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

Edward J. Schwartz, *Carnegie Mellon University;*
JongHyup Lee, *Korea National University of Transportation;*
Maverick Woo and David Brumley, *Carnegie Mellon University*

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

Edward J. Schwartz

Carnegie Mellon University

JongHyup Lee

Korea National University of Transportation

Maverick Woo

Carnegie Mellon University

David Brumley

Carnegie Mellon University

Abstract

There are many security tools and techniques for analyzing software, but many of them require access to source code. We propose leveraging *decompilation*, the study of recovering abstractions from compiled code, to apply existing source-based tools and techniques to compiled programs. A decompiler should focus on two properties to be used for security. First, it should recover abstractions as much as possible to minimize the complexity that must be handled by the security analysis that follows. Second, it should aim to recover these abstractions correctly.

Previous work in control-flow structuring, an abstraction recovery problem used in decompilers, does not provide either of these properties. Specifically, existing structuring algorithms are not *semantics-preserving*, which means that they cannot safely be used for decompilation without modification. Existing structural algorithms also miss opportunities for recovering control flow structure. We propose a new structuring algorithm in this paper that addresses these problems.

We evaluate our decompiler, Phoenix, and 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 systematic studies of end-to-end decompilers. We show that our decompiler outperforms the *de facto* industry standard decompiler Hex-Rays in correctness by 114%, and recovers $30\times$ more control-flow structure than existing structuring algorithms in the literature.

1 Introduction

Security analyses are often faster and easier when performed on source code rather than on binary code. For example, while the runtime overhead introduced by *source*-based taint checkers can be as low as 0.65% [12], the overhead of the fastest *binary*-based taint checker is over 150% [8]. In addition, many security analyses described in the literature assume access to source code. For instance, there are numerous source-based static vulnerability finding tools such as KINT [40], RICH [9], and Coverity [6], but equivalent binary-only tools are scarce.

In many security scenarios, however, access to source code is simply not a reasonable assumption. Common counterexamples include analyzing commercial off-the-shelf software for vulnerabilities and reverse engineering malware. The traditional approach in security has been to directly apply some form of low-level binary analysis that does not utilize source-level abstractions such as types and functions [5, 7, 10, 24]. Not surprisingly, reasoning at such a low level causes binary analysis to be more complicated and less scalable than source analysis.

We argue that *decompilation* is an attractive alternative to traditional low-level binary-based techniques. At its surface, decompilation is the recovery of a program's source code given only its binary. Underneath, decompilation consists of a collection of *abstraction recovery* mechanisms such as indirect jump resolution, control flow structuring, and data type reconstruction, which recover high-level abstractions that are not readily available in the binary form. Our insight is that by reusing these mechanisms, we can focus our research effort on designing security analyses that take advantage of such abstractions for accuracy and efficiency. In fact, when taken to an extreme, we may even use decompilation to leverage an existing source-based tool—be it a vulnerability scanner [27], a taint engine [12], or a bug finder [6]—by applying it to the decompiled program code.

Of course, decompilation is also extremely beneficial in situations where manual analysis is required. For example, practitioners often reverse-engineer program binaries to understand proprietary file formats, study vulnerabilities fixed in patches, and determine the exploitability of crashing inputs. Arguably, any one of these tasks becomes easier when given access to source code.

Unfortunately, current research in decompilation does not directly cater to the needs of many security applica-

tions. A decompiler should focus on two properties to be used for security. First, it should recover abstractions as much as possible to minimize the complexity that must be handled by the actual security analysis that follows. Second, it should aim to recover these abstractions correctly. As surprising as it may sound, previous work on decompilation almost never evaluated correctness. For example, Cifuentes et al.'s pioneering work [13] and numerous subsequent works [11, 14, 16, 39] all measured either how much smaller the output C code was in comparison to the input assembly, or with respect to some 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 new end-to-end binary-to-C decompiler called *Phoenix*[1] and measure our results with respect to correctness and high-level abstraction recovery.

Phoenix makes use of existing research on principled abstraction recovery where possible. Source code reconstruction requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Recent work such as TIE [28], REWARDS [29], and Howard [38] have largely addressed principled methods for recovering data types. In this paper, we investigate new techniques for recovering high-level control structure.

1.1 The Phoenix Structural Analysis Algorithm

Previous work has proposed mechanisms for recovering high-level control flow based on the structural analysis algorithm and its predecessors [20, 23, 39]. However, they are problematic because they (1) do not feature a correctness property that is necessary to be safely used for decompilation, and (2) miss opportunities for recovering control flow structure. Unfortunately, these problems can cause a security analysis using the recovered control structures to become unsound or scale poorly. These problems 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 in earlier

---

[1] Phoenix is named in honor of the famous "Dragon Book" [1] on compilers. According to Chinese mythology, the phoenix is a supreme bird that complements the dragon (compilation). In Greek mythology, the phoenix can be reborn from the ashes of its predecessor. Similarly, a decompiler can recover source code and abstractions from the compiled form of a binary, even when these artifacts seem to have been destroyed.

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 [32, p. 203] is a control flow structuring algorithm that was originally invented to help accelerate data flow analysis. Later, decompiler researchers adapted this algorithm to reconstruct high-level control flow structures such as if-then-else and do-while from a program's control flow graph (see §2.1). We propose that structuring algorithms should 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. Semantics-preservation is important for security analyses to ensure that the analysis of the structured program also applies to the original binary. Surprisingly, we discovered that common descriptions of structural analysis algorithms are *not* semantics-preserving. For example, in contrast to our natural loop schema in Table 4, other algorithms employ a schema that permits out-going edges (e.g., see [20, Figure 3]). This can lead to incorrect decompilation, such as the example in Figure 3. We demonstrate that fixing this and other schemas to be semantics-preserving increases the number of utilities that Phoenix is able to correctly decompile 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 iteratively refines the graph to continue making progress. 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 gotos 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). Missed structure is problematic in security applications because it can hamper syntax-based deductions—such as the fact that body will execute ten times in for (i=0; i<10; i++) {body;}. Control flow structure is also used to explicitly accelerate some analyses (e.g., data flow analysis [2, 17]), and failure to recover structure can undermine the performance of these

algorithms. Unfortunately, even recent structuring algorithms such as the one in [20, Algorithm 2] do not employ refinement in their descriptions, and thus can fail to recover structure 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 should 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 $30\times$ more control-flow structure than existing research in the literature [20, 32, 36], and 28% more than the *de facto* industry standard decompiler Hex-Rays [23]. Our evaluation uses the 107 programs in GNU `coreutils` as test cases, and is an order of magnitude larger than any other systematic end-to-end decompiler evaluation to date.

3. We propose *correctness* 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 [20]), surprisingly the correctness of a decompiler as a whole has never been measured. We show in our evaluation that Phoenix successfully decompiled over $2\times$ as many programs that pass the `coreutils` test suite 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 of Phoenix. The remainder of the paper focuses on our novel structural analysis algorithm, which is Phoenix's 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
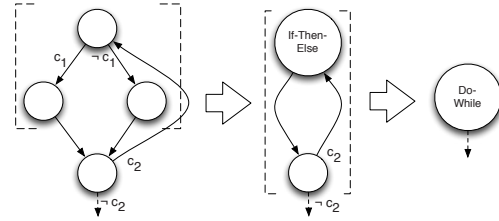

Figure 1: Example of structural analysis.

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. For any node n other than n_s, 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 1 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 cannot recognize loops containing common C constructs such as break and continue. For instance, structural analysis would fail to structure the loop

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

Engel et al. [20] proposed the SESS (single exit single successor) analysis to identify regions that have multiple exits (using break and continue) but share a unique successor. Such exits can be converted into a *tail region* that represents the equivalent control flow construct. In the above example, body would be reduced to a break tail region. Without tail regions, structural analysis stops making progress when reasoning about loops containing multiple exits.

Although the SESS analysis was proposed to help address this problem, the core part of the algorithm, the detection of tail regions, is left unspecified [20, Algorithm 2, Line 15]. We implemented SESS analysis as closely to the paper as possible, but noticed that our implementation often stopped making progress *before* SESS analysis was able to produce a tail region. This can occur when regions do not have an unambiguous successor, or when loop bodies are too complex. Unfortunately, no structure is recovered for these parts of the program. This problem motivated the iterative refinement technique of our algorithm, which we describe in §3.

2.2 System Overview

Figure 2 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 and implementations developed specifically for Phoenix.
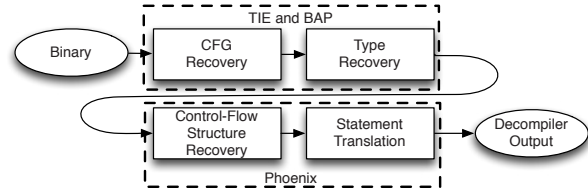

Figure 2: Decompilation flow of Phoenix. Phoenix contains new implementations for control flow recovery and statement translation.

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.

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 [4, 5, 24, 25].

There are mature platforms that already implement this step. We use the CMU Binary Analysis Platform (BAP) [10]. BAP lifts sequential x86 assembly instructions in the CFG into an intermediate language called BIL, whose syntax is shown in Table 1 (see [10]). As we will see, the end goal of Phoenix is to decompile this language into the high-level 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) [4] to recover variable locations. TIE then uses a static, constraint-based type inference system similar to the one used in the ML programming language [31]. 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 III—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 [32] and make programs easier to understand [19]. 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 predefined graph schemas or patterns to the control flow constructs that create the patterns [32]. 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 [20] 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 than previous algorithms. Our algorithm also features semantics-preserving schemas, which allows it to be safely used for decompilation. These topics are a primary focus of this paper, and we discuss them in detail in §3.

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.

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 [13]. 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 to improve readability. Untiling takes several statements and outputs an equivalent high-level source statement. For instance, at a low-level, $\text{High}_1[a\&b]$ 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
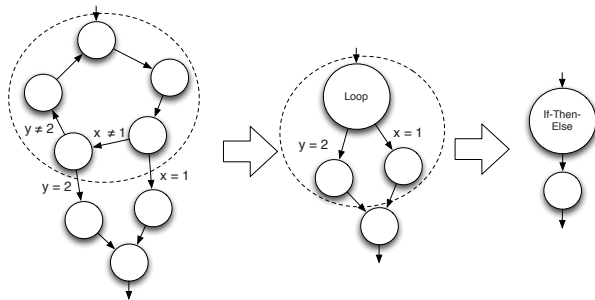
Figure 3: An example of how structural analysis can fail without semantics-preservation.

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 simplify instructions emitted by `gcc`'s code generator. These patterns only improve the readability of the source output, and do not influence correctness or control-flow structure recovery. The output of the statement translation 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, in order 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 a program's CFG. Later, decompiler researchers adapted structural analysis and its predecessor, interval analysis, to recover the control flow structure of decompiled programs [15, 23].

Unfortunately, structural analysis can identify control flow that is consistent with a graph's reachability, but is inconsistent with the graph's semantics.

Such an error from structural analysis is demonstrated in Figure 3. Structural analysis would identify the loop in the leftmost graph 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 both $x = 1$ 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.

Such discrepancies are a problem in security, because they can unintentionally cause unsoundness in analyses. For example, an otherwise sound bug checker, when applied to the program in Figure 3, could state that a bug is present, even if the original program had no bugs.

To avoid unintentional unsoundness, a structural analysis algorithm should preserve the semantics of a CFG during each reduction. Otherwise the recovered control flow structure can become inconsistent with the actual control flow in the binary. Most schemas in structural analysis [32, 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 the schemas in our algorithm to be semantics-preserving increases the number of utilities Phoenix correctly decompiles by 30% (see §4). We describe these modifications 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 Phoenix emits 30× more gotos (from 40 to 1,229) when iterative refinement is disabled.

Recovering structure is important for two reasons. First, structuredness has been shown to help scale program analysis in general [32]. In addition, some analyses use syntactic patterns to find facts, which relies on effective structure recovery. For example, a bug checker might conclude that there is no buffer overflow in

```
char b[10];
int i = 0;
while (i < 10) {
  b[i] = 0;
  i++;
}
```

by syntactically discovering the induction variable i and the loop invariant $i < 10$. If the structuring algorithm does not recover the while loop, and instead represents this loop using `gotos`, the bug checker could be unable to reason that the loop is safe, and output a false positive.

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 [32, p. 203].

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 cyclic (has a loop) or not. For an acyclic region, the algorithm tries to match the subgraph at n to one of the 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 neither matching or refinement 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.

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 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 fails to match a known schema, it may be a switch candidate. *Switch candidates* are regions that would match a switch schema in Table 3 but contain extra edges. 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 $\mathsf{IncSwitch}[\cdot]$ box in Figure 4 would not be identified as an $\mathsf{IncSwitch}[\cdot]$ region because there is an extra incoming edge to the default case node.

A switch candidate is refined by first virtualizing incoming edges to any node other than the switch head. The next step is to ensure there is a single successor of all nodes in the switch. The immediate post-dominator of the switch head is selected as the successor if it is the successor of any of the case nodes. Otherwise, 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 is chosen as the successor. After the successor has been identified, any outgoing edge from the switch that does not go to the successor is virtualized.

After refinement, a switch candidate is usually collapsed to a $\mathsf{IncSwitch}[\cdot]$ 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
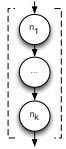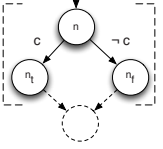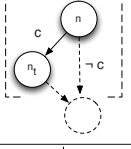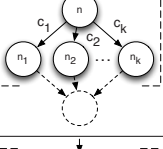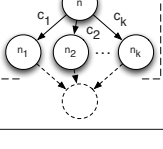
Table 3: Acyclic regions.

	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 are pairwise disjoint and satisfy $\bigvee_{i \in [1,k]} c_i \neq true$.
	Switch$[n, (c_1,n_1), \cdots, (c_k,n_k)]$: Complete switch region. The outgoing conditions are pairwise disjoint and satisfy $\bigvee_{i \in [1,k]} c_i = true$.

Figure 4: Complete and incomplete switches.

	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 be virtualized during refinement to match this schema.

Table 4: Cyclic regions.

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$[\cdot]$. However, because the default node handles all other cases, together they constitute a Switch$[\cdot]$.

3.5 Cyclic Regions

If the subgraph at node n is cyclic, the algorithm tries to match a loop at n to one of the cyclic loop patterns. 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. Distinct loops at node n can be identified 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$[\cdot]$ loops test the exit condition before executing the loop body, whereas DoWhile$[\cdot]$ 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 nat-
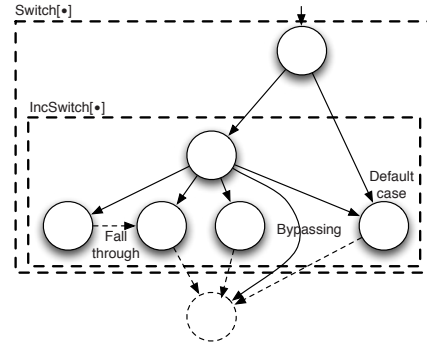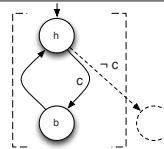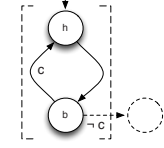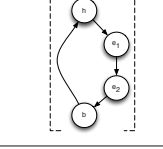
ural loop. Natural loops do not represent one particular C looping construct, but can be caused by code such as

```
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$[\cdot]$ regions are decompiled to

$$\text{while (1) } \{\ldots\},$$

which has no exits, the body of the loop must trigger any loop exits. In Phoenix, the loop exits are represented by 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, loop refinement begins. Cyclic regions may fail to match loop schemas because (1) there

```
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  }
```

```
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  }
```

```
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  }
```

(a) Original source code

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

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

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

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, the one with the most incoming edges is selected, and incoming edges to the other entrances are virtualized.

The next step is to identify the type of loop. If there is an exit edge from the loop header, the loop is a While[·] candidate. If there is an outgoing edge from the source of the loop's back edge (see §2.1), it is a DoWhile[·] candidate. Otherwise, any exit edge is selected and the loop is considered a NatLoop[·] candidate. The exit edge determines the successor of the loop, i.e., the statement that is executed immediately after the loop. The 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 to the loop header use the `continue` tail regions, while edges 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 [32]. 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, a `break` statement 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, Phoenix outputs gotos, as in Figure 5(c). In our evaluation (§4), we show that enabling the new loop membership definition decreased the numbers of gotos Phoenix emitted by 45% (73 to 40).

The last loop refinement step is to remove edges that may prevent the loop body from being collapsed. This can happen, for instance, when 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.

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, the edges that remain reflect more structure.

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 `gotos`. This corresponds to recovering $30\times$ more structure (40 gotos vs 1,229) than structural analysis without iterative refinement.

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 consisted of 29,652 lines of code before our additions. We measured the number of lines of code using 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 tests to measure correctness.

- **Structuredness.** Recovering control flow structure helps program analysis and humans alike. Structured code is easier for programmers to understand [19], and helps scale program analysis in general [32]. 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., $1 - (\text{LOC decompiled}/\text{LOC assembly})$ [13]. 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 side-steps 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 [11, 13, 21, 22, 39]. Previous work that does measure correctness [20, 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 Overview

We tested Phoenix on the GNU `coreutils` 8.17 suite of utilities. `coreutils` consists of 107^2 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 an order of magnitude larger than any other systematic end-to-end decompiler evaluation in which specific metrics were defined and measured.

Tested Decompilers In addition to Phoenix, we tested the latest publicly available version of the academic decompiler Boomerang [39] and Hex-Rays [23], 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 [35], DISC [26], and *dcc* [13]. However, these compilers either produced pseudo-code (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).

---

[2]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.

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. 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`[3]. 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 Coreutils Experiment 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 orig-

inal source code and decompiled source code. We implemented the substitution method by using CIL [33] 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 failure modes 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 tests, or fail. We do not count the number of failed 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 test. Thus, it would be

---

[3]Although it seems like this should be possible to implement using only a C header file containing some `typedef`s, a `typedef` has its qualifiers fixed. For instance, `typedef int t` is equivalent to `typedef signed int t`, and thus the type `unsigned t` is not allowed because `unsigned signed int` is contradictory.
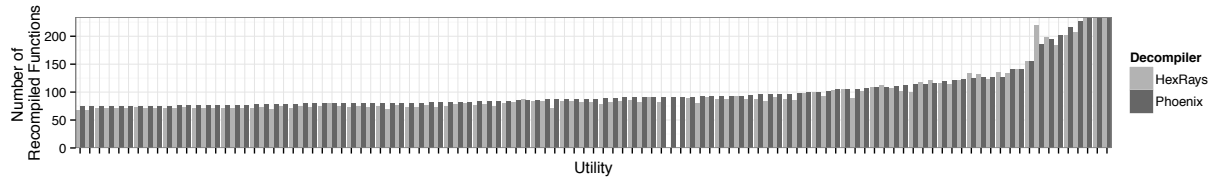
Figure 6: The number of functions that were decompiled and recompiled by each decompiler, broken down by utility. Hex-Rays failed on two utilities for unknown reasons.

	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 measurements for the `coreutils` experiment. These results includes two utilities for which Hex-Rays recompiled zero functions (thus trivially passing correctness).

misleading to conclude that a recompiled program performed well by "only" failing one test.

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 tests. Excluding those two utilities, Hex-Rays only compiled 26 utilities that passed the 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 an observable 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.

	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 measurements for the `coreutils` experiment. The statistics only reflect the 8,676 recompilable functions output by both decompilers.

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 [20, 32, 36] will recover less structure. The results also suggest that Hex-Rays employs a technique similar to iterative refinement.

5 Limitations and Future Work

5.1 BAP Failures

Phoenix uses BAP [10] 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 analy-
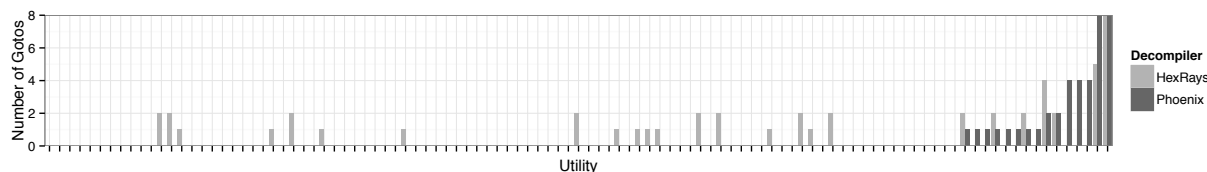
Figure 7: The number of gotos emitted by each decompiler, broken down by utility. Only functions that were decompiled and recompiled by both decompilers are counted.

sis (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 is necessary to know the type of *g* before calling it.

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 `f` takes a pointer to an integer, and a function calls `f(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 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 interpro-

cedural 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 s {int a; struct s *next;}
```

will be inferred as

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

which does not specify what type of element `next` points to. Since Phoenix is intended to be the front-end 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 and structural analysis. 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 [39, ch. 2]. Another in-depth overview is available online [18].

Modern decompilers typically trace their roots in Cifuentes' 1994 thesis on *dcc* [13], a decompiler for 80286 to C. The structuring algorithm used in *dcc* is based on interval analysis [2]. Cifuentes proposed the compression ratio metric (see §4.2), but did not measure correctness on the ten programs that *dcc* was tested on [14]. 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 [16]. Again, they did not test the correctness of the decompilation output. Cifuentes [13] pioneered the technique of recovering short-circuit evaluations in compound expressions (e.g., x && (!y || z) in C).

Chang et al. [11] 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. [39]. 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 [37], 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 [23] 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 [35] 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 [2] and Cooke [17] in the 1970's and commonly known as "interval analysis". Sharir [36] 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. [20] 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) { body; break; } appears in a loop, then the statements represented by body would belong to a tail region. Engel et al. have extensively tested their implementation of SESS in a source-to-source compiler. However, their SESS analysis does not use iterative refinement, and can get stuck on unstructured code. We show in our evaluation that this leads to a large amount of structure being missed. Their exact algorithm for detecting tail regions is also left unspecified [20, Algorithm 2, Line 15].

Another line of related work lies in the area of program schematology, of which "Go To Statement Considered Harmful" by Dijkstra [19] is the most famous. Besides the theoretical study of the expressive power of goto vs. high-level control statements (see, e.g., [34]), this area is also concerned with the automatic structuring of (unstructured) programs, such as the algorithm by Baker [3].

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 [38] recover types from dynamic traces. Other work has focused on C++-specific issues, such as virtual table recovery [21, 22].

7   Conclusion

We presented Phoenix, a new binary-to-C decompiler designed to accurately 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 uses iteratively refinement to recover more control flow structure than existing algorithms. We evaluated Phoenix and the *de facto* industry standard decompiler, Hex-Rays, on correctness and amount of control flow structure recovered. Phoenix decompiled twice as many utilities correctly as Hex-Rays, and recovered more structure.

Acknowledgments

References

[1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

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

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

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

[5] Sebastien Bardin, Philippe Herrmann, and Franck Vedrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 54–69. Springer, 2011.

[6] 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. *Communications of the ACM*, 53(2):66–75, 2010.

[7] The BitBlaze Binary Analysis Platform. http://bitblaze.cs.berkeley.edu, 2007.

[8] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2011.

[9] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2007.

[10] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[11] Bor-yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of Low-Level Code Using Cooperating Decompilers. In *Proceedings of the 13th International Symposium on Static Analysis*, pages 318–335, 2006.

[12] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 39–50, 2008.

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

[14] Cristina Cifuentes. Interprocedural Data Flow Decompilation. *Journal of Programming Languages*, 4(2):77–99, 1996.

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

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

[17] John Cocke. Global Common Subexpression Elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, pages 20–24, 1970.

[18] The Decompilation Wiki. http://www.program-transformation.org/Transform/DeCompilation. Page checked 6/25/2013.

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

[20] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced Structural Analysis for C Code Reconstruction from IR Code. In *Proceedings of the 14th International Workshop*

on Software and Compilers for Embedded Systems, pages 21–27. ACM, 2011.

[21] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[22] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 240–243. IEEE, 2010.

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

[24] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[25] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

[26] Satish Kumar. DISC: Decompiler for TurboC. http://www.debugmode.com/dcompile/disc.htm. Page checked 6/25/2013.

[27] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, 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*. The Internet Society, 2011.

[29] 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*. The Internet Society, 2010.

[30] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 111–127. Springer, 2002.

[31] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[32] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[33] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[34] 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.

[35] REC Studio 4—Reverse Engineering Compiler. http://www.backerstreet.com/rec/rec.htm. Page checked 6/25/2013.

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

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

[38] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011.

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

[40] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 163–177, 2012.