not utilize refinement, including existing structural analysis algorithms [19, 31, 35], require 30× more gotos (from 40 to 1,229). As far as we know, our work is the first to specify and implement iterative refinement for decompilation.

3.1 Algorithm Overview

We focus on the novel aspects of our algorithm in this paper and refer readers interested in any structural analysis details elided to standard sources [31].

Like vanilla structural analysis, our algorithm visits nodes in post-order in each iteration. Intuitively, this means that all descendants of a node will be visited (and hence had the chance to be reduced) before the node itself. The algorithm's behavior when visiting node n depends on whether the region at n is acyclic (has no loop) or not. For an acyclic region, the algorithm tries to match the subgraph at n to an acyclic schemas (§3.2). If there is no match, and the region is a switch candidate, then it attempts to refine the region at n into a switch region (§3.4). If n is cyclic, the algorithm compares the region at n to the cyclic schemas (§3.5). If this fails, it refines n into a loop (§3.6). If both matching and refinement do not make progress, the current node n is then skipped for the current iteration of the algorithm. If there is an iteration in which *all* nodes are skipped, i.e., the algorithm makes no progress, then the algorithm employs a last resort refinement (§3.7) to ensure that progress can be made in the next iteration.

3.2 Acyclic Regions

The acyclic region types supported by Phoenix correspond to the acyclic control flow operators in C: sequences, ifs, and switches. The schemas for these regions are shown in Table 3. For example, the $\mathsf{Seq}[n_1, \cdots, n_k]$ region contains k regions that always execute in the listed sequence. $\mathsf{IfThenElse}[c,\ n,\ n_t,\ n_f]$ denotes that n_t is executed after n when condition c holds, and otherwise n_f is executed.

Our schemas match both shape and the boolean predicates that guard execution of each node, to ensure semantics preservation. These conditions are implicitly described using meta-variables in Table 3, such as c and $\neg c$. The intuition is that shape alone is not enough to distinguish which control structure should be used in decompilation. For instance, a switch for cases $x = 2$ and $x = 3$ can have the diamond shape of an if-then-else, but we would not want to mistake a switch for an if-then-else because the semantics of if-then-else requires the outgoing conditions to be inverses.

3.3 Tail Regions and Edge Virtualization

When no subgraphs in the CFG match known schemas, the algorithm is stuck and the CFG must be refined before more structure can be recovered. The insight behind *refinement* is that removing an edge from the CFG may allow a schema to match, and *iterative refinement* refers to the repeated application of refinement until a match is possible. Of course, each edge in the CFG represents a possible control flow, and we must represent this control flow in some other way to preserve the program semantics. We call removing the edge in a way that preserves control flow *virtualizing* the edge, since the decompiled program behaves as if the edge was present, even though it is not.

In Phoenix, we virtualize an edge by collapsing the source node of the edge into a tail region (see §2.1). Tail regions explicitly denote that there should be a control transfer at the end of the region. For instance, to virtualize the edge (n_1, n_2), we remove the edge from the CFG, insert a fresh label l at the start of n_2, and collapse n_1 to a tail region that denotes there should be a `goto` l statement at the end of region n_1. Tail regions can also be translated into `break` or `continue` statements when used inside a switch or loop. Because the tail region explicitly represents the control flow of the virtualized edge, it is

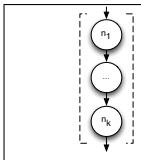
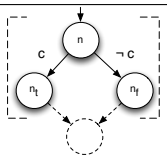
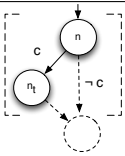
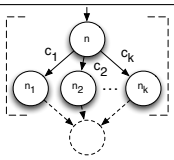
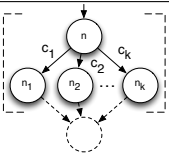
	Seq[$n_1, \cdots, n_k$]: A block of sequential regions that have a single predecessor and a single successor.
	IfThenElse[c, n, n_t, n_f]: If-then-else region.
	IfThen[c, n, n_t]: If-then region.
	IncSwitch[n, $(c_1,n_1),\cdots,(c_k,n_k)$]: Incomplete switch region. The outgoing conditions must be pairwise disjoint and satisfy $\bigvee_{i\in[1,k]} c_i \neq \mathit{true}$.
	Switch[n, $(c_1,n_1),\cdots,(c_k,n_k)$]: Complete switch region. The outgoing conditions must be pairwise disjoint and satisfy $\bigvee_{i\in[1,k]} c_i = \mathit{true}$.

Table 3: Acyclic regions.

safe to remove the edge from the graph and ignore it when doing future pattern matches.

3.4 Switch Refinement

If the subgraph at node n is acyclic but fails to match a known schema, we try to refine the subgraph into a switch. Regions that would match a switch schema in Table 3 but contain extra edges are *switch candidates*. A switch candidate can fail to match the switch schema if it has extra incoming edges or multiple successors. For instance, the nodes in the IncSwitch[·] box in Figure 4 would not be identified as an IncSwitch[·] region because there is an extra incoming edge to the default case node.

We first refine switch candidates by ensuring that the switch head is the only predecessor for each case node. We remove any other incoming edge by virtualizing them. The next step is to ensure there is a single successor of all nodes in the switch. To find the successor, we first identify the immediate post-dominator of the switch head. If this node is the successor of any of the case nodes, we select it as the switch successor. If not, we select the node that (1) is a successor of a case node, (2) is not a case node itself, and (3) has the highest number of incoming edges from case nodes. After we have identified the successor, we remove all outgoing edges from the case nodes to other nodes by virtualizing them.

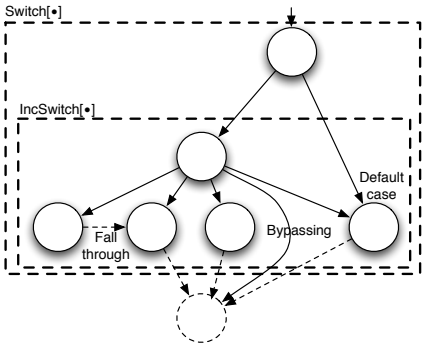


Figure 4: Complete and incomplete switches.

After refinement, a switch candidate is usually collapsed to a IncSwitch[·] region. For instance, a common implementation strategy for switches is to redirect inputs handled by the default case (e.g., $x > 20$) to a default node, and use a jump table for the remaining cases (e.g., $x \in [0,20]$). This relationship is depicted in Figure 4, along with the corresponding region types. Because the jump table only handles a few cases, it is recognized as an IncSwitch[·]. However, because the default node handles all other cases, together they constitute a Switch[·].

3.5 Cyclic Regions

If the subgraph at node n is cyclic, we first test if it matches a cyclic pattern. The first step is to identify any loops of which n is the loop header. It is possible for a node to be the loop header of multiple loops. For instance, nested do-while loops share a common loop header. We identify distinct loops at node n by finding back edges pointing to n (see §2.1). Each back edge (n_b, n) defines a loop body consisting of the nodes that can reach n_b without going through the loop header, n. The loop with the smallest loop body is reduced first. This must happen before the larger loops can match the cyclic region patterns, because there is no schema for nested loops.

As shown in Table 4, there are three types of loops. While[·] loops test the exit condition before executing the loop body, whereas DoWhile[·] loops test the exit condition after executing the loop body. If the exit condition occurs in the middle of the loop body, the region is a natural loop. Natural loops do not represent one particular C looping construct, but can be caused by code such as

```
while (1) { body1; if (e) break; body2; }
```

Notice that our schema for natural loops contains no outgoing edges from the loop. This is not a mistake, but is required for semantics-preservation. Because NatLoop[·] regions are decompiled to

```
while (1) {...},
```

which has no exits, the body of the loop must trigger any loop exits. In Phoenix, the loop exits are represented by

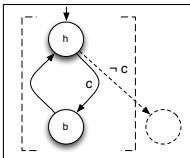
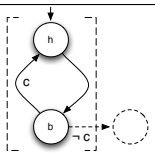
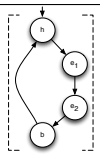
	While[c,h,s,b]: A while loop.
	DoWhile[c,h,b] : A do-while loop.
	NatLoop[$h,b,e_1 \cdots e_k$] : A natural loop. Note that there are no edges leaving the loop; outgoing edges must have been virtualized during refinement to match this schema.

Table 4: Cyclic regions.

a tail region, which corresponds to a `goto`, `break`, or `continue` in the decompiled output. These tail regions are added during loop refinement, which we discuss next.

3.6 Loop Refinement

If any loops were detected with loop header n that did not match a loop schema, we start loop refinement. Cyclic regions may fail to match loop schemas because 1) there are multiple entrances to the loop, 2) there are too many exits from the loop, or 3) the loop body cannot be collapsed (i.e., is a proper region).

The first step of loop refinement is to ensure the loop has a single entrance (nodes with incoming edges from outside the loop). If there are multiple entrances to the loop, we select the one with the most incoming edges, and virtualize the other entrance edges.

The next step is to identify the type of loop we have. If there is an exit edge from the loop header, we have a While[·] candidate. If there is an outgoing edge from the source of the loop's back edge (see §2.1), we have a DoWhile[·] candidate. Otherwise, we select any exit edge and have a NatLoop[·]. The exit edge determines the successor of the loop, i.e., the statement that is executed immediately after the loop. The lexical successor in turn determines which nodes are lexically contained in the loop.

Phoenix virtualizes any edge leaving the lexically contained loop nodes other than the exit edge. Edges that go to the loop header use the `continue` tail regions, while edges that go to the loop successor use the `break` regions. Any other virtualized edge becomes a `goto`.

In our first implementation, we considered the lexically contained nodes to be the loop body defined by the loop's back edge[2]. However, we found this definition introduced `goto` statements when the original program had `break` statements, as in Figure 5(a). The `puts("c")` statement is *not* in the loop body according to the standard definition, because it cannot reach the loop's back edge, but it *is* lexically contained in the loop. Obviously, to be able to use the `break` statement, it must be lexically contained inside the loop body, or there would be no loop to break out of.

Our observation is that the nodes lexically contained in the loop should intuitively consist of the loop body *and* any nodes that execute after the loop body but before the successor. More formally, this corresponds to the loop body, and the nodes that are dominated by the loop header, excluding any nodes reachable from the loop's successor without going through the loop header. For example, `puts("c")` in Figure 5(b) is considered as a node that executes between the loop body and the successor, and thus Phoenix places it lexically inside the loop. When Phoenix uses the standard loop membership definition used in structural analysis [31], Phoenix outputs `goto`s, as in Figure 5(c). To quantify this, enabling the new loop membership definition decreased the numbers of `goto`s Phoenix emitted by 45% (73 to 40) in our evaluation (§4).

The last loop refinement step is to remove edges that may prevent the loop body from being collapsed. This can happen for instance because a `goto` was used in the input program. This step is only performed if the prior loop refinement steps did not remove any edges during the latest iteration of the algorithm. For this, we use the last resort refinement on the loop body, which we describe below.

3.7 Last Resort Refinement

If the algorithm does not collapse any nodes or perform any refinement during an iteration, Phoenix removes an edge in the graph to allow it to make progress. We call this process the last resort refinement, because it has the lowest priority, and always allows progress to be made. Last resort refinement prefers to remove edges whose source does not dominate its target, nor whose target dominates its source. These edges can be thought of as cutting across the dominator tree. By removing them, we leave edges that reflect more structure because they reflect a dominator relationship.

4 Evaluation

In this section, we describe the results of our experiments on Phoenix. At a high level, these results demonstrate that Phoenix is suitable for use in program analysis. Specifically, we show that the techniques employed by Phoenix lead to significantly more correct decompilation and more recovered structure than the *de facto* industry standard Hex-Rays. Phoenix was able to decompile 114% more utilities that passed the entire `coreutils` test suite than Hex-Rays (60 vs 28). Our results show that employing semantics-preserving schemas increased correctness by 30% (from 46 to 60). We attribute most remaining correctness errors in Phoenix to type recovery (see §5). Phoenix was able to structure the control flow for 8,676 functions using only 40 `goto`s. This corresponds to recovering 30×

```
1  int f(void) {
2    int a = 42;
3    int b = 0;
4    while (a) {
5      if (b) {
6        puts("c");
7        break;
8      } else {
9        puts("d");
10     }
11     a--;
12     b++;
13   }
14   puts("e");
15   return 0;
16 }
```

(a) Loop refinement example

```
1  t_reg32 f (void) {
2    t_reg32 var_20 = 42;
3    t_reg32 var_24;
4    for (var_24 = 0; var_20 != 0;
5    var_24 = var_24 + 1) {
6      if (var_24 != 0) {
7        puts("c");
8        break;
9      }
10     puts("d");
11     var_20 = var_20 - 1;
12   }
13   puts("e");
14   return 0;
15 }
```

(b) Phoenix decompiled output of (a) *with* new loop membership definition

```
1  t_reg32 f (void)
2  {
3    t_reg32 var_20 = 42;
4    t_reg32 var_24;
5    for (var_24 = 0;
6    var_20 != 0; var_24 = var_24 + 1)
7    {
8      if (var_24 != 0) goto lab_1;
9      puts("d");
10     var_20 = var_20 - 1;
11   }
12   lab_2:
13   puts("e");
14   return 0;
15   lab_1:
16   puts("c");
17   goto lab_2;
18 }
```

(c) Phoenix decompiled output of (a) *without* new loop membership definition

Figure 5: Loop refinement with and without new loop membership definition.

more structure (40 gotos vs 1,229) than our implementation of the latest structural analysis algorithm in the literature [19], which highlights the importance of iterative refinement. Hex-Rays also recovered less structure than Phoenix, requiring 51 `goto`s for the same set of functions.

4.1 Phoenix Implementation

Our implementation of Phoenix consists of an extension to the BAP framework. We implemented it in OCaml, to ease integration with BAP, which is also implemented in OCaml. Phoenix alone consists of 3,766 new lines of code which were added to BAP. Together, the decompiler and TIE comprise 8,443 lines of code. For reference, BAP consists of 29,652 lines of code before our additions. We measured the number of lines of code with David A. Wheeler's `SLOCCount` utility.

4.2 Metrics

We propose two *quantitative* dimensions for evaluating the suitability of decompilers for program analysis, and then evaluate Phoenix on them:

- **Correctness** Correctness measures whether the decompiled output is equivalent to the original binary input. If a decompiler produces output that does not actually reflect the behavior of the input binary, it is of little utility in almost all settings. For program analysis, we want decompilers to be correct so that the decompiler does not introduce imprecision. In our experiments we utilize high-coverage unit tests to measure correctness.
- **Structuredness** Recovering control flow structure helps program analysis and humans alike. Structured code is easier for programmers to understand [18], and helps scale program analysis in general [31]. Therefore, we propose that decompiler output with fewer unstructured control flow commands such as `goto` are better.

The benefit of our proposed metrics is that they can be evaluated quantitatively and thus can be automatically measured. These properties makes them suitable for an objective comparison of decompilers.

Existing Metrics Note that our metrics are vastly different than those appearing in previous decompiler work. Cifuentes proposed using the ratio of the size of the decompiler output to the initial assembly as a "compression ratio" metric, i.e., 1-(LOC decompiled/LOC assembly) [11]. The idea was the more compact the decompiled output is than the assembly code, the easier it would be for a human to understand the decompiled output. However, this metric sidesteps whether the decompilation is correct or even compilable. A significant amount of previous work has proposed no metrics. Instead, they observed that the decompiler produced output, or had a manual qualitative evaluation on a few, small examples [9, 11, 20, 21, 38]. Previous work that does measure correctness [19, 28] only focuses on a small part of the decompilation process, e.g., type recovery or control flow structuring. However, it does not measure end-to-end correctness of the decompiler as a whole.

4.3 Coreutils Experiment Motivation

We tested Phoenix on the GNU `coreutils` 8.17 suite of utilities. `coreutils` consists of 107^3 mature, standard programs used on almost every Linux system. `coreutils` also has a suite of high-coverage tests that can be used to measure correctness. Though prior work has studied individual decompiler components on a large scale (see §6), to the best of our knowledge, our evaluation on `coreutils` is orders of magnitude larger than any other end-to-end decompiler evaluation.

Tested Decompilers In addition to Phoenix, we tested the latest publicly available version of the academic decompiler Boomerang [38] and Hex-Rays [22], the *de*

facto industry standard decompiler. We tested the latest Hex-Rays version, which is 1.7.0.120612 as of this writing.

We also considered other decompilers such as REC [34], DISC [26], *dcc* [11]. However, these compilers either produced pseudocode (e.g., REC), did not work on x86 (e.g., *dcc*), or did not have any documentation that suggested advancements beyond Boomerang (e.g., DISC).

We encountered serious problems with both Boomerang and Hex-Rays in their default configurations. First, Boomerang failed to produce any output for all but a few `coreutils` programs. The problem was that Boomerang would get stuck while decompiling one function, and would never move on to other functions. We looked at the code, but there appeared to be no easy or reasonable fix to enable some type of per-function timeout mechanism. Boomerang is also no longer actively maintained. Second, Hex-Rays did not output compliant C code. In particular, Hex-Rays uses non-standard C types and idioms that only Visual Studio recognizes, and causes almost every function to fail to compile with `gcc`. Specifically, the Hex-Rays website states: "[...] the produced code is not supposed to be compilable and many compilers will complain about it. This is a deliberate choice of not making the output 100% compilable because the goal is not to recompile the code but to analyze it." Even if Hex-Rays output is intended to be analyzed rather than compiled, it should still be correct modulo compilation issues. After all, there is little point to pseudo-code if it is semantically incorrect.

Because Hex-Rays was the only decompiler we tested that actually produced output for real programs, we investigated the issue in more detail and noticed that the Hex-Rays output was only uncompilable because of the Visual Studio idioms and types it used. In order to offer a conservative comparison of Phoenix to existing work, we wrote a post-processor for Hex-Rays that translates the Hex-Rays output to compliant C. The translation is extremely straightforward. For example, one of the translations is that types such as `unsigned __intN` must be converted to `uintN_t`[4]. All experiments are reported with respect to the post-processed Hex-Rays output. We stress this is intended to make the comparison more fair: without the post-processing Hex-Rays output fails to compile using `gcc`.

4.4 Experimental Results Details

4.4.1 Setup

Testing decompilers on real programs is difficult because they are not capable of decompiling all functions. This means that we cannot decompile every function in a binary, recompile the resulting source, and expect to have a working binary. However, we would like to be able to test the functions that *can* be decompiled. To this end, we propose the substitution method.

The goal of the substitution method is to produce a *recompiled* binary that consists of a combination of original source code and decompiled source code. We implemented the substitution method by using CIL [32] to produce a C file for each function in the original source code. We compiled each C file to a separate object file. We also produced object files for each function emitted by the decompiler in a similar manner. We then created an initial recompiled binary by linking all of the original object files (i.e., object files compiled from the original source code) together to produce a binary. We then iteratively substituted a decompiler object file (i.e., object files compiled from the decompiler's output) for its corresponding original object file. If linking this new set of object files succeeded without an error, we continued using the decompiler object file in future iterations. Otherwise we reverted to using the original object file. For our experiments, we produced a recompiled binary for each decompiler and utility combination.

Of course, for fairness, we must ensure that the recompiled binaries for each decompiler have approximately the same number of decompiled functions, since non-decompiled functions use the original function definition from the `coreutils` source code, which presumably passes the test suite and is well-structured. The number of recompilable functions output by each decompiler is broken down by utility in Figure 6. Phoenix recompiled 10,756 functions in total, compared to 10,086 functions for Hex-Rays. The Phoenix recompiled binaries consist of 82.2% decompiled functions on average, whereas the Hex-Rays binaries contain 77.5%. This puts Phoenix at a slight disadvantage for the correctness tests, since it uses fewer original functions. Hex-Rays did not produce output after running for several hours on the `sha384sum` and `sha512sum` utilities. Phoenix did not completely fail on any utilities, and was able to decompile 91 out of 110 functions (82.7%) for both `sha384sum` and `sha512sum`. (These two utilities are similar). We discuss Phoenix's limitations and common reasons for failure in §5.

4.4.2 Correctness

We test the correctness of each recompiled utility by running the `coreutils` test suite with that utility and *original* versions of the other utilities. We do this because the `coreutils` test suite is self-hosting, that is, it uses its own utilities to set up the tests. For instance, a test for `mv` might use `mkdir` to setup the test; if the recompiled version of `mkdir` fails, we could accidentally blame `mv` for the failure, or worse, incorrectly report that `mv` passed the test when in reality the test was not properly set up.

Each tested utility U can either pass all unit tests, or fail. We do not count the number of failed unit tests, because many utilities have only one test that exercises them. We have observed decompiled utilities that crash on every execution and yet fail only a single unit test. Thus, it would be misleading to conclude that a decompiler performed well by "only" failing one test.

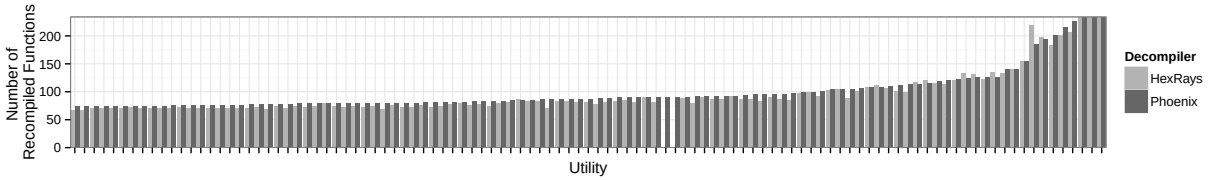


Figure 6: The number of functions that were decompiled and recompiled by each decompiler, broken down by utility. Note that Hex-Rays got stuck on two utilities for unknown reasons; this is not a mistake.

	Phoenix	HR
Correct utilities recompiled	60	28
Correct utilities recompiled (semantics-preservation disabled)	46	n/a
Percentage recompiled functions (correct utilities only)	85.4%	73.8%

Table 5: Correctness statistics for the `coreutils` experiment. Note that this includes two utilities for which Hex-Rays recompiled zero functions (thus trivially passing correctness).

	Phoenix	HR
Total gotos	40	51
Total gotos (without loop membership)	73	n/a
Total gotos (without refinement)	1,229	n/a

Table 6: Structuredness statistics for the `coreutils` experiment. The statistics only reflect the 8,676 recompilable functions output by both decompilers.

The results of the correctness tests are in Table 5. To summarize, Hex-Rays recompiled 28 utilities that passed the `coreutils` test suite. Phoenix was able to recompile 60 passing utilities (114% more). However, we want to ensure that these utilities are not simply correct because they consist mostly of the original `coreutils` functions. This is not the case for Phoenix: the recompiled utilities that passed all tests consisted of 85.4% decompiled functions on average, which is actually higher than the overall Phoenix average of 82.2%. The correct Hex-Rays utilities consisted of 73.8% decompiled functions, which is less than the overall Hex-Rays average of 77.5%. As can be seen in Figure 6, this is because Hex-Rays completely failed on two utilities. The recompiled binaries for these utilities consisted completely of the original source code, which (unsurprisingly) passed all unit tests. Excluding those two utilities, Hex-Rays only compiled 26 utilities that passed the unit tests. These utilities consisted of 79.4% decompiled functions on average.

We also re-ran Phoenix with the standard structural analysis schemas, including those that are *not* semantics-preserving, in order to evaluate whether semantics-preservation has a noticeable effect on correctness. With these schemas, Phoenix produced only 46 correct utilities. This 30% reduction in correctness (from 60 down to 46) illustrates the importance of using semantics-preserving schemas.

4.4.3 Structuredness

Finally, we measure the amount of structure recovered by each decompiler. Our approach here is straightforward: we count the number of `goto` statements emitted by each decompiler. To ensure a fair comparison, we only consider the intersection of recompilable functions emitted by both decompilers, which consists of 8,676 functions. Doing otherwise would penalize a decompiler for outputting a function with `goto` statements, even if the other decompiler could not decompile that function at all.

The overall structuredness results are depicted in Table 6, with the results broken down per utility in Figure 7. In summary, Phoenix recovered the structure of the 8,676 considered functions using only 40 `gotos`. Furthermore, Phoenix recovered significantly less structure when either refinement (1189 more `gotos`) or the new loop membership definition (33 more) was disabled. Our results suggest that structuring algorithms without iterative refinement [19, 31, 35] will recover less structure. Thus, Hex-Rays likely employs a technique similar to iterative refinement.

5 Limitations and Future Work

5.1 BAP Failures

Phoenix uses BAP [23] to lift instructions to a simple language that is then analyzed. BAP does not have support for floating point and other exotic types of instructions. Phoenix will not attempt to decompile any function that contains instructions which are not handled by BAP. BAP can also fail for other reasons. It uses value set analysis (VSA) to perform CFG recovery, and to help with other parts of the decompilation process. If VSA or other mandatory analyses fail, then the decompiler cannot proceed. These problems can cascade to affect other functions. For instance, if CFG recovery for function g fails and function f calls g, function f will also fail, because it calls a function with an unknown type.

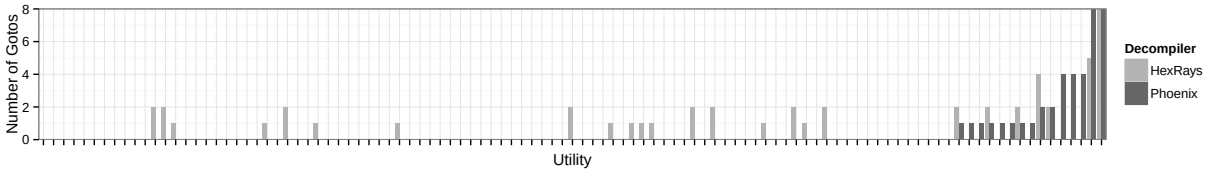


Figure 7: The number of gotos emitted by each decompiler, broken down by utility, only including functions that were decompiled and recompiled by *both* decompilers.

5.2 Correctness Failures

Because Phoenix is designed for program analysis, we want it to be correct. Our experiments show that, although Phoenix significantly improves over prior work with respect to correctness, Phoenix's output is not always correct. The good news is that we can attribute most correctness errors in Phoenix to the underlying type recovery component we used, TIE [28]. Many of the problems, which we describe below, only became apparent when TIE was stress-tested by Phoenix.

Iterative Variable Recovery TIE does not always identify all local variables. For instance, if function `foo` takes a pointer to an integer, and a function calls `foo(x)`, then TIE infers that `x` is a subtype of a pointer to an integer. However, TIE does *not* automatically infer that `*x`, the locations that `x` can point to, are potentially unrecovered integer variables. TIE does not recover such variables because it would need to iteratively discover variables, generate and solve type constraints to do so. Unfortunately, undiscovered variables can cause incorrect decompilation for Phoenix. For example, if the undiscovered variable is a struct on the stack, space for the struct is never allocated, which allows the called function to read and overwrite other variables on the stack of the callee. This is the leading cause of correctness errors in Phoenix. In the future, we plan to investigate running type recovery until the set of known variables reaches a fix point.

Calling Conventions TIE currently assumes that all functions use the `cdecl` calling convention, and does not support multi-register (64-bit) return values. Unfortunately, this can make Phoenix output incorrect or uncompilable code. In the future, we plan to use an interprocedural liveness analysis to automatically detect calling conventions based on the behavior of a function and the functions that call it. Our goal is to detect and understand calling conventions automatically, even when they are non-standard. This is important for analyzing malware, some of which uses unusual calling conventions to try to confuse static analysis.

Recursive Types TIE has no support for recursive types, although these are used quite frequently for data structures like linked lists and binary trees. This means that the type

```
struct foo {int a; struct foo *next;}
```

will be inferred as

```
struct foo {int a; void* next;}
```

which does not specify what type of element `next` points to. Since Phoenix is intended to be the frontend of an analysis platform, we would like to recover the most specific type possible. We plan to investigate more advanced type inference algorithms that can handle recursive types.

6 Related Work

At a high level, there are three lines of work relevant to Phoenix. First, work in end-to-end decompilers. Second, work in control structure recovery, such as loop identification or structural analysis, which spans both decompiler and compiler literature. Third, literature pertaining to type recovery.

Decompilers The earliest work in decompilation dates back to the 1960's. For an excellent and thorough review of the literature in decompilation and several related areas up to around 2007, see Van Emmerik's thesis [38, ch. 2]. Another in-depth overview is available online [17].

Modern decompilers typically trace their roots in Cifuentes' 1994 thesis on *dcc* [11], a decompiler for 80286 to C. The structuring algorithm used in *dcc* is based on interval analysis [1]. Cifuentes proposed the compression ratio metric (see §4.2), but did not measure correctness on the ten programs that *dcc* was tested on [12]. Since compression is the target metric, *dcc* outputs assembly if it encounters code that it cannot handle. Cifuentes et al. have also created a SPARC asm to C decompiler, and measured compressibility and the number of recovered control structures on seven SPEC1995 programs [14]. Again, they did not test the correctness of the decompilation output. Cifuentes [11] pioneered the technique of recovering short-circuit evaluations in compound expressions (e.g., `x && (!y || z)` in C). Her technique has been improved over the years by other researchers [19, 40].

Chang et al. [9] also use compressibility in their work on cooperating decompilers for the three programs they tested. Their main purpose was to show they can find bugs in the decompiled source that were known to exist in the binary. However, correctness of the decompilation itself was not verified.

Boomerang is a popular open-source decompiler started by Van Emmerik as part of his Ph.D. [38]. The main idea

of Van Emmerik's thesis is that decompilation analysis is easier on the Single Static Assignment (SSA) form of a program. In his thesis, Van Emmerik's experiments are limited to a case study of Boomerang coupled with manual analysis to reverse engineer a single 670KB Windows program. We tested Boomerang as part of our evaluation, but it failed to produce any output on all but a few of our test cases after running for several hours.

The structuring algorithm used in Boomerang first appeared in Simon [36], who in collaboration with Cifuentes proposed a new algorithm known as "parenthesis theory". Simon's own experiments showed that parenthesis theory is faster and more space efficient than the interval analysis-based algorithm in *dcc*, but recovers less structure.

Hex-Rays is the *de facto* industry decompiler. The only information we have found on Hex-Rays is a 2008 write-up [22] by Hex-Rays' author, Guilfanov, who revealed that Hex-Rays also performs structural analysis. However, Hex-Rays achieves much better structuredness than vanilla structural analysis, which suggests that Hex-Rays is using a heavily modified version. There are many other binary-to-C decompilers such as REC [34] and DISC [26]. However, our experience suggests that they are not as advanced as Hex-Rays.

Our focus is on decompiling C binaries. Other researchers have investigated decompiling binaries from managed languages such as Java [30]. The set of challenges they face are fundamentally different. On the one hand, these managed binaries contain extra information such as types; on the other hand, recovering the control flow itself in the presence of exceptions and synchronization primitives is a difficult problem.

Control Structure Recovery Control structure recovery is also studied in *compilation*. This is because by the time compilation is in the optimization stage, the input program has already been parsed into a low-level intermediate representation (IR) in which the high-level control structure has been destroyed. Much work in program optimization therefore attempts to recover the control structures.

The most relevant line of work in this direction is the elimination methods in data flow analysis (DFA), pioneered by Allen [1] and Cooke [15] in the 1970's and commonly known as "interval analysis". Sharir [35] subsequently refined interval analysis into structural analysis. In Sharir's own words, structural analysis can be seen as an "unparser" of the CFG. Besides the potential to speed up DFA even more when compared to interval analysis, structural analysis can also cope with irreducible CFGs.

Engel et al. [19] are the first to extend structural analysis to handle C-specific control statements. Specifically, their Single Entry Single Successor (SESS) analysis adds a new tail region type, which corresponds to the statements that appear before a `break` or `continue`. For example, suppose `if (b) { foo; break; }` appears in a loop, then the statements represented by `foo` would belong to a tail region. Unfortunately, the SESS analysis does not use iterative refinement, and can get stuck when on unstructured code. We show in our evaluation that this leads to a large amount of structure being missed.

Another line of related work lies in the area of program schematology, of which "Go To Statement Considered Harmful" by Dijkstra [18] is the most famous. Besides the theoretical study of the expressive power of goto vs. high-level control statements (see, e.g., [33]), this area is also concerned with the automatic structuring of (unstructured) programs, such as the algorithm by Baker [2]. Unfortunately, none of these algorithms perform iterative refinement.

Type Recovery Besides control structure recovery, a high-quality decompiler should also recover the types of variables. Much work has gone into this recently. Phoenix uses TIE [28], which recovers types statically. In contrast, REWARDS [29] and Howard [37] recover types from dynamic traces. Other work has focused on C++-specific issues, such as virtual table recovery [20, 21].

7 Conclusion

We presented Phoenix, a new binary-to-C decompiler designed to be accurate and effectively recover abstractions. Phoenix can help leverage the wealth of existing source-based tools and techniques in security scenarios, where source code is often unavailable. Phoenix uses a novel control flow structuring algorithm that avoids a previously unpublished correctness pitfall in decompilers, and can iteratively refine program control flow graphs to recover more control flow structure than existing algorithms. We proposed two metrics for testing decompilers: correctness and structuredness. Based on these two metrics, we evaluated Phoenix and the *de facto* industry standard decompiler, Hex-Rays. Phoenix decompiled twice as many utilities correctly as Hex-Rays, and recovered more structure.

Notes

[1] The algorithm for identifying tail regions is unspecified [19], so it is possible that our implementation behaves differently than the original.

[2] Engel et al. do not report how their algorithm [19] selects the nodes to be considered in the loop body, besides noting that they should have a single successor. We initially used the standard structural analysis definition [31]

[3] The number of utilities built depends on the machine that `coreutils` is compiled on. This is the number applicable to our testing system, which ran Ubuntu 12.04.1 x86-64. We compiled `coreutils` in 32-bit mode because the current Phoenix implementation only supports 32-bit binaries.

[4] Although it seems like this should be possible to implement using only a C header file containing some `typedefs`, a `typedef` has its qualifiers fixed. For instance, `typedef int foo` is equivalent to `typedef signed int foo`, and thus the type `unsigned foo` is not allowed because `unsigned signed int` does not make sense.

References

[1] Frances E. Allen. Control Flow Analysis. In *Symposium on Compiler Optimization*, pages 1–19. ACM, 1970.

[2] Brenda S. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 24(1):98–120, 1977.

[3] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin at Madison, August 2007.

[4] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the Association for Computing Machinery*, 53(2):66–75, 2010.

[6] BitBlaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[7] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: the world's fastest taint tracker. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2011.

[8] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Efficient and accurate detection of integer-based attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[9] Bor-yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of Low-Level Code using Cooperating Decompilers. In *International Symposium on Static Analysis*, pages 318–335, 2006.

[10] Walter Chang and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 39–50, 2008.

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

[12] Cristina Cifuentes. Interprocedural Data Flow Decompilation. *Journal of Programming Languages*, 4:77–99, 1996.

[13] Cristina Cifuentes and K. John Gough. Decompilation of Binary Programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[14] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to High-Level Language Translation. In *International Conference on Software Maintenance*, pages 228–237. IEEE Comput. Soc, 1998.

[15] John Cocke. Global Common Subexpression Elimination. In *ACM Symposium on Compiler Optimization*, pages 20–24, New York, New York, USA, 1970. ACM Press.

[16] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the Symposium on Principles of Programming Languages*, 1982.

[17] The decompilation wiki. http://www.program-transformation.org/Transform/DeCompilation. Page checked 11/9/2012.

[18] Edsger W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[19] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced Structural Analysis for C code Reconstruction from IR Code. In *14th International Workshop on Software and Compilers for Embedded Systems*, pages 21–27, New York, New York, USA, 2011. ACM Press.

[20] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ Decompilation. In *Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[21] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *European Conference on Software Maintenance and Reengineering*, pages 240–243. IEEE, 2010.

[22] Ilfak Guilfanov. Decompilers and Beyond. In *BlackHat*, 2008.

[23] Ivan Jager, Thanassis Avgerinos, Edward J. Schwartz, and David Brumley. BAP: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification*, 2011.

[24] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the Conference on Computer Aided Verification*, 2008.

[25] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

[26] Satish Kumar. DISC: Decompiler for turboc. http://www.debugmode.com/dcompile/disc.htm. Page checked 11/9/2012.

[27] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2001.

[28] 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 Symposium*, February 2011.

[29] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium*, 2010.

[30] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *International Conference on Compiler Construction*, pages 111–127, 2002.

[31] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[32] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[33] W. W. Peterson, T. Kasami, and N. Tokura. On the Capabilities of While, Repeat, and Exit Statements. *Communications of the ACM*, 16(8):503–512, 1973.

[34] Rec Studio 4—reverse engineering compiler. http://www.backerstreet.com/rec/rec.htm. Page checked 11/9/2012.

[35] Micha Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[36] Doug Simon. *Structuring Assembly Programs*. Honours thesis, University of Queensland, 1997.

[37] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a Dynamic Excavator for Reverse Engineering Data Structures. In *Network and Distributed System Security Symposium*, 2011.

[38] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007.

[39] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with kint. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2012.

[40] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. Structuring 2-way Branches in Binary Executables. In *International Computer Software and Applications Conference*, pages 115–118, 2007.