Static Decompilation of Unobfuscated Native Code

Edward L. Sullivan
A53045017

Abstract

Native code decompilation is the process of transforming the low-level binary code of a program into a high-level representation such as source code or pseudocode, often for the purposes of porting applications to new platforms, reverse engineering malware, or checking the correctness of compilers. First, I summarize the static decompilation process, the main challenges that make it difficult, and the existing population of commercial and academic tools. Second, I describe mature techniques to tackle the major subtasks of vanilla decompilation, including disassembly, idiom detection, data abstraction recovery, and control flow structuring. Here I also highlight a major subgoal of decompilers, namely retargetability, and contrast the opposing approaches of a C++ specific decompiler and a fully-retargetable decompiler. Third, I summarize a few highly experimental research areas that attempt to infer more information than traditional decompilation by applying AI and deprogramming strategies. Fourth, I consider the main countermeasures to decompilation and discuss alternative techniques that may outperform or augment decompilation. I conclude by mentioning several gaps in the existing literature and suggesting the most promising areas for future work.

1 Introduction

The manual analysis of binary or assembly code by hand is an extremely difficult, tedious, and error-prone process which requires expert skill and meticulous knowledge of assembly language. As an automated tool, decompilers overcome many of the aforementioned challenges and lower the barrier to entry into software reverse engineering by attempting to recreate source code from binary code, as illustrated in Figure 1. In terms of speed, modern decompilers such as Hex-Rays are capable of decompiling several megabytes of binary code in a few minutes. In terms of skill threshold, decompilers produce high-level source code or pseudo-code so that the human analyst does not need a mastery of assembly language. In terms of inaccuracies, decompilers are not susceptible to accidental mistakes, yet as with manual analysis, decompilers suffer mistakes caused by the inherent conceptual complexity of decompilation.

1.1 Applications of Decompilation

Decompilers are useful in several diverse situations, some honest and some nefarious. To begin, let us consider scenarios in which the original source code is available. A software engineer might use a decompiler to verify that the output of an undependable compiler is correct. Similarly, a compiler designer might use a decompiler to study the effects of a new optimization technique.

Decompilation techniques are often valuable in tools other than decompilers. [55] created a source-to-source transformer which removes *goto* statements from source code and outputs functionally equivalent structured source code. [23] created a combined editor, decompiler, compiler and language for which there is never a source code file, but instead the source code is rendered in the editor after being decompiled from the executable file. Such a novel tool could be valuable for several applications: embedded systems that are device programmable and which have small persistent storage could benefit from not needing a source file; debuggers could more accurately display synchronized executable and source code code; open source projects would only need to publish a single file instead of both the source and executable files.

The primary application of decompilers is source code recovery, which is often a large component of reverse engineering. The source code for a particular program might be inaccessible either because it is the property of another entity or because it has accidentally become lost or deleted.

Potential reasons to recover source code include: porting an executable to a different platform, performing a security audit of untrusted third-party software, maintaining a lost codebase when rewriting code would be too slow or costly, learning the internal details of a com-

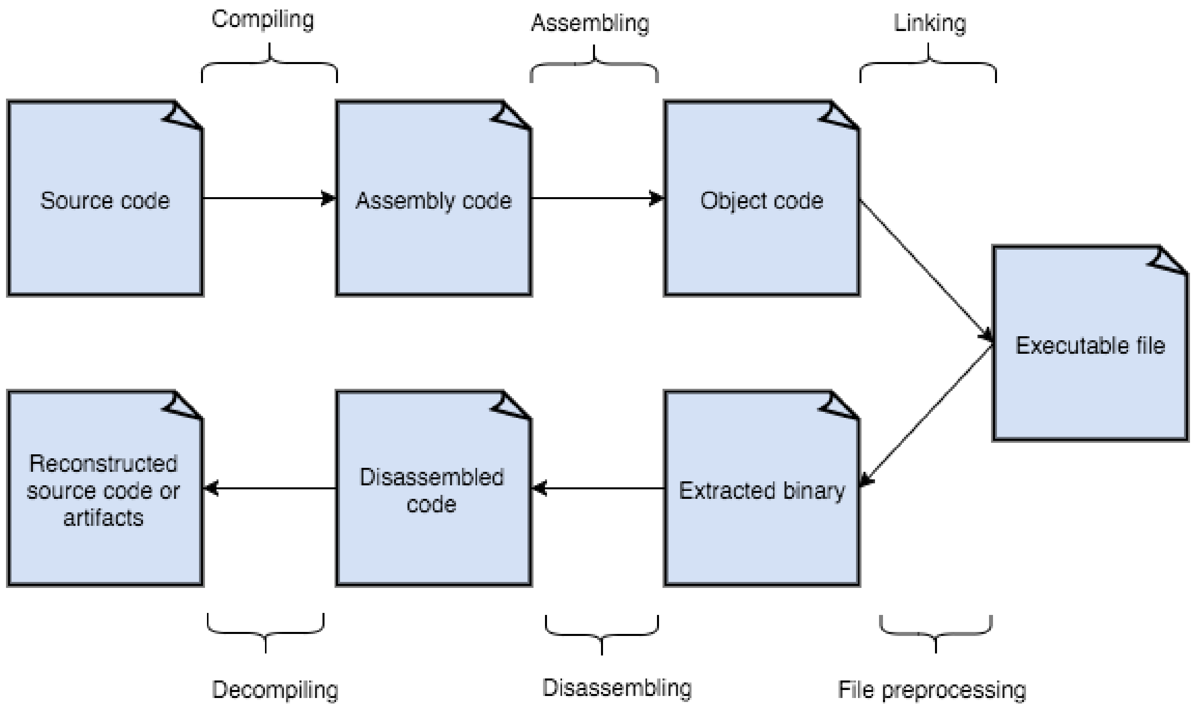


Figure 1: The life stages of code with respect to reverse engineering

petitor's algorithm, searching for zero-day bugs to exploit in commercial-off-the-shelf software, investigating an unpublished API or ABI, maliciously modifying and replacing firmware, studying an unpublished file format or network protocol, penetration testing of critical infrastructure, etc. [61] present a case study in which they used their Boomerang decompiler to help a client who had bought the rights to a Windows application for speech analysis. The client, who was primarily interested in recovering the algorithm, had the source code for an earlier prototype of the application, but lacked the source code for the most recent version.

Another big motivation for recovering source code is that the population of source code analysis tools is much richer than the population of binary analysis tools. Popular source-level static code analysis tools (SCAT) for bug finding and program understanding include Codesurfer [9], Codesonar [33], Coverity [15], Klocwork [2], Parasoft [3], Cast [1], FindBugs [11], PMD [4], KINT [66], and RICH [17]. Corresponding binary analysis tools are uncommon. Furthermore, many source level analyzes are faster than their binary counterparts.

1.2 Properties and Goals of Decompilers

The specific goals of a decompiler depend on its intended application, but the most common reoccurring goals, in terms of the decompiler's output, include functional equivalence (correctness), semantic equivalence, and readability, in no particular order. For the internal design of a decompiler, designers aim for modularity and retargetability. Consider a decompiler designed for helping engineers port applications to new hardware platforms. At most, the engineer will need to make a few small changes to the decompiled code for compatibility purposes before recompiling it for the new platform. However, the engineer will not need to read or edit large swaths of code to understand or alter its algorithms. In this case, correctness and functional equivalence of the output would be a higher priority than readability and semantic equivalence. In contrast, for a decompiler aimed at reverse engineering another company's application, readability and semantic equivalence would matter most. For malware analysis, retargetability would be crucial given the diverse and expanding population of devices targeted by attackers.

1.3 Is Decompilation Possible?

In a pessimistic simile, [22] complains that decompilation is "about as easy as reconstructing a pig from a sausage." Supporting that claim, [18] lists many obstacles that make decompilation difficult. First and foremost, the challenge of separating code from data in a von Neumann architecture, "even in programs that do not allow such practices as self-modifying code, is equivalent to the halting problem, since it is unknown in general whether a particular instruction will be executed or not (e.g. consider the code following a loop)." Other decompilation hurdles include: handling self-modifying code, unravelling packed or hidden malware, accounting for architecture-dependent behavior such as instruction prefetching, and filtering out start-up code added by the compiler or linker. Even if the program has already been perfectly disassembled, there are still many challenges created by the compilation process: loss of identifiers for variables and functions, loss of variables replaced by operations of registers and memory, loss of types, loss of explicit functions, loss of high-level control flow constructs replaced by conditional and unconditional jumps, and introduction of idioms.

However, [35] takes a more optimistic viewpoint of decompilation, explaining that "compilation preserves program behavior, which would mean that a sausage could grunt and run." Specifically, [35] highlights that, although a completely automated decompiler might not be possible in all cases, there are techniques that achieve at least approximate solutions for a large class of practical input programs. Next the authors define a subclass of practical input programs to target and declare several assumptions about said subclass, including that the programs are not obfuscated, are not self-modifying, were compiled from C source code, and strictly conform to the C standard, which means that the runtime behaviors of the programs never depend on undefined or implementation-specific details (e.g. the result of assigning a pointer to an integer variable is not officially defined in the C standard). Finally, the authors describe several methods, implemented in their TyDec decompiler, that work well for the aforementioned restricted set of input programs.

1.4 Design Overview

As shown in Figure 2, static decompilers are generally designed in three sequential components: a platform-specific front-end, a middle-end that operates on an internal representation (IR) of the program, and a back-end that outputs pseudo-code or code for a specific language. The front-end typically extracts the relevant binary code from the input executable file, disassembles the binary into assembly code, and then converts the assembly code to a internal representation. The specific internal representation varies by decompiler, but usually it is a platform independent intermediate language, possibly incorporating a control flow graph and possibly adhering to Static Single Assignment (SSA) form. The middle end is the meat of the decompiler. It applies "abstraction recovery mechanisms" such as structuring of control flow, recovery of variables and types, function identification, function parameter identification, idiom detection, etc. Finally, the back-end attempts to generate code, usually in the C language.

1.5 Existing Decompilers

Maurice Halstead, of Navy Electronic Labs, is credited for creating the first decompiler, D-Neliac, in 1960 [30]. D-Neliac took as input binary programs for the IBM 7094 and output Neliac source code suitable for the Univac 1108. Ultimately, D-Neliac demonstrated the practicality of decompilation.

The dcc decompiler [18], created as graduate work by Cristina Cifuentes in the early 1990s, served as foundational inspiration for all modern research in decompilation, but has long since been abandoned.

Hex-Rays [27] is currently the most popular commercial decompiler and it is based on IDA Pro [28], an interactive disassembler made by the same company. Hex-Rays also supports plugins and an interactive debugger tool. Both tools are closed-source, however, software pirates have previously used Hex-Rays to circumvent the DRM of Hex-Rays itself. Most academic publications use Hex-Rays as a point of comparison.

Many of the best modern decompilation tools remain unpublished and are only referred to in academic research papers. Phoenix [16] and DREAM [70] are purportedly the most effective tools for control flow structuring (§2.4). TIE [42] and REWARDS [44] are purportedly the most effective tools for data abstraction recovery (§2.3). Additionally, many government-sponsored or private reverse engineering teams likely have their own secret home-brewed decompilation tools.

The RetDec decompiler [39], for which an online browser interface is available, is purportedly the best decompiler in terms of retargetability (§2.5). In contrast, the open-source SmartDec [26] tool narrowly focuses on decompiling and outputting C++ code (§2.5.2).

A other few open-source decompilers, such as Boomerang [61], are available, but none of them have strong reputations.

[71] discusses a technique for dynamic decompilation that is resilient to packing and disassembly-level obfuscation techniques. Please see §4 for a discussion of tools, such as unpackers, emulators, taint analyzers, etc., that may augment the static decompilation process or overcome obfuscations.

Bytecode decompilation is also an area of research, but it is sufficiently different from native-code decompilation that we have left it out of this paper.

2 Major Subtasks of Decompilation

2.1 Disassembly

Disassembly is the process that converts binary code to assembly code and it is an essential prerequisite to decompilation. On the surface, disassembly may appear to be a trivially straightforward process because there is a one-to-one mapping between binary opcodes and assem-

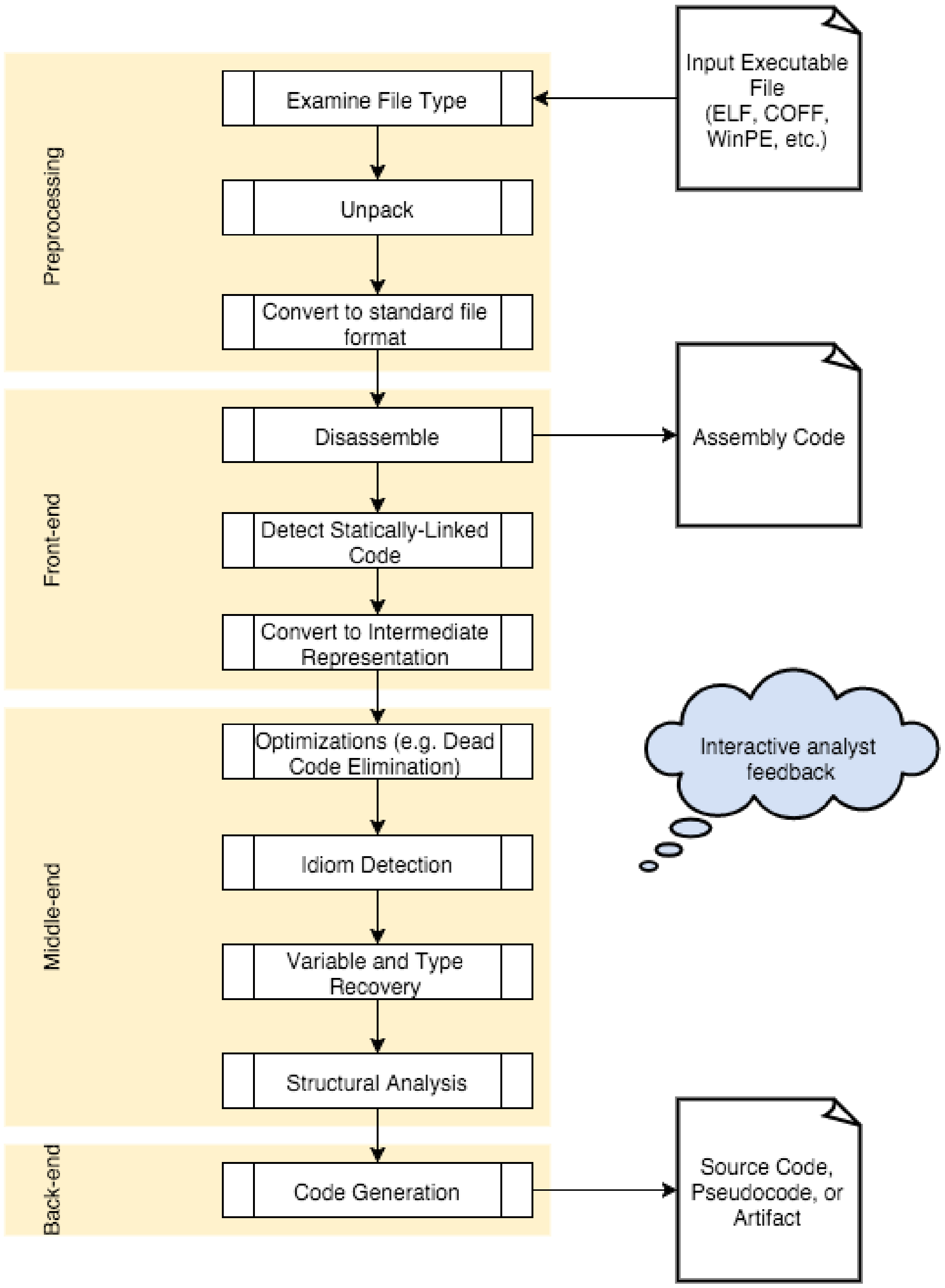

Preprocessing
Examine File Type
Unpack
Convert to standard file format
Input Executable File (ELF, COFF, WinPE, etc.)
Front-end
Disassemble
Assembly Code
Detect Statically-Linked Code
Convert to Intermediate Representation
Middle-end
Optimizations (e.g. Dead Code Elimination)
Interactive analyst feedback
Idiom Detection
Variable and Type Recovery
Structural Analysis
Back-end
Code Generation
Source Code, Pseudocode, or Artifact

Figure 2: Typical design of a decompiler. Some decompilers may have fewer stages, additional stages, or re-ordered stages.

bly instruction mnemonics. However, the disassembly process is complicated by the fact that assembly instructions are not necessarily located contiguously in the binary file. In fact, on processors designed with the Von Neumann architecture, including the Intel Pentium series and 80x86 series, data and code share the same address space and can be intermixed within the binary program.

```
A:
    jmp     B
    db      0xAA
B:
    mov     eax, C
    add     eax, 4
    jmp     eax
C:
    mov     ebx, 10
    add     ebx, 4
```

Listing 1: Sample code that defeats basic linear-sweep disassembly and basic graph traversal strategies.

The snippet of assembly code in Listing 1 demonstrates this potential interweaving of data and code: the data byte 0*xAA* is located directly between a *jmp* instruction and a *mov* instruction. In this scenario, the simplest strategy for disassembly, a linear sweep of the program until a non-instruction is found, would fail to correctly disassemble the program because it would treat the byte 0*xAA* as the start of an instruction after correctly decoding the jump instruction. The linear sweep strategy is available in basic disassembly tools such objdump.

A more sophisticated disassembly strategy is needed to overcome the difficulty of interweaved data and code. In particular, modern disassemblers such as IDA Pro perform a linear sweep, but instead of always proceeding sequentially, they follow each unconditional jump and they follow both branches of conditional jumps. Therefore, when disassembling the program in Listing 1, the byte of data 0*xAA* would never be reached and hence it would never be mistaken for an instruction. An additional benefit of this traversal process is that it creates the control flow graph (CFG) of the program. Unfortunately, even this graph traversal strategy has a weakness: indirect jumps.

Assembly code jump instructions usually have one operand, namely the target address. Whereas a direct jump specifies a numerically hardcoded target address, indirect jumps specify a register or a memory cell (to be dereferenced) that will hold the target address at runtime. Therefore, it is difficult to determine the target address of an indirect jump during static analysis because the runtime values of registers or memory cells may be hard to predict without actually executing the code. The x86 *ret* instruction, which pops an address off the stack and unconditionally jumps to it, is one of the most common examples of an indirect jump. Indirect jumps are also commonly used to implement *switch* structures and function pointers.

Many modern disassemblers that apply the aforementioned CFG traversal strategy do not attempt to resolve the targets of indirect jumps, or they make unprincipled heuristic-based guesses about potential target locations. For example, IDA Pro makes the assumption that all *ret* instructions actually return to the caller function, which might not be true in obfuscated malware [37]. Thus the example in Listing 1 might stifle some rudimentary disassemblers that are unable to resolve the target of the indirect jump.

To overcome the challenge of indirect jumps, [37] developed an "iterative disassembler" called Jakstab. Jakstab applies the previously mentioned CFG traversal strategy, but it resolves indirect jumps by performing a dataflow analysis on the partially constructed CFG. After resolving an indirect jump, Jakstab resumes the CFG traversal. Thus, by iteratively switching between CFG traversal and dataflow analysis, Jakstab is able to resolve more indirect jumps and disassemble more code than previous tools. Testing on commercial software indicates that Jakstab dramatically outperforms IDA Pro.

In summary, disassembling is still an active area of research and one of the remaining weak-points in the decompilation toolchain. §4.3 discusses many obfuscation techniques that dramatically increase the difficulty of disassembling, as well as countermeasures to overcome those obfuscations. [65] discusses the need for "reassemblable disassembly" and proposes a strategy to make the output code of the disassembler relocatable.

2.2 Instruction Idiom Detection

Idioms are sequences of assembly instructions that implement a small high-level-language operation and that have been optimized with the unintended side effect that their high-level purpose or meaning is no long readily apparent from the code. The most common idiom example is the use of the XOR instruction to set a register's value to 0, rather than explicitly assigning the immediate value 0 to the register.

Idioms are used for several reasons: optimizing for program speed, optimizing for program size, or allowing floating-point instructions to be emulated in software with integer instructions when no floating-point hardware is available. Idiom usage may vary by compiler, optimization setting of the compiler, and target platform.

Unfortunately, idioms make decompilation and program comprehension more difficult because, as with natural language idioms, their figurative meaning or ultimate purpose might not be obvious from looking at the literal meaning of their comprised instructions. Therefore, decompilers often attempt to detect and replace idioms with more intuitive statements. The idiom detection process can be difficult, however, because compilers frequently reorder instructions, potentially placing an unrelated instruction between two instructions of the same idiom, in an attempt to maximize CPU pipeline utilization. Likewise, compilers may sometimes spread idioms across basic blocks.

[39] has implemented the most advanced idiom-detection algorithm to-date as part of their fully-retargetable decompiler RetDec, discussed in §2.5.1. To overcome the aforementioned position-dependence challenges, namely the reordering and spreading of instructions across basic blocks, their algorithm sequentially inspects every basic block and treats each instructions "as a possible root of a derivation tree containing one particular instruction idiom." Furthermore, for efficiency purposes, their algorithm operates on the program at a middle-end internal representation, after dead-code elimination has been performed and platform-specific opera-

tions have been removed.

2.3 Data Abstraction Recovery

A high-level language like C usually allows a programmer to create an unrestricted number of variables with names and types, both primitive and composite, either on the stack (local variables), heap (dynamically allocated variables), or data segment (global, constant, and static variables). However, during the compilation process, the abstraction of typed variables is lost and replaced with operations on memory cells and a finite number of registers.

At first glance, little information can be gleaned from the disassembly. For example, unless the original source code was compiled with debugging information, variable names are lost. Within a function, the number of local variables is not obvious, especially because multiple variables can be stored at the same location on the stack if they have disjoint liveness ranges. Additionally, it is not obvious whether a 32-bit register or spot on the stack corresponds to a signed integer, an unsigned integer, or a pointer. Composite data types further complicate the analysis because their internal fields are represented in memory in a manner indistinguishable from a group of primitive variables.

Despite these difficulties, most decompilers include an analysis stage for data abstraction reverse engineering. Identification of variables and their types facilitates manual understanding of the code and it enables the decompiled source code to be recompiled with more accuracy.

2.3.1 Variable Recovery

The first stage of data abstraction reconstruction is variable recovery, which attempts to discover the existence and location (aliases) of source-level variables by analyzing the disassembly. The common strategy is to examine memory access patters. For example, we might infer that a function has two parameters and one local variable because it accesses data at two distinct positive offsets and one negative offset from the stack frame's base pointer.

For a more detailed explanation of variable recovery, consider TIE [42], the most advanced system in the literature for variable and type recovery. TIE's first step is to convert the program, initially in an intermediate representation similar to assembly language, into static single assignment (SSA) form such that each assignment to a register variable defines a new register variable and each store to memory defines a new memory context. Using SSA ensures that different uses of the same register or memory slot are treated independently, as they may correspond to separate variables with different types in the source program. Next, TIE applies a variable recovery algorithm that the authors call DVSA, possibly named for its similarity to Value Set Analysis (VSA) [13]. DVSA outputs a conservative list of variable memory locations, including aliases, although the authors do not fully explain the algorithm.

2.3.2 Type Recovery

The second and more difficult stage of data abstraction reconstruction is type recovery, which attempts to determine the types of variables based on how they are used. For example, if two 32-bit registers are the operands to an integer *add* operation, and we already know that one of the operand registers corresponds to a pointer variable, then we can infer that the other operand register must correspond to an integer (since it does not make sense to add two pointers). Another strategy for inferring type is studying the variables that are used to store the arguments and return value of system calls and calls to standard imported library functions with known prototypes.

There are two competing approaches to type recovery: static and dynamic.

The authors of REWARDS [44], one of the most recent type-inference tools, took a dynamic approach because they anticipated that accounting for control flow–a necessity for static type inference–would have been too challenging. Unfortunately, dynamic approaches typically suffer from limited program coverage, especially since they cannot easily merge the results of multiple different traces without accounting for control flow analysis. Furthermore, REWARDS may produce contradictory results when executing the same program with different inputs.

Until recently, most static approaches relied upon imprecise heuristics and produced inaccurate results. Generally, these tools "employ[ed] some knowledge about well-known function prototypes to infer parameters, and then use[d] proprietary heuristics that seem[ed] to guess the type of remaining variables such as locals" [42].

The TIE (Type Inference on Executables) tool, developed by [42], is the first principled approach to static type inference and it dramatically outperforms all existing systems, including REWARDS. The authors of TIE define two criteria to quantitatively evaluate type inference tools. First, 'conservativeness' ensures that the tool does not guess or infer any type information beyond what is concretely supported by the binary. Secondly, 'precision' ensures that the tool provides as detailed type information as possible. To achieve these two goals, TIE outputs a bounded range of possible types for each variable, rather than reporting a single type for each variable, as most other tools do.

TIE uses a lattice system to represent the subtype relationships between types. For example, $unsigned32-bitint$ and $signed32-bitint$ are both subtypes of the less-precised type $32-bitnumber$. The tool works by first generating type constraints for each variable, based on how they are used in every instruction and function call,

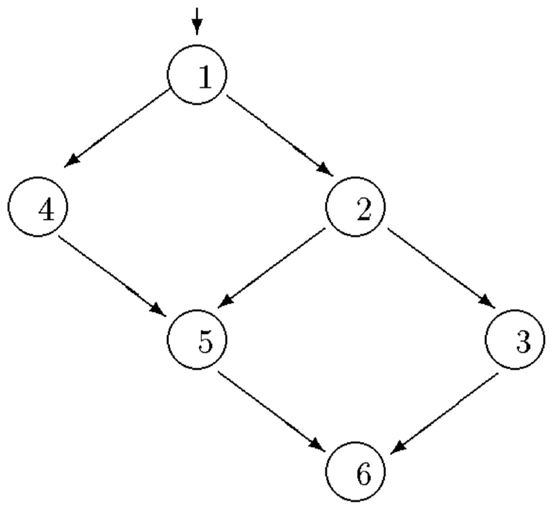


Figure 4: The control flow graph of an example proper region. Image borrowed from [18].

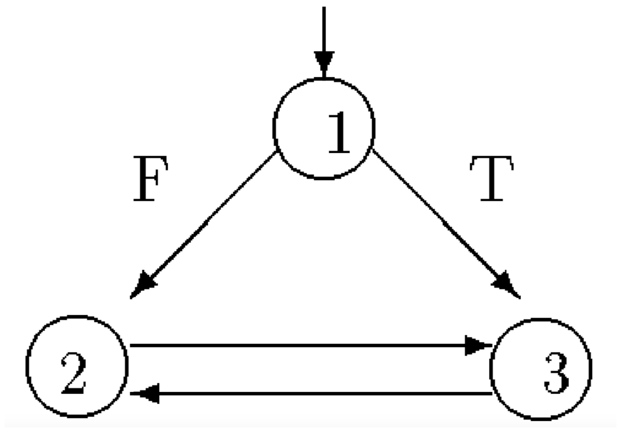


Figure 5: The control flow graph of an example improper region. Image borrowed from [18].

and then solving the constraint system.

2.4 Control Flow Structuring

High-level languages like C usually support several control structures such as two-way conditionals ($if-then-else$), single branching conditionals ($if-then$), pre-tested loops (*while*), post-tested loops ($do-while$), multi-way branch statements (*switch*), etc. Most languages also allow unstructured jumps (*goto*). Most assembly languages, in contrast, only allow conditional jumps and unconditional jumps. A naive approach to decompilation would simply convert all of these jumps into *goto* statements, resulting in valid and functionally correct source code. However, as we will see, there are several reasons why it is useful to infer higher-level control constructs from the disassembly. This structure recovery process is known as control flow structuring and it is a key component of most decompilers.

Figure 3, borrowed from [18], informally defines the class of DRECn structures, named after its grammar: Do-while, Repeat-End, Cyclic, and n nested infinite loops. C code written without *goto* statements is “structured” or “compositional,” meaning that the resulting control flow graph can be decomposed into DRECn structured subgraphs, each containing a single entry and possibly multiple exits to a common successor node. In terms of interval theory, the control flow graph of structured C code is “reducible.”

On the other hand, the control flow graphs in Figures 4 and 5 are unstructured and “irreducible,” ultimately meaning that there are no high-level C constructs (i.e., DRECn subgraphs) capable of representing their structure without the help of *goto* statements. Irreducible acyclic graphs are termed “proper regions,” while irreducible cyclic graphs are termed “improper regions.” The name “proper region” may be slightly misleading because, for the purposes of control flow structuring, proper regions are just as problematic as improper regions.

Note that the use of *goto* statements in the source code does not guarantee that the resulting control flow graph will be unstructured–if the *gotos* are used in structured ways, never jumping to the middle of another structure, then the resulting graph will be structured.

Unfortunately, some binary code corresponds to unstructured, irreducible control flow graphs. In these cases, only partial structure can be recovered and the decompiled output code will have to include one or more *goto* statements. There are several common culprits that create this unstructured code. First and probably most common, many compiler optimizations involve a technique called code motion that may alter the control flow. Second, authors of malware and propriety software may intentionally obfuscate their code with abnormal control flow to make it resilient against reverse engineering attempts. Third, programmers occasionally use *goto* statements intentionally. There are a few rare situations where a *goto* statement may actually make source code more readable. For instance, a programmer might use a *goto* statement to mimic a high-level control structure not available in C, such as a multi-level loop exit. Finally, some benchmarks use *goto* statements.

Since the decompiler usually cannot make assumptions about whether any unstructured jumps have been incorporated into the input program, either by the original programmer, an obfuscator, or the compiler, the decompiler must be prepared to handle arbitrary unstructured code. Thus the decompiler’s output language must allow *goto* statements, even though most decompilers aim to use as few *goto* statements as possible.

There are two main reasons why recovering structure

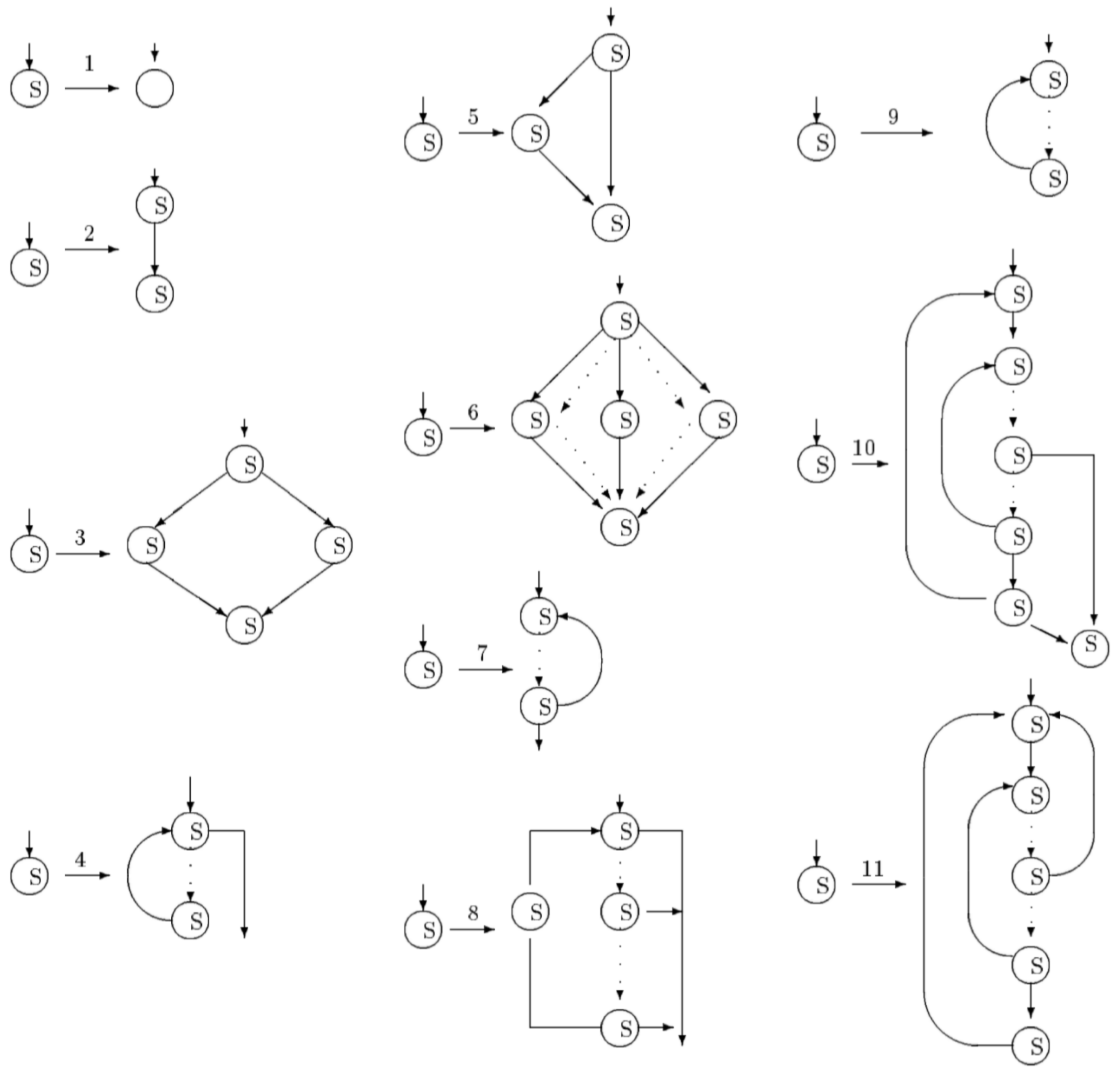


Figure 3: Informal grammar of DRECn constructs borrowed from [18].

is valuable. First, manual analysis of the code is dramatically easier. Second, structured code is required or helpful for several automated code analysis tools. Many parallelizing and optimizing compilers rely upon structured strategies when performing the data flow analysis. For instance, transforming a program to Static Single Assignment form, which is often used for dead code elimination, is faster and more straightforward if the code is structured. Additionally, many tree-based representations of a program, such as Abstract Syntax Trees, require the program to be structured. In the realm of computer security, source code vulnerability scanning tools frequently rely upon high-level syntax to detect potential buffer offerflow vulnerabilities [16].

```
if (cond) goto L1;
statement_1;
...
statement_i;
while (expr) {
    statement_j;
    ...
L1: ...
    statement_n;
}
```

Listing 2: A snippet of unstructured code with a goto statement whose target label is inside a loop below the goto statement. Borrowed from [23].

```
goto_L1 = cond;
if (!goto_L1) {
    statement_1;
    ...
    statement_i;
}
while (goto_L1 || expr) {
    if (goto_L1) goto L1;
    statement_j;
    ...
L1  goto_L1 = false;
    ...
    statement_n;
}
```

Listing 3: The code from Listing 2 modified in a functionally equivalent way so that the goto statement has been relocated into the same loop as its target label. Borrowed from [23].

```
goto_L1 = cond;
if (!goto_L1) {
    statement_1;
    ...
    statement_i;
}
while (goto_L1 || expr) {
    if (!goto_L1) {
        statement_j;
        ...
    }
L1  goto_L1 = false;
    ...
    statement_n;
}
```

Listing 4: The code from Listing 2 modified in a functionally equivalent way so that the goto statement has been eliminated. Note that the code in Listing 3 was an intermediate step in this transformation.

Approaches to control flow structuring fall into two categories: those which alter the code semantics and those which do not. Within the first category are techniques which may change the underlying design of the code, despite producing functionally-equivalent source code.

For instance, [23] introduced a simple technique to reliably remove all *goto* statements from C source code, but their technique involved creating several new Boolean variables and rearranging basic blocks of code. Consider the unstructured C code in Listing 2 which has a *goto* statement whose target label is inside of a loop. Listing 3 shows an intermediate transformation in which the *goto* statement has been moved into the body of the same loop. Finally Listing 4 shows the ultimate transformation after the *goto* statement has been fully eliminated.

Note that all three snippets of code are functionally equivalent, but only the code in Listing 4 is structured. Unfortunately, their structured output code in Listing 4 is arguably more complex and difficult to read than the original code with a *goto* statement in Listing 2. Another drawback of their technique is that it slightly worsens the runtime performance by up to 20% in benchmarks with many *goto* statements.

Another semantic-altering strategy for overcoming unstructuredness in the control flow graph is node splitting (a.k.a, code replication) [69][8], which, as demonstrated in Figure 6, increases the length of the code. [49] provides such an algorithm guaranteed to structure any program without needing to alter any predicates or create compound predicates. In their analysis, [49] observes that the structured version of the program is always worse than the original version in terms of space and time complexity. This observation leads the authors to propose using the measured growth in runtime requirements as a quantitative metric for the structural complexity of the input program. Ultimately, the authors include a sobering disclaimer that, while "the transforms developed ... might help to unravel some knotty problems, ... they cannot produce logical poetry from tangled nonsense."

Additionally, some decompiler authors have tried to evade *goto* statements by introducing new high-level control flow constructs that are not available in the decompiler's output language, usually C. Given the fundamental ways in which these semantic-altering strategies change the code and its apparent meaning, they are generally not suitable for decompilers aimed at reverse engineering.

Instead, most modern decompilers apply graph structuring algorithms that preserve the semantics of the program and do not involve inserting additional variables, replicating code, or introducing new control flow constructs. The most foundational work in this field is Cristina Cifuentes' thesis [18], which served as a guideline for many later decompiler designers and which outlines a structuring technique called interval analysis. However, most recent research focuses on an algorithm called "structural analysis," which is a more effective and efficient form of interval analysis.

Figure 7 provides a straightforward example of the structural analysis algorithm applied to a simple control flow graph. The basic idea of structural analysis [47] is traverse the control flow graph in reverse post-order, which is different from pre-order. During the traversal, the algorithm checks to see if the current node is the head of a subgraph that matches the pattern of a defined control construct. There are two categories of defined control constructs: acyclic (sequence, if-then-else, if-then, incomplete switch, complete switch) and cyclic (while, do-while, infinite/natural). For a particular node in the traversal, the algorithm first attempts to match the node's subgraph to the acyclic patterns and then to the cyclic patterns. If a match is found, the subgraph of nodes in the CFG are collapsed into a single abstract node representing that acyclic or cyclic pattern. Edges are then added to connect the new abstract node to the neighbors of the now-collapsed subgraph. The process of traversing the CFG and collapsing nodes is continued until the CFG is a trivial graph (i.e., it contains exactly one node). At the same time as nodes are collapsed, an abstract syntax tree is generated to reflect the newly identified control constructs of the collapsed subgraphs.

Note that, in addition to standard acyclic and cyclic regions, which are reducible, the algorithm also detects non-reducible acyclic and cyclic regions called proper (Figure 4) and improper regions (5), respectively. Usually an improper region has multiple entry nodes to a

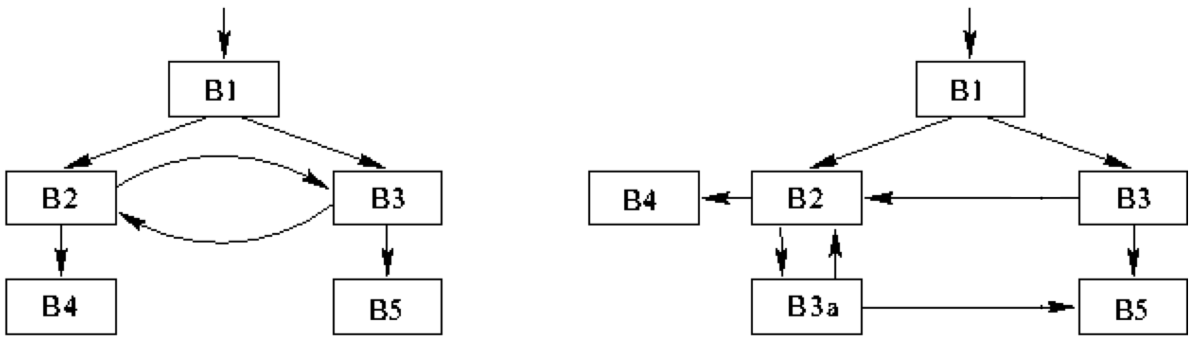


Figure 6: The control flow graph on the left is an improper region because it contains a multi-entrance cycle. Consequently, there is no way to represent the program just using the high-level control flow constructs of C (i.e., the decompiler must resort to using a *goto* statement). However, if the decompiler applies a semantic-altering technique, such as node-splitting, demonstrated on the right, then the decompiler may output functionally-equivalent code that is structured and, hence, does not contain any *goto* statements. In this case, block B3a was inserted as a duplicate of block B3 and several inter-block edges were modified so as to make the control flow graph structured. This image was borrowed from [8].

loop. A proper region has no specific offending characteristics besides the fact that it does not correspond to any of the defined control constructs. During the analysis, the algorithm collapses both proper and improper subgraphs in the same manner it collapses the defined subgraphs, but these proper and improper regions do not correspond to any defined control constructs that can be reflected in the syntax tree or the resulting source code.

To understand why the algorithm works, consider the fact that all the defined control constructs have a single entry point, and hence represent reducible regions. For a more thorough description of structural analysis, refer to [47].

The Phoenix decompiler, developed by [16], is currently one of the most advanced tools for control flow structuring and it applies an improved version of structural analysis. There a few major insights achieved in the Phoenix project.

The first insight is that the standard algorithm for structural analysis, as applied by previous academic projects, actually results in functionally incorrect code. Unfortunately, since few of the previous decompilation projects in the literature actually reported results in terms of functional correctness, this finding was not obvious. The authors of Phoenix then discovered the underlying mistake in the standard structural analysis algorithm: the algorithm solely considers the control flow graph without accounting for the conditions associated with various edges. For example, in many cases, before collapsing a subgraph, it is necessary to determine if the edges leaving a node represent mutually exclusive conditions or not. This detail was ignored in previous implementations of structural analysis. As a result of this finding, the authors of Phoenix defined a new characteristic essential for control flow structuring, namely "semantic preservation." Moreover, they formally defined functional correctness as a critical metric for future research.

The second major insight of the Phoenix project was that the standard structural analysis algorithm overlooked many opportunities to identify structure. By developing a novel technique called iterative refinement, illustrated in Figure 8, Phoenix was able to reduce the final number of *goto* statements by a factor of 30. The key idea to iterative refinement is that, rather than treating proper and improper regions as completely non-reducible, an edge can be carefully selected and removed from the control flow graph (by inserting a *goto* statement into the output code) such that the previous group of nodes forming an improper or proper region now form a region that can be entirely reduced into defined control constructs. Additionally, the authors of Phoenix determined a new definition for the group of nodes that formed the body of a loop.

A final noteworthy technique that frequently accompanies structural analysis is called short-circuit recovery. Developed by [18] and implemented in many decompilers such as dcc and Phoenix, short-circuit recovery attempts to reconstruct compound conditional statements with multiple clauses separated by Boolean operators, rather than outputting several nested or adjacent *if* structures. Figure 9 provides a summary of four common short-circuit simplifications. An important question is whether or not recovering compound conditional statements actually makes the code more readable for a human analyst. In some cases, a few nested if statements may be more readable than a single if statement containing many Boolean operators. [48] quantitatively account for the complexity of conditional statements as a key metric in their formal evaluation of decompilers.

Recently [70] presented an alternative algorithm to

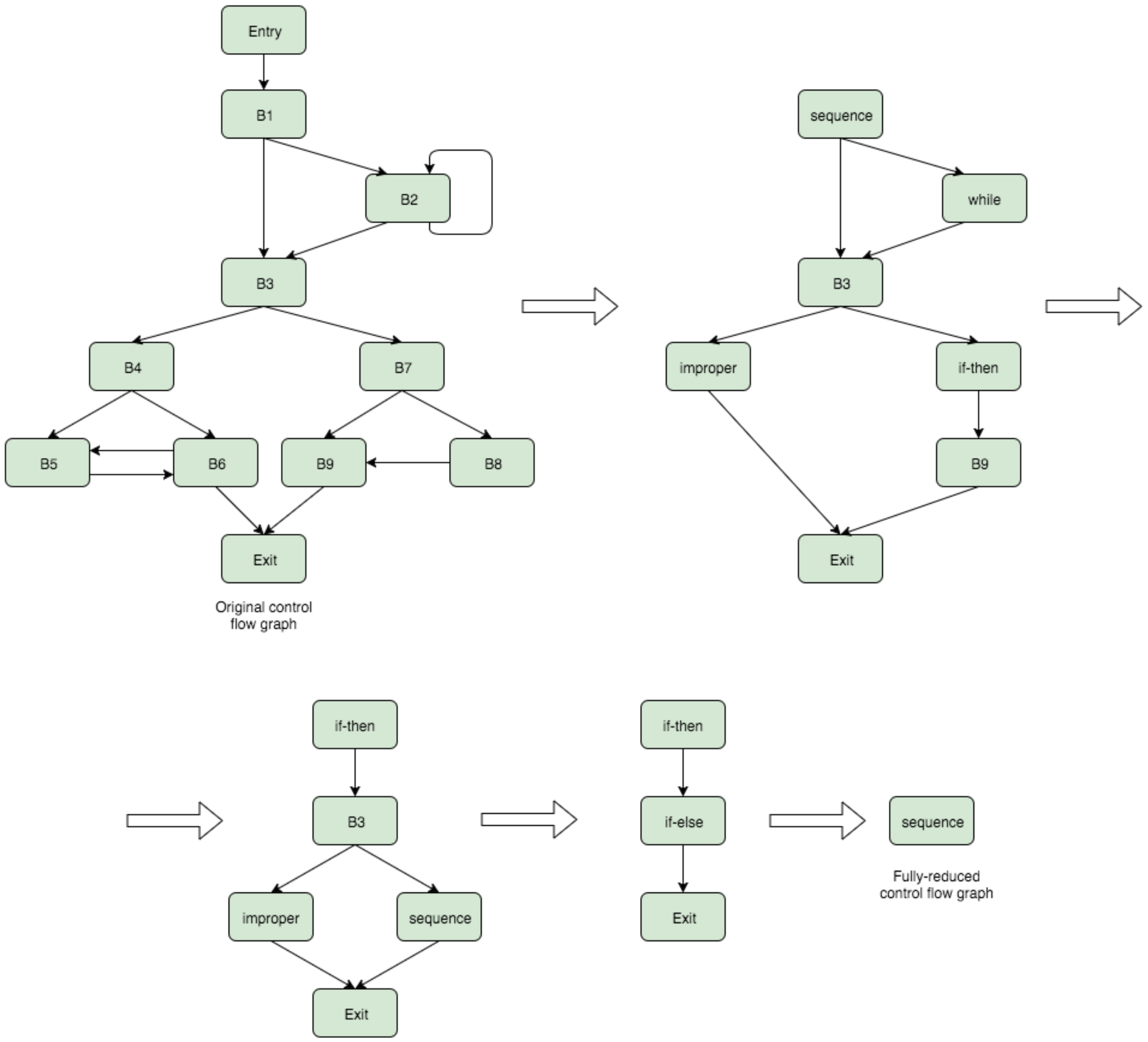


Figure 7: The structural analysis algorithm may take as input the sample control flow graph in the top left corner, and through several steps, ultimately reduce it to the trivial graph in the bottom right corner. Although not shown, the algorithm also produces an abstract syntax tree to record the control flow constructs that it finds. This example is a slightly modified version of an example in [47].

structural analysis in their DREAM compiler, which matches Phoenix in terms of correctness and outperforms Phoenix in terms of structuredness and compactness. The DREAM compiler can fully structure any arbitrary program correctly and usually compactly. DREAM's approach involves a novel method called "patter-independent structuring," left out of this discussion for brevity, as well as semantics-preserving transformations of cyclic regions. It theory, DREAM's output source code could have an entirely different CFG from the input program, despite maintaining functional equivalence. It remains unclear whether DREAM's alterations to the CFG intolerably obscure the intuitive meaning of the code as intended by the original programmer.

The success of Phoenix and DREAM in terms of correctness and structure recovery indicates that the topic of control flow structuring is now a relatively mature field. However, the two approaches have tradeoffs: DREAM recovers more structure than Phoenix, but at the risk of altering the CFG and hence corrupting the underlying essence of the program as originally composed. Potential future research could attempt to determine which strategy is actually most helpful for the human analyst.

Unfortunately, it remains unclear exactly how much room for improvement remains. Potential future research could explore the exact causes of remaining *goto* statements in the output of the decompiler. Is the control flow graph fundamentally irreducible because the orig-

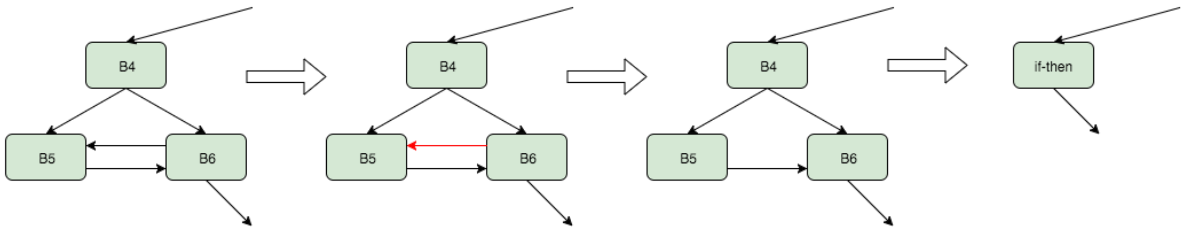


Figure 8: The life stages of code with respect to reverse engineering

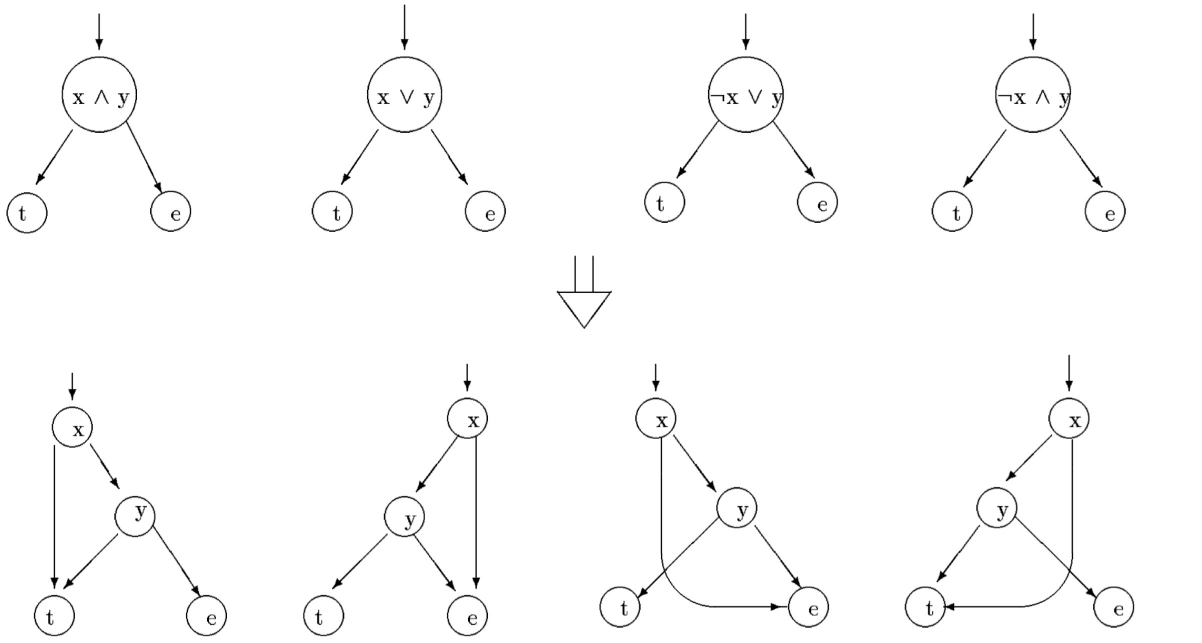


Figure 9: Four examples that demonstrate how compound conditional statements in source code may be implemented in a cascading group of single-condition statements. Note that each example demonstrates a technique called short-circuiting: if the result of the first statement provides enough information to fully resolve the compound statement, then the program does not bother checking the second condition. For example, as shown on the left side of the diagram, if the first condition of an inclusive-or statement is true, then the whole statement is true and there is no need to check the second condition. This image was borrowed from [18].

inal programmer used a *goto* or the compiler performed code motion? Or, on the other hand, do existing techniques still miss opportunities to recover available structure?

A topic tangentially related to control flow structuring is that of quantitatively evaluating control flow complexity. [12] compares several deterministic metrics for describing the intricacy of a control flow graph. In theory, the transformations to a control flow graph to allow it to be structured without *goto* statements could potentially inadvertently increase the complexity of the graph.

2.5 Retargetability

2.5.1 Fully-Retargetable Decompilation

The Retargetable Decompiler (RetDec) [62] [63], developed as part of the Lissom Project at Brno University, is the most comprehensive present day attempt to design a flexible decompiler that is easily and quickly extensible to handle new and diverse platforms. The main motivation behind the project is the rapidly growing population of hardware and software platforms targeted by malware. As opposed to the era when x86 dominated the world of consumer-grade electronics, and hence security tools could concentrate on just one ISA, today's world has a heterogeneous and ubiquitous population of smartphones and Internet-of-Things devices, many of which carry sensitive personal information.

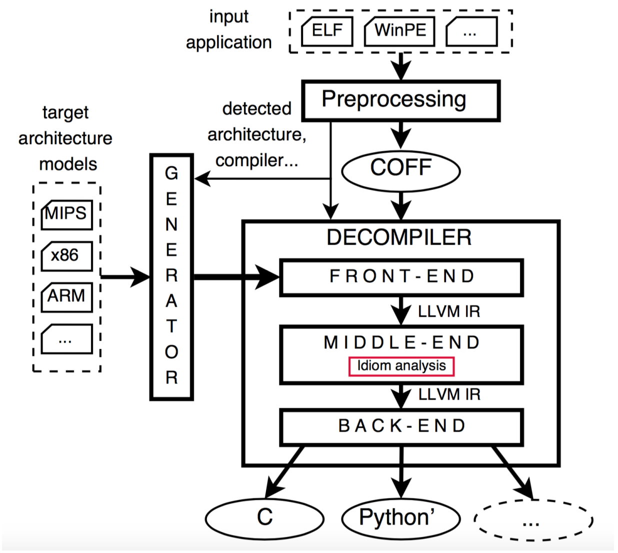


Figure 10: Schematic of the RetDec Retargetable Decompiler, borrowed from [40].

To achieve their goal of retargetability, the RetDec designers repurposed two existing tools not initially intended for decompilation, namely the ISAC architecture description language (ADL) and the LLVM compiler framework.

The ISAC ADL is a key part of the toolchain for automatically generating disassemblers for new processors. Specifically, ISAC is a mixed ADL, meaning that it is a language capable of formally describing a processor in terms of both its physical resources and its operational behavior. The resource description might include information about the registers and cache levels, while the operational description might explain the semantics of each instruction in the ISA using a C-like language.

Once an engineer writes a description of a processor using the ISAC language–a process which requires between one and three months of work for a single person–then an automated tool called the "semantics extractor" can process said description and generate an "instruction decoder" for that processor. The instruction decoder is basically a disassembler, except that its output is not assembly code. Rather its output is code in a standard platform-independent low-level language, namely LLVM IR.

Figure 10 provides a high-level schematic of RetDet. Since all input programs to RetDet are converted to LLVM IR during the initial stages of decompilation, the remainder of the decompiler, in theory, does not need to be retargetable. Rather, the remainder of the decompiler can focus exclusively on LLVM IR, regardless of any new processors and instruction sets. The ability to quickly generate instruction decoders for new processors is therefore the heart of RetDec's retargetability.

In practice, there are a few later stages of the decompilation process which may slightly benefit from knowledge of the target platform. For example, during the idiom detection stage, the decompiler may save a significant amount of time by not scanning for idioms that are specific to other platforms or compilers.

An import question is why the creators of RetDec selected LLVM IR instead of other possible languages suited for internal representation. First and foremost, the creators of RetDec wanted to leverage LLVM's advanced built-in tools for optimizing, transforming, and analyzing code, rather than reinventing the wheel. Specifically, the LLVM opt tool performs efficient passes for constant folding, expression propagation, dead code elimination, loop optimizing, etc. Second, LLVM IR provides a sufficient amount of generality. It's "RISC-like instruction set captures the key operations of ordinary processors, but avoids most of machine-specific constraints." Third, LLVM IR is amenable to many optimization and analysis algorithms because it uses SSA form and strict type rules on all instructions. Fourth, LLVM is extremely well supported and documented.

RetDec also includes a retargetable strategy for idiom detection, discussed in §2.2, as well as a retargetable technique for identifying functions and their parameters irrespectively of the calling conventions.

Unfortunately, the remaining components of RetDec are fairly unremarkable. Its techniques for structuring and type recovery fall far short of the cutting edge, as defined by Phoenix and TIE, respectively. However, nothing about RetDec's retargetable design would inhibit it from adopting the approaches of Phoenix and TIE.

A conspicuous shortcoming of the RetDec project is that the authors provide no experimental measurements or observational evidence to evaluate RetDec in terms of its main goal, retargetability. Given that the ultimate purpose of retargetability is to reduce the time-cost of building a decompiler for a new platform, it would be valuable to know if RetDec, compared to other decompilers, actually saves the engineer time when adding support for new processors.

Although few authors of other decompilers emphasize retargetability as their main objective, most other decompilers do in fact follow a modular design and use a platform-independent internal representation. The front-end is usually the only processor-specific stage. Thus, there is no obvious reason to assume that retargeting said decompilers would be difficult or costly.

Unfortunately, few decompiler authors report the number of man-hours or man-months they spent building their front-ends. In contrast, the authors of RetDec claim that a single person can write the ISAC ADL description for a new processor in approximately one to three months.

[43] has taken the theme of retargetability to an extreme far beyond RetDec. The authors have created

a system, named the Transformer Specification Language (TSL), that systematically generates retargetable tools for machine code analysis. In contrast to RetDec, which solely strives for retargetable decompilation, TSL broadly applies to tools for static analysis, dynamic analysis, and symbolic execution. In their own words, their goal is: “Given the concrete semantics for a language, how can one systematically create the associated abstract transformers?” They demonstrate their system by creating a few applications, including a version of CodeSurfer, that were automatically retargeted to x86 and PowerPC32.

2.5.2 C++ Decompilation

The output of a decompiler can be any high-level representation of the input program. Since various decompilers are designed with different applications in mind, they often produce output in dramatically different formats. For example, a decompiler for porting programs might produce strictly-conforming C code, while, in contrast, a decompiler designed for reverse engineering malware might produce highly readable, but non-compilable pseudo-Python code. In particular, the decompiler’s output language does not need to be the same as the source language used to create the machine code, especially because the user of the decompiler often does not know what source language and compiler were used to create the machine code. In practice, most machine code decompiler designers choose to produce C code as the high-level output because C is simple, well-known, well-supported by source-level tools and compilers, powerful in its ability to support low-level memory operations, and capable of expressing most features of other languages.

Unfortunately, programs written in C++ result in machine code that is more intricate than that of programs written in C and, consequently, comprehending the decompiled output is usually more difficult. To make matters worse, C++ is rapidly increasing in popularity, even within malware such as Zeus, Conficker.D, Agobot, etc [46].

Thus some research has focused specifically on decompiling C++ programs and producing C++ output. Along these lines, the author of the Hex-Rays decompiler [29] has suggested that a plugin could be created to post-process the C output of Hex-Rays and produce C++ code. However, more recent research indicates that, to extract the most information from a C++ program, it is necessary to exploit the low-level mechanisms and patterns used by C++ compilers within the assembly code.

Loosely speaking, C++ is a superset of C and it mostly uses the same control structures, primitive data types, and syntax as C. C++ uses the notion of classes to expand upon C with object-oriented programming features, including abstraction, encapsulation/information hiding, inheritance, and polymorphism. Additionally, C++ includes more support for exception handling and a standard library. Fortunately for decompiler designers, the three most common C++ compilers, GCC, Clang and MSVC, all implement C++’s object-oriented features using similar strategies. Moreover, each compiler implements C++ classes in a structured and deterministic way, making the recovery of C++ class information a relatively straightforward process. Below is a brief summary of techniques for decompiling C++’s class features. For a more complete description of C++ decompilation, please see [46].

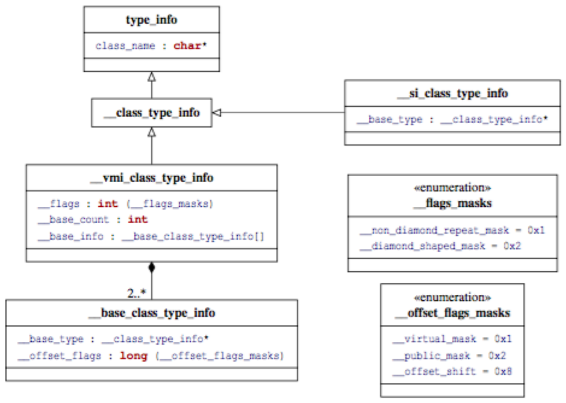

type_info
class_name : char*
__class_type_info
__si_class_type_info
__base_type : __class_type_info*
__vmi_class_type_info
__flags : int (__flags_masks)
__base_count : int
__base_info : __base_class_type_info[]
«enumeration» __flags_masks
__non_diamond_repeat_mask = 0x1
__diamond_shaped_mask = 0x2
2..*
__base_class_type_info
__base_type : __class_type_info*
__offset_flags : long (__offset_flags_masks)
«enumeration» __offset_flags_masks
__virtual_mask = 0x1
__public_mask = 0x2
__offset_shift = 0x8

Figure 11: Layout of the RTTI structure used by the GCC and Clang compilers. Image borrowed from [46].

The layout in memory of every C++ class is simply a C structure that sequentially contains the class’s data members, both regular and inherited, and an optional virtual table pointer at the beginning when the class has any virtual methods. Thus, ordinary techniques for C data abstraction recovery are suitable for determining the members of a C++ class.

There are two different techniques for determining the hierarchy of polymorphic classes. The first technique is based on parsing of Run-Time Type Information (RTTI) records. For every polymorphic class, there is a structure called an RTTI record that is used to implement dynamic querying of object type. The RTTI structure indicates the parents of the derived class and the class name. Therefore, it is easy to reconstruct the entire hierarchy of polymorphic classes by locating and parsing all the RTTI structures. Figure 11, borrowed from [46], outlines the RTTI structure used the clang and gcc compilers.

Finding the RTTI records is usually simple because the first element of each virtual table is a pointer to the corresponding RTTI structure. Virtual tables are often located in the read-only data segment and usually they consist of a list of virtual function pointers, prepended with a pointer to the corresponding RTTI record. To find all the virtual tables, linearly sweep through each word of the read-only data section. The start of a virtual table will be indicated by a pointer to a function in the text

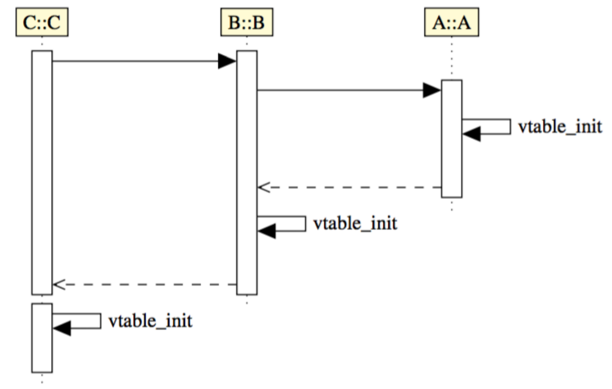


Figure 12: Sample call-sequence of constructors for the initialization of an object of class C, which is derived from Class B, which, in turn, is derived from Class A. Image borrowed from [46].

segment such that the pointer itself is referenced from somewhere in the text segment. All subsequent elements in the virtual table will be function pointers that are not referenced anywhere else.

Unfortunately, many C++ projects deliberately exclude RTTI information, so as to eschew both its potential misuse and its performance cost. Thus the second approach to polymorphic class hierarchy recovery, described by [26], does not depend on RTTI records. Rather, the approach makes negative inferences about the relationships of classes based on their virtual tables. For example, if the class Foo's vtable is smaller than class Bar's vtable, then class Foo is definitely not a derived class of Bar. Additionally, the approach relies upon studying the order of constructor of constructor initializations to infer the hierarchy. Figure 12, borrowed from [46], illustrates how knowledge of the constructor call-sequence may leak information about the class hierarchy. When an object of a derived class is initialized, the derived class's constructor first calls the constructors of its base classes, before initializing its own vtable fields and executing the code within its own body. Thus, from Figure 12, we may infer that C is a derived class of B, which is a derived class of A.

To resolve class pointers which may be polymorphic, modern techniques apply a lattice model, as illustrated in Figure 13 which is borrowed from [46]. If a pointer variable P references an object of class E, then later references an object of class F, then at most we can conclude the P might be a pointer to class D or P might be a *void* pointer.

Unlike virtual methods, non-virtual methods lack a parameter for a virtual table pointer and thus it may be difficult to distinguish these methods from ordinary C functions in code produced by Clang or GCC. Fortunately, the calling convention of the MSVC C++ compiler always reveals the the class of non-virtual methods.

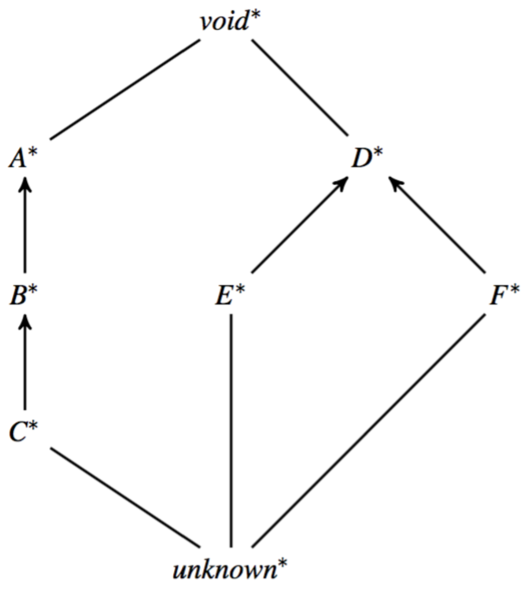


Figure 13: Sample lattice used to infer the possible types of a polymorphic class-pointer. The interior edges represent class inheritance relationships. In this example, C is a derived class of B, which is a derived class of A. Image borrowed from [46].

Beyond class reconstruction, there are also techniques aimed at decompiling C++ exception handling features and standard library usage, but these strategies are left out of this discussion for brevity.

To summarize, much of C++ decompilation relies upon understanding and exploiting the predictable mechanisms of common C++ compilers. An unfortunate pitfall of this compiler-based approach is that many of the decompilation techniques are probably brittle and easily thwarted by low-level obfuscation.

3 Experimental Approaches to Decompilation

3.1 AI-Based Decompilation

Artificial Intelligence (AI) shows promising potential to solve a few specific subproblems of decompilation that are fundamentally outside the grasp of traditional techniques. In particular, [24] highlights a few scenarios that lack concrete evidence in the assembly code to distinguish between multiple potentially correct source code representations. For example, in the C standard calling convention, a function's return value is placed in *%eax* register. However, since *%eax* is a general purpose register and may be used for arbitrary operations, in addition to holding return values, it is frequently unclear whether a function actually has a return value, or whether the value stored in *%eax* is just a residual term leftover from an intermediate calculation. While a few heuristics exist (e.g., if function *foo* calls function *bar* and then reads *%eax* before writing to it, then likely *bar* does have a return value), there are many situations in which the decompiler must simply guess about the existence of a return value. By default in these uncertain situations, the Hex-Rays decompiler will always guess affirmatively that there is a return value. Another major decompilation subproblem involving similar ambiguity is choosing the specific type of a variable. Often times, there is no evidence to indicate whether a 32-bit register or stack slot corresponds to a signed integer, an unsigned integer, or a pointer. In these cases, the most advanced type recovery tool, TIE, outputs a bounded range of potential types since it cannot definitely choose the one correct type for the variable.

To solve these ambiguous subproblems, recent research has attempted to apply AI to decompilation. In particular, [24] describe an infrastructure for automatically creating training sets for use in a decompilation context. They demonstrate their system on the aforementioned problem of determining if a function returns a value. Ultimately, they produce a dataset that, for each function in a program, maps a known Boolean (does the source function have a return value) to a series of assembly code patterns that occur with the compiled function. This technique could then be applied to many programs, studied with many different compilers and many different optimization levels, in order to produce a large training set. An AI engine would then using the training set to produce a predictive model. Based on the predictive model, a new heuristic could be implemented as a plugin to a traditional decompiler.

Fortunately, the authors outline their system in a highly parallelizable and practical design to enable the creation of very large training sets. Unfortunately, the authors provide no results indicating the performance of the learned heuristics.

3.2 Deprogramming

Whereas decompiling is the process of recovering the source code of a program, deprogramming is the process of recovering the original programmer's ideas, patterns, styles, and intentions from a program [21]. Essentially, deprogramming is decompilation taken to the next higher level of abstraction, as illustrated in Figure 14. Both processes have several applications in common, and in many cases, the two processes complement each other. For example, when reverse engineering a program to identify potential security vulnerabilities, it might useful to identify locations in the code that follow risky design patterns susceptible to race conditions. Likewise, deprogramming might be useful when quickly attempting to learn the algorithm or architecture of a competing com-

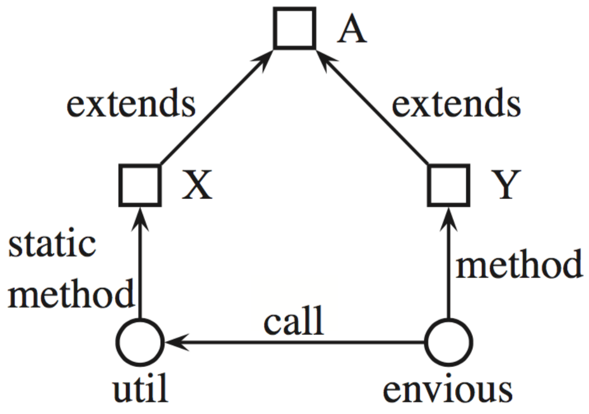


Figure 15: Layout of the "sibling envy" design pattern, often target for removal during refactoring projects. Image borrowed from [21].

pany's product. In some cases, deprogramming may rely upon access to source code, thereby augmenting the decompiler's output. In other cases, deprogramming could act directly on the assembly code and apply techniques similar to traditional decompilation.

Other potential applications of deprogramming, mentioned by [21], include: issuing warnings when software engineers commit code with poorly-chosen design patterns; identifying functions that are most in need of refactoring; providing a high-level structural view of a large codebase to programmers; semantically detecting illegally or precariously copied and pasted code snippets; automatically generating documentation that easily stays synchronized with the code; recalling the forgotten ideas and concepts underlying a legacy piece of software; and creating fingerprints of different programmers' coding habits so as to allow attributing authorship of a valued piece of software, of malware, or of a homework assignment.

Deprogramming is a relatively immature field, but there are a few promising projects. [21] created a tool called DeP which is currently capable of examining Java bytecode or source code and locating instances of specific design patterns. For example, a software engineer refactoring a large program might want to quickly find and redesign all the instances of the sibling envy design pattern, shown in Figure 15, because the pattern misuses the concept of inheritance and ultimately leads to less readable and less maintainable code. Figure 16 shows DeP's graphical user interface which highlights every occurrence of the user-specified pattern.

Additionally, DeP provides refactoring capabilities in the form of drag-and-drop GUI that allows programmers to clearly understand the structural ramifications of potential refactoring operations. The refactoring tool may also partially automate some of the source code modifications. Under the hood, DeP relies upon the cre-

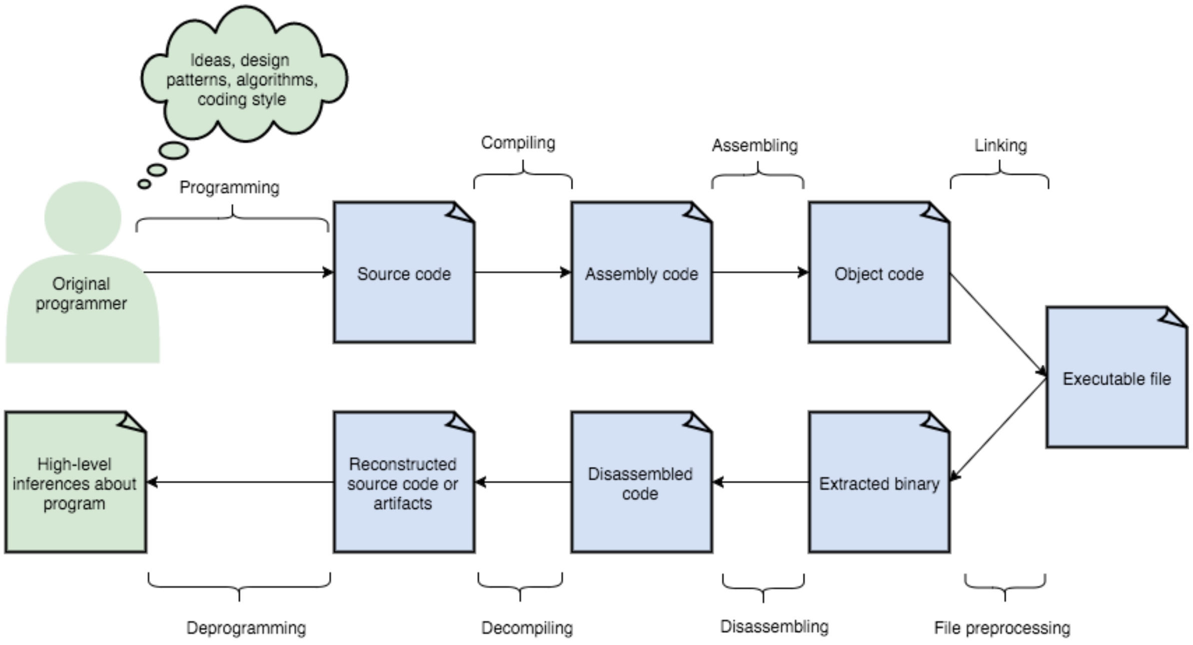
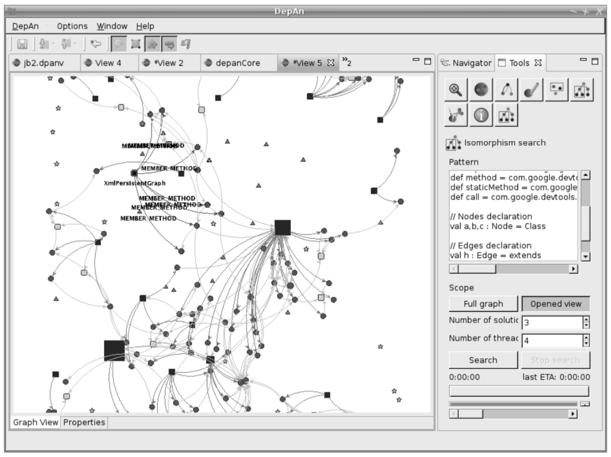


Figure 14: The life stages of code with respect to reverse engineering



```
def method = com.google.devt
def staticMethod = com.google
def call = com.google.devtools.

// Nodes declaration
val a,b,c : Node = Class

// Edges declaration
val h : Edge = extends
```

Number of solutic 3
Number of threac 4
0:00:00 last ETA: 0:00:00

Figure 16: Typical GUI view of the DeP refactoring tool. Image borrowed from [21].

ation of a dependency graph showing the membership, inheritance, and fault-propagation relationships between classes, fields, files, functions, etc. DeP applies a modified version of the VF2 algorithm with heavy pruning to locate subgraph isomorphisms and identify design patterns.

Another successful deprogramming tool is MOSS [7], which is used to semantically detect academic cheating on programming assignments, even when the students apply basic obfuscation strategies such as renaming variables, reordering instructions, and inserting dead code.

4 Obfuscation and Other Countermeasures to Decompilation

Software authors often defend their programs against decompilation for several reasons: protecting Intellectual Property (IP), hindering anti-malware analysis and detection, and strengthening software security against attacks such as buffer overflows or firmware replacement. Here we will briefly explore several obfuscation techniques, loosely categorized. Due to the extensive cat-and-mouse nature of research in the fields of program obfuscation and analysis, here we only skim the surface of popular back-and-forth countermeasures.

4.1 Limiting Access to the Program's Binary Code

An analyst cannot decompile a program that he does not have. The following techniques attempt to limit access to the executable program.

"Automatic software diversity," also known as "code morphing," is the practice, popular among vendors, of compiling and distributing many functionally-equivalent, yet internally diverse, executables from the same source code. Thus, if an attacker develops a sploit on his own instance of a vulnerable program, it is unlikely that the sploit will succeed on the targets' versions of the program, since each deployed executable might have different control flow, instruction orderings, stack layouts, buffer sizes, etc. [41] provides a survey of the existing research and [6] implements code morphing to protect cryptographic code in OpenSSL from Differential Power Attacks on an ARM SoC.

Additionally, viruses and worms, which seek to evade signature-based detection schemes, often include and rely upon "metamorphic engines," which produce diverse bootstrap code when infecting new hosts [59].

"Mobile code download" is another access-limiting technique. [25] outlines a strategy that client/server applications can use to ensure the tamper-resistance of the client code. Specifically, their system depends upon requiring periodic replacements of the client's self-checking module, sent by the server.

Often hardware enables additional access-limiting strategies. For certain embedded systems, any program obfuscations that hurt performance or that remove reliability guarantees may be unacceptable. Instead, these embedded systems often rely on secure hardware design in order to restrict access to the firmware. Consider the following design: at system startup, bootstrap code within an SoC loads the encrypted application code from insecure external flash; next, the bootstrap code decrypts the application code and stores the resulting executable in the SoC's internal RAM. In such a system, extracting the firmware may require expert cryptanalysis or the expensive and destructive processes of IC decapsulation and scanning electron microscopy [58].

Similarly relying upon hardware support, [36] demonstrates a method to conceal the control-flow-related binary code of applications that run on FPGA soft microprocessors. The authors created a tool to remove the "branching function," which performs obfuscated routing, from the application code and instead implement said routing with a hardware module in the FPGA. Thus, assuming that the adversary cannot overcome the bitstream encryption of the FPGA, the adversary will be left with only the application binary, which provides an incomplete view of the program.

Code packing is another strategy for limiting access to a program's binary code, but, due to its widespread prominence, we will discuss it separately in §4.2.

4.2 Code Packing and Self-Modifying Code

Static analysis of a binary may become impractical if the program can overwrite or create its own code at runtime, which is often the case for defensive malware and just-in-time (JIT) compilers. Many malware programs apply a technique called code packing to hinder binary analysis and anti-virus detection strategies.

As [56] concisely defines it, "a packed binary is one that contains a payload of compressed or encrypted code that it unpacks into its address space at runtime." The portion of the program that performs the unpacking during runtime is called the "bootstrap code" or "metacode."

Before being unpacked, the packed payload essentially appears as random data to any static decompiler. Thus, in order to perform static decompilation on a packed program, it is first necessary to extract the unpacked payload binary.

In many cases, the target malware may apply multiple rounds of packing, often to minimize the size of the exposed bootstrap code or to increase the difficultly of analysis. For example, a small bootstrap function may unpack a larger bootstrap function, which subsequently unpacks the malware. The [] packer tool applys seven layers of packing.

There are two common static approaches to extracting the payload binary code. The X-Ray technique attempts to decrypt or decompress the packed payload usually without studying the bootstrap code [50]. Initially, the X-Ray technique may examine the statistical characteristics of the packed payload and infer if the payload has been compressed, and if so, with which compression algorithm. If, alternatively, the payload has been encrypted, the X-Ray technique may apply known plaintext attacks in order to help decrypt the payload. The X-Ray technique usually fails if the packing algorithm used a tough encryption algorithm or applied more than one layer of packing.

Another static extraction strategy, which several groups have tried to automate, is to identify the piece of bootstrap code that performs the unpacking and use it as a guide for building a custom unpacker tool [20].

Dynamic tools for payload extraction rely on the fact, during runtime, the target program usually unpacks itself. Often these tools aim specifically to identify code that has been written then executed. One particular tool, Renovo, uses whole-system emulation with Qemu in order to evade detection as it follows the target's execution and logs memory writes on a per-word basis, allowing it accurately detect unpacked and executed payload code. Another tool, EtherUnpack, uses the Xen hypervisor to achieve the same goal. Other dynamic tools take a course-grained approach and only watch execution and writes at page granularity. Generally the coarse-grained approaches are much quicker and more suitable for antivirus tools.

A topic closely related to dynamic payload extraction is virus detection. Many antivirus products apply algorithms based off of “generic decryption.” Similar to battle between static analysis and obfuscation, virus detection is a cat-and-mouse game.

Depending on the application, the dynamic analysis may execute the target program for a fixed amount of time, or until the unpacking is finished. Unfortunately, many malware samples may apply techniques to complicate detection and analysis. For example, a virus may only actual unpack itself 10% of the time or hide its entry point within the host program or delay unpacking until the antivirus software times out.

To exacerbate the difficulty of analysis, some packer tools apply a subtly more troublesome form of packing: code overwriting, also known as “self-modifying code.” For example, the program may overwrite bootstrap code or payload code with new payload code. Thus there is never a single point in time when all the code of the program is unpacked and present. In an extreme case, the program may use a small finite-size buffer as the exclusive location for unpacked code. After executing the code in the buffer, the program overwrites the buffer with a new segment of unpacked code to execute, and then repeats the process.

Analyzing self-modifying code is particularly difficult due to the challenge of representing it statically. Recent research has focused on creating an enhanced CFG to represent statically every piece of the code, even those occupying the same addresses at different times. Unfortunately, most existing unpacking tools only construct the CFG at the end of analysis, and therefore they only capture one snapshot version of the code.

4.3 Hindering Static Disassembly

Another class of obfuscation techniques focuses on disrupting the static disassembly process, with the assumption that the adversary has overcome the aforementioned challenges in §4.1 and §4.2 and has complete access to the unpacked program binary.

[45] provides a strategy to partially desynchronize linear-sweep disassembly, in architectures allowing variable-length instructions, by inserting partial instructions (“junk code”) before basic blocks that are unreachable via fall-through.

Several techniques to confuse recursive-traversal disassembly include:

- deviating from the assumed convention that functions return to the address immediately following the call instruction [45][56];
- multiplexing and demultiplexing many unconditional jumps or function calls through a single intricate “branching function” to obscure control flow analysis [45][36];
- disguising unconditional jumps as conditional jumps via “opaque predicates” [19];
- tampering with the call stack and applying abnormal uses of the ret instruction [56];
- applying control transfers via signals and exceptions [52];
- inserting fake jump tables [45];
- fuzzing of popular disassemblers and code analysis tools with the hopes of finding obscure instructions and states that cause the analysis tool to output an incorrect result [56]

4.4 Convoluting the Program to Prevent Comprehension

A further class of obfuscation tactics attempts to impede the analyst’s understanding of the program, given that the analyst has already successfully disassembled the program.

For example, several tactics can increase the difficultly of detecting function boundaries. It is possible to conceal the start and end of a function by using alternative instruction sequences in place of ‘call’ and ‘ret’. On the other hand, using the ‘call’ and ‘ret’ instructions abnormally for intra-function control flow may mislead analysts into detecting more functions than are actually present.

Anther technique to confuse function identification is the scrambling of basic blocks so that they are no longer contiguous in memory for a given function. A similar and compatible tactic is the sharing of basic blocks between multiple overlapping functions. To make matters worse, it also possible to overlap basic blocks if the instruction set contains variable-length instructions.

Other common strategies for misleading the analysts include: obfuscating constants, adding superfluous predicates to conditions, flattening a group of basic blocks into a single switch structure inside of a while loop, intermixing and disobeying function calling conventions (e.g. treating a status flag as a function return value), inserting do-little and do-nothing code, and interpreting custom bytecode instructions through an emulator inside the executable.

A new area of research that impedes program comprehension is “cryptographic obfuscation,” not to be confused with the entirely separate technique of cryptographic packing. Whereas cryptographically packed code is just binary code that has been encrypted, cryptographically obfuscated code is unencrypted binary code whose functional behavior, expressed as a Boolean function of non-negative arity, is mathematically-provably difficult and expensive to determine by either analyzing the code or black-box testing. [10] provided the first prototype of cryptographic obfuscation by implementing a 16-bit point function, which returns true if and only if the input value exactly matches a secret value. Using 32 cores, the authors' tool required 387 minutes to obfuscate the 16-bit function, resulting in a 31 GB program. Executing the program required 11 minutes and 3 GB of RAM.

4.5 Using the Law as a Deterrent

In some jurisdictions, decompilation, and reverse engineering in general, may be explicitly made legal or illegal.

In the US, the Digital Millennium Copyright Act (DMCA) [5] expressly allows a person who has legally obtained a copy of a program to reverse engineer said program, including circumventing any defenses, for the purpose of enabling interoperability. Likewise, the law allows creation of tools to assist in reverse engineering for interoperability.

On the other hand, the Bowers vs. Baystate Technologies court case [60] established the controversial precedent that End-User License Agreements (EULA), which typically disallow reverse engineering, may override fair use rights and copyright laws that explicitly allow reverse engineering.

5 Alternative Strategies to Static Decompilation

We have already examined several subtasks of decompilation, including disassembly (§2.1), type recovery (§2.3.2), and control-flow structuring (§2.4), that benefit from dynamic techniques. In this section, we will consider additional scenarios in which alternative strategies may outperform or augment static decompilation.

As previously explained in §4.2, static decompilation is often insufficient for analyzing packed or heavily-obfuscated binaries, and consequently, techniques such as x-raying, unpacker building, generic decryption, and extended-CFG creation are needed to extract the unpacked code prior to static decompilation.

One additional challenge created by packed malware is that security analysts may waste time redundantly examining multiple polymorphic instances of a single malware program because it is not immediately apparent that the samples are all closely-related mutations of one another. To address this issue, [31] created a filter to, quickly and without unpacking, detect when a new sample of malware is actually closely related to a previously-analyzed sample. The critical insight behind the tool is that the compression and encryption strategies applied by packers usually "do not break, in the packed versions, all the similarities existing between the original versions of two programs." Using the tool, analysts only have to examine 20% - 33% of incoming malware samples. [51] provides a similar tool to automatically analyze, dynamically and statically, malware samples and then recognize and categorize new or previously-discovered behaviors. Ultimately, the tool saves time for the analyst, who can quickly understand a new malware sample in terms of defined behaviors and focus on examining samples with interesting behaviors.

Many behavior-based strategies for malware detection and analysis do not need to recover or examine the program's source code, sometimes not even the assembly code. [64] presents a tool to detect malware based on anomalous plug-load behavior in IoT devices. [68] demonstrates a sandbox tool for automatically dynamically monitoring and logging all the system calls executed by a malware sample. The tool relies upon "code patching" strategies like Windows API hooking and DLL injection. [38] demonstrates an efficient malware detection tool that characterizes malware samples based on information flows amongst system calls. Similarly, there are dynamic fine-grained tools, such as interactive debuggers, that apply code patching to create software-based breakpoints. Not surprisingly, there are also several obfuscation techniques, such as self-checksumming and stolen-byte insertion, that can hinder static and dynamic code patching.

Two additional dynamic techniques useful in software security are forward symbolic execution and taint analysis. [57] provides the first formal definitions of the two techniques. "Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input. Dynamic forward symbolic execution automatically builds a logical formula describing a program execution path, which reduces the problem of reasoning about the execution to the domain of logic." Their applications include: finding unknown bugs, creating test cases, analyzing malware, and generating filters to block malicious input. Fuzzing is an additional dynamic strategy which may help detect vulnerabilities.

Decompilation is often helpful for porting applications to new instruction sets or modifying binaries, but there are alternative methods. "Binary translation," also called "binary rewriting" or "binary recompilation," is the process of modifying binary code into another form of binary code, without using source code as an intermediate step. Binary translation can be performed in hardware or software and it can be performed statically or dynamically.

Static translation faces the same major challenge as disassembling, namely finding all the code. [14] provides a strategy, based on superoptimization methods, for automatically generating translators between instruction sets. They achieved 67% performance converting PowerPC to x86 on big benchmarks.

Dynamic binary translation involves actually executing the target program. Every time a new basic block is discovered, it is translated and the result is cached. At every opportunity, jump instructions are modified to point to the cached versions of their target basic blocks. In contrast to emulation, which performs a per-instruction read-decode-execute cycle and does not memoize the results, dynamic translation translates a whole basic block at a time and saves the result. One major cause of slowdown in dynamic translation is the need to translate the "Source binary Program Counter (SPC)" to the "Translated Program Counter (TPC)" for every indirect jump, a process requiring 10 instructions. However, [34] provides a novel and efficient solution: using the SPC to index a redirecting table, instead of using a hash tables or software prediction. Dynamic recompilation is an extension of dynamic translation. It profiles the program and optimizes hot spots.

In addition to porting applications, static binary rewriting may be used to enhance programs in terms of security, memory usage, performance, etc. [54] and [67] demonstrate binary rewriting techniques to improve software security.

To reuse legacy code in new projects, it is not always necessary to recover the source code. Binary code reuse is the practice of selecting segments of binary code from an application and then recompiling said code segments so that they may be linked with other code modules to create a new application. Common security tasks that may benefit from binary reuse include: inspecting and analyzing malware, grafting security-critical functions into new programs, recovering source code, and updating legacy software. [71] recently created a technique, coined "trace-oriented program (TOP)," which uses dynamic decompilation techniques to implement binary reuse. The key idea is, "given a binary (possibly obfuscated), through dynamic analysis of its execution, we collect instruction traces and translate the executed instructions into a high level program representation using C with templates and inlined assembly (for better performance)." The resulting code can be linked with other components to create a new program. Through its dynamic approach, TOP avoids many typical obstacles faced by static decompilation, including extracting packed code and resolving indirect jumps during disassembly. However, when TOP collections several execution traces of the target program, it may not necessarily achieve full program coverage. Thus the authors instrument the code produced by TOP with safety checks to ensure that the program counter never reaches a previously unencountered address. If a single branch from conditional jump instruction is missing from the traces, TOP includes an exception handler to display a warning and gracefully quit the program if, during runtime, the program counter reaches the absent branch. They apply a similar strategy for indirect jumps.

In the realm of software security, decompilation is often helpful for fixing bugs like buffer offerflows in legacy code. However, there are many alternative strategies for preventing exploits, some of which may be cheaper and faster. Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) usually require no modifications to the binary, although DEP cannot be applied for programs which execute dynamically-created code, such as Just-In-Time compilers. Stack guards, also known as "stack canaries," are another protection mechanism and they are usually added to the program during compilation, but [54] demonstrates a technique to add stack guards using binary translation when source code is not available. Control Flow Integrity is another technique that is usually applied during compilation, but [72] demonstrates an implementation based on binary translation for Commercial Off-The Shelf (COTS) binaries.

Many other tools exist that use techniques similar to decompilation but without attempting to recover source code. [53] successfully shows a strategy for detecting plagiarized Android applications by fingerprinting binaries' "method-level Abstract Syntax Trees." [32] attempts to "recover the lineage given a set of program binaries."

6 Future Research Directions

A thorough evaluation of state-of-the-art decompiler technology is difficult because many reverse engineering groups, both in government and private industry, are biased against publishing any details about their internally-developed tools. While there is no direct evidence available that nation states' defense agencies have developed in-house decompilers, it is unlikely that they all rely exclusively upon a consumer-grade product like Hex-Rays, which was developed by a small team and which only began support for x64 processors in June, 2014. Thus our perspective, based primarily on academic papers, is limited and probably underestimates the true capabilities of the reverse engineering world. Nevertheless, we here present the apparent gaps in existing research and suggest future projects.

The most glaringly gap in the literature is the lack of any human studies to test the usability of decompilers. Currently, there are several instances in which the designer of the decompiler must weigh the tradeoffs of multiple alternative strategies that affect the output of the decompiler. Currently, most designers make an decision based on gut-instinct, with little or no concrete evidence to support their decision.

Furthermore, the academic community currently suffers due to the lack of agreed upon metrics by which to compare decompilers. The TIE and Phoenix projects partially solved this issue by introducing the quantitative metrics of correctness, structuredness, type precision, and type conservativeness. However, metrics are needed to evaluate and contrast the usability, retargetability, and overall reverse engineering effectiveness of a decompiler.

Finally, there is significant room for improvement in disassembling and decompiling packed and heavily obfuscated programs. Some of the most promising recent advances have been hybrid, incorporating both static and dynamic methods.

7 Conclusion

I have presented the traditional stages of static decompilation, which are fairly mature and successful for unobfuscated targets. I have also examined decompilation's role within the larger context of reverse engineering packed and obfuscated malware. The most promising future research areas include collecting human feedback on decompiler usability, determining standardized metrics and evaluation techniques, combining static analysis with dynamic analysis, developing innovative strategies for deprogramming, and overcoming the packing and obfuscation strategies of malware.

Acknowledgments

I would like to thank my advisors, Kirill Levchenko and Stefan Savage, for their wise guidance during my career as a graduate student. Additionally, Kirill provided high-level feedback to help guide my exploration of decompilation in this report. Geoff Voelker gave me great strategies for quickly finding quality literature spanning the full breadth of a field.

References

[1] Cast static source code analysis tool for developers. `http://www.castsoftware.com/products/code-analysis-tools`.

[2] Klocwork static source code analysis tool. `http://www.klocwork.com/capabilities/static-code-analysis`.

[3] Parasoft static source code analysis tool for C and C++. `https://www.parasoft.com/product/static-analysis-cc/`.

[4] PMD static source code analysis tool and copy-paste detector. `https://pmd.github.io/`.

[5] U.S. code title 17, section 1201 (f) - circumvention of copyright protection systems. `https://www.law.cornell.edu/uscode/text/17/1201`.

[6] AGOSTA, G., BARENGHI, A., AND PELOSI, G. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 77–82.

[7] AIKEN, A. Moss: A system for detecting software plagiarism. `https://theory.stanford.edu/~aiken/moss/`, 2015.

[8] ALLAN, S. Advanced compilers course notes on control flow analysis. `http://digital.cs.usu.edu/~allan/AdvComp/Notes/controld/controld.html#con:nsplit`, 2001.

[9] ANDERSON, P., AND TEITELBAUM, T. Software inspection using CodeSurfer. In *In Workshop on Inspection in Software Engineering* (2001).

[10] APON, D., HUANG, Y., KATZ, J., AND MALOZEMOFF, A. J. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive 2014* (2014), 779.

[11] AYEWAH, N., AND PUGH, W. The Google findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2010), ISSTA '10, ACM, pp. 241–252.

[12] BAKER, A., AND ZWEBEN, S. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering 6*, 6 (1980), 506–512.

[13] BALAKRISHNAN, G., AND REPS, T. DIVINE: Discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation*, B. Cook and A. Podelski, Eds., vol. 4349 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 1–28.

[14] BANSAL, S., AND AIKEN, A. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 177–192.

[15] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.

[16] BRUMLEY, D., LEE, J., SCHWARTZ, E. J., AND WOO, M. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 353–368.

[17] BRUMLEY, D., SONG, D. X., CHIUEH, T., JOHNSON, R., AND LIN, H. RICH: automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007* (2007), The Internet Society.

[18] CIFUENTES, C. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[19] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 184–196.

[20] COOGAN, K., DEBRAY, S., KAOCHAR, T., AND TOWNSEND, G. Automatic static unpacking of malware binaries. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering* (Washington, DC, USA, 2009), WCRE '09, IEEE Computer Society, pp. 167–176.

[21] COPPEL, Y., AND CANDEA, G. Deprogramming large software systems. In *Proceedings of the Fourth Conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2008), HotDep'08, USENIX Association, pp. 3–3.

[22] EASTWOOD, A. It's a hard sell - and hard work too. (software reengineering). *Computing Canada 18* (1992), 35.

[23] EROSA, A., AND HENDREN, L. Taming control flow: a structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on* (May 1994), pp. 229–240.

[24] ESCALADA, J., AND ORTIN, F. An adaptable infrastructure to generate training datasets for decompilation issues. In *New Perspectives in Information Systems and Technologies, Volume*

2, . Rocha, A. M. Correia, F. . B. Tan, and K. . A. Stroetmann, Eds., vol. 276 of *Advances in Intelligent Systems and Computing*. Springer International Publishing, 2014, pp. 85–94.

[25] FALCARIN, P., SCANDARIATO, R., AND BALDI, M. Remote trust with aspect-oriented programming. In *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on* (April 2006), vol. 1, pp. 6 pp.–.

[26] FOKIN, A., DEREVENETC, E., CHERNOV, A., AND TROSHINA, K. SmartDec: Approaching C++ decompilation. In *Reverse Engineering (WCRE), 2011 18th Working Conference on* (Oct 2011), pp. 347–356.

[27] GUILFANOV, I. Hex-rays decompiler. `https://www.hex-rays.com/products/decompiler/`.

[28] GUILFANOV, I. Ida pro interactive disassembler. `https://www.hex-rays.com/products/ida/`.

[29] GUILFANOV, I. Decompilers and beyond. `https://www.youtube.com/watch?v=00EqvVtLdJo`, 2008.

[30] HALSTEAD, M. H. *Machine-independent computer programming*. Spartan Books, 1962.

[31] JACOB, G., COMPARETTI, P., NEUGSCHWANDTNER, M., KRUEGEL, C., AND VIGNA, G. A static, packer-agnostic filter to detect similar malware samples. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, U. Flegel, E. Markatos, and W. Robertson, Eds., vol. 7591 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 102–122.

[32] JANG, J., WOO, M., AND BRUMLEY, D. Towards automatic software lineage inference. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 81–96.

[33] JETLEY, R. P., JONES, P. L., AND ANDERSON, P. Static analysis of medical device software using CodeSonar. In *Proceedings of the 2008 Workshop on Static Analysis* (New York, NY, USA, 2008), SAW '08, ACM, pp. 22–29.

[34] JIA, N., YANG, C., WANG, J., TONG, D., AND WANG, K. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. *SIGPLAN Not. 48*, 7 (Mar. 2013), 1–12.

[35] K. TROSHINA, A. C., AND DEREVENETC, E. C decompilation: Is it possible? In *Proceedings of International Workshop on Program Understanding. Altai Mountains, Russia* (2009), pp. 18–27.

[36] KAINTH, M., KRISHNAN, L., NARAYANA, C., VIRUPAKSHA, S. G., AND TESSIER, R. Hardware-assisted code obfuscation for FPGA soft microprocessors. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (San Jose, CA, USA, 2015), DATE '15, EDA Consortium, pp. 127–132.

[37] KINDER, J., AND VEITH, H. Jakstab: A static analysis platform for binaries. In *in CAV, ser. LNCS* (2008), Springer, pp. 423–427.

[38] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 351–366.

[39] KŘOUSTEK, J., POKORNÝ, F., AND KOLÁŘ, D. A new approach to instruction-idioms detection in a retargetable decompiler. *Computer Science and Information Systems (ComSIS) 11*, 4 (2014), 1337–1359.

[40] KŘOUSTEK, J. *Retargetable Analysis of Machine Code*. PhD thesis, Faculty of Information Technology, Brno University of Technology, CZ, 2015.

[41] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (May 2014), pp. 276–291.

[42] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011* (2011), The Internet Society.

[43] LIM, J., AND REPS, T. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst. 35*, 1 (Apr. 2013), 4:1–4:59.

[44] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium* (West Lafayette, IN, 2010), CERIAS '10, CERIAS - Purdue University, pp. 5:1–5:1.

[45] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 290–299.

[46] MATULA, I. P. Decompilation of C++ binaries. Master's thesis, Faculty of Information Technology (FIT) of Brno University of Technology, 2014.

[47] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[48] NAEEM, N., BATCHELDER, M., AND HENDREN, L. Metrics for measuring the effectiveness of decompilers and obfuscators. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on* (June 2007), pp. 253–258.

[49] OULSNAM, G., AND OF QUEENSLAND. DEPARTMENT OF COMPUTER SCIENCE, U. *Unravelling Unstructured Programs*. Technical report (Queensland). University of Queensland, Department of Computer Science, 1980.

[50] PERRIOT, F., AND FERRIE, P. Principles and practices of x-raying. In *Virus Bulletin Conference* (2004).

[51] POLINO, M., SCORTI, A., MAGGI, F., AND ZANERO, S. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Almgren, V. Gulisano, and F. Maggi, Eds., vol. 9148 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 121–143.

[52] POPOV, I. V., DEBRAY, S. K., AND ANDREWS, G. R. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 19:1–19:16.

[53] POTHARAJU, R., NEWELL, A., NITA-ROTARU, C., AND ZHANG, X. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Engineering Secure Software and Systems*, G. Barthe, B. Livshits, and R. Scandariato, Eds., vol. 7159 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 106–120.

[54] PRASAD, M., AND CKER CHIUEH, T. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference* (2003), pp. 211–224.

[55] RIBIC, S., AND SALIHBEGOVIC, A. High level language translator with machine code as representation of the source code. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on* (June 2007), pp. 777–782.

[56] ROUNDY, K. A., AND MILLER, B. P. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv. 46*, 1 (July 2013), 4:1–4:32.

[57] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[58] SNIDER, D. IC decapsulation - exposing semiconductor devices for analysis. `http://www.ial-fa.com/blog/ic-decapsulation-exposing-semiconductor-devices-for-analysis`, September 2014.

[59] SZOR, P., AND FERRIE, P. Hunting for metamorphic. `https://www.symantec.com/.../hunting.for.metamorphic.pdf`, 2003.

[60] U.S. COURT OF APPEALS, F. C. Harold l. bowers (doing business as hlb technology), plaintiff-cross appellant, v. baystate technologies, inc., defendant-appellant. `https://law.resource.org/pub/us/case/reporter/F3/320/320.F3d.1317.01-1109.01-1108.html`.

[61] VAN EMMERIK, M., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (Washington, DC, USA, 2004), WCRE '04, IEEE Computer Society, pp. 27–36.

[62] ĎURFINA, L., KŘOUSTEK, J., ZEMEK, P., AND KÁBELE, B. Detection and recovery of functions and their arguments in a retargetable decompiler. In *19th Working Conference on Reverse Engineering (WCRE'12)* (Kingston, Ontario, CA, 2012), IEEE, pp. 51–60.

[63] ĎURFINA, L., KŘOUSTEK, J., ZEMEK, P., KOLÁŘ, D., HRUŠKA, T., MASAŘÍK, K., AND MEDUNA, A. Design of a retargetable decompiler for a static platform-independent malware analysis. *International Journal of Security and Its Applications 5*, 4 (2011), 91–106.

[64] VIRTA LABS. Powerguard: Security analytics outside the box. `https://www.virtalabs.com/`.

[65] WANG, S., WANG, P., AND WU, D. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 627–642.

[66] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with KINT. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 163–177.

[67] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)* (Orlando, Florida, December 2012), pp. 299–308.

[68] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy 5*, 2 (Mar. 2007), 32–39.

[69] YAKDAN, K., ESCHWEILER, S., AND GERHARDS-PADILLA, E. REcompile: A decompilation framework for static analysis of binaries. In *Malicious and Unwanted Software: "The Americas" (MALWARE), 2013 8th International Conference on* (Oct 2013), pp. 95–102.

[70] YAKDAN, K., ESCHWEILER, S., GERHARDS-PADILLA, E., AND SMITH, M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security (NDSS), ISOC* (2015).

[71] ZENG, J., FU, Y., MILLER, K. A., LIN, Z., ZHANG, X., AND XU, D. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 487–498.

[72] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 337–352.