Sound C Code Decompilation for a subset of x86-64 Binaries

Freek Verbeek[1,2], Pierre Olivier[3], and Binoy Ravindran[1]

[1] Virginia Tech, Blacksburg VA , USA
[2] Open University of The Netherlands
[3] University of Manchester, UK

Abstract. We present FoxDec: an approach to C code decompilation that aims at producing sound and recompilable code. Formal methods are used during three phases of the decompilation process: control flow recovery, symbolic execution, and variable analysis. The use of formal methods minimizes the trusted code base and ensures soundness: the extracted C code behaves the same as the original binary. Soundness and recompilablity enable C code decompilation to be used in the contexts of binary patching, binary porting, binary analysis and binary improvement, with confidence that the recompiled code's behavior is consistent with the original program. We demonstrate that FoxDec can be used to improve execution speed by recompiling a binary with different compiler options, to patch a memory leak with a code transformation tool, and to port a binary to a different architecture. FoxDec can also be leveraged to port a binary to run as a unikernel, a minimal and secure virtual machine usually requiring source access for porting.

1 Introduction

Research in program analysis, verification, and engineering often assumes a context where source code is available. However, numerous safety-critical systems in automotive, aerospace, medical and military domains are built out of components whose source code is unavailable [41]. In the case of proprietary software, a customer is dependent on the vendor for maintenance, patching, and verification [42]. In such contexts, *decompilation* can be useful. Decompilation ideally produces *sound* (functionally equivalent to the binary) and *recompilable* C code.

The majority of existing decompilation tools do not satisfy these two properties [11,1,15,25,33,16,19,17,14]. These papers do not evaluate soundness, focusing on other metrics such as readability and code size [9]. It is a well-known issue in many existing tools that decompiled C code is not functionally equivalent to the binary [9,42]. The only exception is Phoenix [9], which provides decompilation based on semantics-preserving control-flow recovery. Phoenix, however, does not define its soundness, nor does it provide a soundness proof of its control-flow recovery algorithm. We refer to Section 5 for a more detailed comparison to existing decompilation tools.

This paper presents FoxDec (Formal x86-64 Decompilation): sound and recompilable C code decompilation from x86-64 binaries. During three key stages of decompilation formal methods are used. First, we formally verified a control-flow recovery algorithm using the Isabelle/HOL theorem prover [34]. Second, we present a symbolic execution engine that uses rewrite rules – formally proven correct using the Isabelle/HOL theorem prover – to aggregate small state-changes induced by assembly instructions to higher-level programming constructs. Third, we show how the Z3 theorem prover [13] can be used to formally establish a relation between symbolic memory regions in the binary and variables in the decompiled C code. Taken together, these three uses of formal methods increase the trustworthiness that the decompiled C code is sound.

We show that soundness and recompilability allow FoxDec to be useful in the following contexts, each of which is discussed on Section 3:

Binary Patching When source code is unavailable, performing a patch at the machine code or assembly level is highly complex [42]. By decompiling a binary as C code, one can patch at the source code level, which is significantly easier. As an example, we take a binary with a memory leak. We decompile to C, apply the code transformation tool Coccinelle [37] to patch the leak, and recompile [37].
Binary Porting Using FoxDec, one can take an x86-64 binary, decompile, then recompile it for any other little-endian architecture. It can thus be an alternative to software emulators such as QEMU [5], which suffers from significant slowdowns ranging from 5 to 1000x [10]. FoxDec also enables porting binaries to run as *unikernels* [28]. Most of the existing unikernel models require recompilation or relinking to port an existing application, thus requiring source code [35]. As examples, we port binaries from x86 to ARM, and a binary of the PARSEC [6] Blackscholes program to a unikernel.
Binary Analysis Verification and analysis tools typically operate on source code. The low-level intricacies occurring in binaries make analysis difficult. We show that through C code decompilation, FoxDec enables the application of standard off-the-shelf source code analysis tools on binaries. For example, we use Frama-C to determine ranges for variables and check for buffer overflows in the binary of the GNU Coreutils word-count (`wc`) program [24].
Binary Improvement Different compilers offer variable program performance, compilation speed, binary sizes, etc., which vary with the compiled program and the compilation options. C decompilation enables recompiling a binary with different settings. As an example, we show that it is possible to improve execution speed of functions in a binary containing implementations of floating-point functions, simply by decompiling and recompiling them.

The approach has limitations. User-interaction is required for 1.) providing information on function signatures, and 2.) inclusion of header files. Soundness of a specific step – namely, introducing references to variables – cannot be

```
 1: push rbp
 2: mov rbp, rsp
 3: mov dword ptr [rbp - 0x10], edi
 4: mov dword ptr [rbp - 0x8], 1
 5: mov dword ptr [rbp - 0x4], 0
 6: mov dword ptr [rbp - 0xc], 0
 7: sub rbp, 8
 8: jmp 12
 9: shl qword ptr [rbp], 1
10: imul qword ptr [rbp], 3
11: add dword ptr [rbp - 0x4], 1
12: mov eax, dword ptr [rbp - 0x4]
13: cmp eax, dword ptr [rbp - 0x8]
14: jb 9
15: mov rax, qword ptr [rbp]
16: pop rbp
17: ret
```

Fig. 1. Running Example

guaranteed since without type-information it is undecidable whether a value is a pointer. In that case, the decompiled variable reference is annotated with a soundness warning and further user-interaction is mandated. Moreover, we cannot deal with indirect branching and thus consider only a subset of all possible x86 binaries.

The research contributions of this paper are: 1.) C code decompilation such that for key stages soundness criteria have been formalized; 2.) the demonstration that this produces efficiently executable code: to the best of our knowledge, no related work exists that provides numbers on execution speed of the recovered code; 3.) the demonstration that sound and recompilable C code recovery can be used for binary patching, porting, analysis, and improvement. To the best of our knowledge, no previous decompilation tool targets soundness, recompilability *and* is based on formal methods. Project information can be found at: `https://llrm-project.org/`; all code and proofs are available at: `https://doi.org/10.5281/zenodo.3952034`.

2 C Code Extraction

We demonstrate the steps of decompilation on the assembly code in Figure 1. The code first initializes local variables. It then loops over lines 12-14-9-12 until the carry flag is false (`jb` looks at that flag).

2.1 Step 1: Binary to Control-Flow Graph

The Control-Flow Graph (CFG) is a graph with basic blocks (i.e., lists of instructions) as vertices and edges with labels from a set F of flags. In order to obtain a CFG from a binary, two steps are required: *disassembly* and *CFG recovery*. Both these steps have been extensively studied in literature [22,3,7,44,23,2,4].

In order to express the soundness of a CFG, we use the notion of paths. A *path in the binary* is defined as any list of instructions such that there exists a possible execution of the binary that visits exactly these instructions. Let I denote the set of instructions, and let $[I]$ denote lists of instructions. We use $\text{is\_path}_{\mathsf{bin}}(\pi)$ to denote that a list π of instructions of type $[I]$ is a path in the binary. A *path in the CFG* is a list of lists of instructions. Let $\text{is\_path}_{\mathsf{cfg}}(\pi, g)$ denote that π of type $[[I]]$ is a path in CFG g. The following notion of soundness is a reformulation of the concept of an ideal CFG from [44].

CFG Soundness: CFG g is sound, if and only if:

$$\text{is\_path}_{\mathsf{bin}}(\pi) \iff \exists \pi' \cdot \text{is\_path}_{\mathsf{cfg}}(\pi', g) \wedge \text{flatten}(\pi') = \pi$$

FoxDec implementation & argument for soundness: We use `Ramblr` for disassembly [42]. A generic and provably sound approach to disassembly is outside the scope of this paper. However, we limit the applicability of FoxDec to binaries without indirect branching. As result, the disassembler knows at all times at which addresses it needs to disassemble instructions. Under this limitation, disassembly can be done in a provably sound way. Similarly, CFG extraction is done by a reimplementation similar to angr's CFGFast [40]. The algorithm straightforwardly produces a CFG, by starting at a known entry point and considering for each encountered instruction its effect on the instruction pointer. Again, the limitation of no indirect branching ensures soundness.

2.2 Step 2: CFG to Abstract Code

We formulate a datatype for *abstract code* that is able to represent the decompiled program at all stages of decompilation. This datatype expresses a program as a combination of control flow, basic blocks, and branching decisions. Each basic block is represented by a polymorphic type β; branching decisions are represented using a polymorphic type ϕ.

$$\begin{aligned}\mathsf{acode}(\beta,\phi) := {} & \texttt{Block}\ \beta \mid \texttt{Skip} \mid \texttt{Continue} \mid \texttt{Break}\ \mathsf{ID} \mid \mathsf{acode}(\beta,\phi)\ ;\ \mathsf{acode}(\beta,\phi) \\ & \mid \texttt{If}\ \phi\ \texttt{Then}\ \mathsf{acode}(\beta,\phi)\ \texttt{Else}\ \mathsf{acode}(\beta,\phi)\ \texttt{Fi} \\ & \mid \texttt{Loop}\ \mathsf{acode}(\beta,\phi)\ \texttt{Pool}\ \texttt{Resume}\{(\mathsf{ID}, \mathsf{acode}(\beta,\phi))\}\end{aligned}$$

Abstract code consists of basic blocks, skips, sequential execution, if statements, and loops. There is only one type of loop that has no exit condition and thus loops infinitely if it does not contain a break. A `Continue` has the same semantics as the C continue statement, i.e., forcing the next iteration of a loop. A `Break` is also similar to the C break, but it optionally has an argument. In case of a loop with multiple exit points, the `Break` can use an ID to identify which exit has been used. After the loop, a `Resume` statement can execute code based on which exit was taken. For example, if a loop breaks due to a `Break` i statement and the loop is followed by a `Resume` that contains the pair (i, a), then abstract code a is executed. If the set of `Resume` is empty, we will omit it.

Let $a = \mathsf{acode}(\beta, \phi)$ be abstract code. A *path of the abstract code* is a list of elements of type β. Let $\text{is\_path}_{\mathsf{ac}}(\pi, a)$ denote that π of type $[\beta]$ is a path of abstract code a.

Step 2 consists of generating abstract code with $\beta = [I]$ and $\phi = F$, i.e., with the same basic blocks and branching decisions as the CFG. It is thus a function cfg_to_ac that takes as input a CFG and produces an element a of type $\mathsf{acode}([I], F)$.

Abstract Code Extraction Soundness: Abstract code extraction is sound, if and only if:

$$\text{is\_path}_{\mathsf{cfg}}(\pi, g) \Longleftrightarrow \text{is\_path}_{\mathsf{ac}}(\pi, \text{cfg\_to\_ac}(g))$$

FoxDec implementation & argument for soundness: Yakdan et al. provide an algorithm for extracting control flow structures from a CFG [45]. The algorithm considers a certain subgraph. Initially, this subgraph is the entire CFG, but for each loop the body forms a new subgraph. This recursive nature allows extraction of nested loops. The function breaks down the current subgraph into sequential statements. Edges back to the entry node of the subgraph are `Continue` statements, edges exiting the subgraph are `Break` statements.

We modeled an adopted version of the method of Yakdan et al. in the Isabelle/HOL theorem prover. This provides a formalized function cfg_to_ac, which was proven sound (the proof files are made available). Subsequently, we implemented the exact algorithm as formalized in Isabelle/HOL.

Example 1. Control flow reconstruction produces the following for the running example:

```
Block 1 −> 12
Loop
  Block 12 −> 14
  If CF Then Block 14 −> 12 Else Break Fi
Pool
Block 14 −> ret
```

A block `l` −> `l′` consists of the instructions from `l` up to (excluding) `l′`. One loop has been identified, which is exited if the carry flag is false. The final block runs until and including the `ret` instruction.

2.3 Step 3: Symbolic Execution

The next step is to transform the basic blocks in the abstract code to symbolic state changes. We achieve this by running symbolic execution. The purpose is twofold: a.) to aggregate the state changes induced by the individual instructions to a set of larger state changes, and b.) to express those state changes in a more architecture-independent fashion.

Formally, symbolic execution is a function symb that takes as input a basic block b of type $[I]$ and produces an element of type $\{A_{SP}\}$. Here elements of type A_{SP} are *assignments* over state parts, i.e., $A_{SP} = \{(SP, E_{SP})\}$. An assignment is denoted by $sp \coloneqq v$. The left-hand-side is an element of type SP, i,e, a state part. A state part is either a register, a flag, or a *memory region* represented by an address a and a number of bytes s. The latter is denoted by $[a, s]$. An assignment $[a, s] \coloneqq v$ of a value to a memory region is done in little-endian fashion, i.e., value v is split up into a byte list and then reversed. All assignments are mutually independent.

The right-hand side is an element of type E_{SP}, i.e., expressions over state parts. These expressions consist of common bit-vector operations including taking bit subsets and concatenation as introduced above, logical operators, casting operators, and floating-point, signed and unsigned arithmetic operators. The expression $*[a, s]$ denotes dereferencing: reading s bytes from memory address a.

To express soundness, we need to compare 1.) the actual behavior of executing assembly instructions with 2.) the semantics of the symbolic expressions. First, let $\text{exec}(b, \sigma)$ denote execution. It takes as input a basic block b of type $[I]$ and a concrete state σ and runs each instruction consecutively. Function calls are treated symbolically (this models non-deterministic user-input). Second, let $\text{eval}_{\mathsf{E}}(e, \sigma)$ denote an evaluation function for expressions. It evaluates the expression as much as possible, but again leaves results of function calls symbolic. The execution of an assignment evaluates both the left and the right-hand side in the current state, and then updates the current state. This leads to a function $\text{eval}_{\{\mathsf{A}_{SP}\}}$ that evaluates a set of assignments, i.e., that evaluates symbolized basic blocks.

Symbolic Execution Soundness: Symbolic execution is sound, if and only if, for any basic block b of type $[I]$:

$$\forall \sigma \ \cdot \ \text{exec}(b, \sigma) = \text{eval}_{\{\mathsf{A}_{SP}\}}(\text{symb}(b), \sigma)$$

This notion of soundness is context-insensitive, i.e., it considers each block separately. Let $\text{symb}_{\mathsf{ac}}(a)$ apply symbolic execution to all blocks. It thus takes as input abstract code a of type $\mathsf{acode}([I], F)$ and produces symbolized abstract code of type $\mathsf{acode}(\{A_{SP}\}, E_{SP})$. This ensures that:

$$\text{is\_path}_{\mathsf{ac}}(\pi, a) \Longleftrightarrow \text{is\_path}_{\mathsf{ac}}(\text{map symb } \pi, \text{symb}_{\mathsf{ac}}(a))$$

Here map is the standard map function. Moreover, soundness of symbolic execution implies that for any path π of type $[[I]]$ (produced by Step 2):

$$\forall \sigma \ \cdot \ \text{exec}(\text{flatten}(\pi), \sigma) = \text{eval}_{\mathsf{path}}(\text{map symb } \pi, \sigma)$$

Here function $\text{eval}_{\mathsf{path}}$ consecutively runs evaluation on the given list of symbolized basic blocks. It formulates that executing paths from the CFG is equivalent to evaluating the symbolized abstract code.

FoxDec implementation & argument for soundness: The key elements of symbolic execution are *instruction semantics* and *rewrite rules*. Instruction semantics consists of a set of symbolic assignments per instruction. For example, the `add` instruction updates flags and its first operand with symbolic expressions. We use the formal semantics of Roessle et al [39]. They leverage the work of Heule et al. [21] which produces machine-learned semantics for a large set of instructions. Their semantics have been proven to be highly reliable. Roessle et al. translated these into a bitvector language and formalized them in the Isabelle theorem prover [34]. We have taken the semantics of this model and programmed them as symbolic assignments. Aggregating the semantics of individual instructions leads to large expressions. Simplification rules are necessary to maintain scalability and readability. We have written a simplification engine that uses arithmetic, logical, and bit-vector based simplification rules. Each of these rules has been proven correct in the Isabelle/HOL theorem prover.

Example 2. Symbolic execution produces the following for the running example:

```
Block {rbp ≔ rsp − 8, ∗[rsp − 8, 4] ≔ 1, ∗[rsp − 16, 4] ≔ edi, ...}
Loop
  Block {CF ≔ ∗[rbp − 4, 4] > ∗[rbp − 8, 4], ...}
  If CF Then Block {∗[rbp, 8] ≔ ∗[rbp, 8] ∗ 6, ...} Else Break Fi
Pool
Block {rax ≔ ∗[rbp, 8], ...}
```

First, one can see that the basic blocks have become sets of mutually independent assignments. For example, the first block assigns the 4-byte value 1 to address $\texttt{rsp} - 8$. This is due to instructions 2 and 4. Second, one can see that semantics have been aggregated, e.g., multiplication by 6 instead of left-shifting and times 3. Also, instructions 12 and 13 have been aggregated into a single assignment to the carry flag.

2.4 Step 4: Variable Analysis

The key purpose of variable analysis is to establish which memory regions correspond to which variables. Three types of variables exist: local, global, or heap variables. Local variables are stored in the stack frame, relative to either the stack pointer (`rsp`) or the frame pointer (`rbp`). Global variables are stored in the data sections of a binary. Their addresses are typically immediates, i.e, constant values that represent a certain offset with respect to where the binary is stored in memory. Heap variables are represented by pointers.

Before memory regions can be matched to variables, any address computation must be expressed relative to the initial state. Consider the running example. Within the loop, region $[\texttt{rbp}, 8]$ actually overlaps with region $[\texttt{rsp} - 8, 4]$ from the first block. This is because during the loop, the following invariant holds: $\texttt{rbp} = \texttt{rsp}_0 - 8$. Expressed in initial values it is easy to see these two regions should be mapped to the same variable: $[\texttt{rsp}_0 - 8, 8]$ and $[\texttt{rsp}_0 - 8, 4]$.

The first step of variable analysis is thus *invariant propagation.* In the running example, the invariant assigned to line 1 will contain $\texttt{rsp} = \texttt{rsp}_0$. At line 12, it will contain $\texttt{rbp} = \texttt{rsp}_0 - 8$. In similar fashion, initial values of the form x_0 are propagated for all registers and memory regions. After invariant propagation, it is checked whether all addresses are expressed in terms of initial values.

The second step replaces memory regions with variables. Local variables are identified by finding regions whose address computation includes the stack- or frame pointer. For example, the symbolic expression $*[\texttt{rsp}_0 - \texttt{4}, 4]$ is replaced by a symbolic expression consisting of some 32-bit local variable lv. Global variables are identified by finding symbolic expressions of the form $[i, s]$ with both i and s immediates. For example, the symbolic expression $*[4257968, 8]$ is replaced by a 64-bit global variable gv. For each global variable, we retrieve the initial value of the global variable from the data sections of the binary. The remaining set of memory regions constitute heap variables. These require no modification during variable analysis. For example, $*[\texttt{rax}, 4]$, dereferencing the pointer in register `rax`, is not modified by this step.

Generally, regions may be different but still map to the same variable. This happens if the regions are *necessarily overlapping.* Two regions $[a, s]$ and $[a', s']$ are necessarily overlapping, notation $[a, s] \sim [a', s']$, if and only if in *any* state addresses a and a' resolve to regions that share at least one byte. Such regions are merged during variable analysis.

Variable Analysis Soundness: Assume that the addresses of all memory regions are expressed in terms of the initial state. Let $\mathit{var}[a, s]$ return the variable that is being substituted for region $[a, s]$. Variable analysis is sound, if and only if, for any two accessed memory regions $[a, s]$ and $[a', s']$, anywhere in the abstract code:

$$[a, s] \sim [a', s'] \iff \mathit{var}[a, s] = \mathit{var}[a', s']$$

FoxDec implementation & argument for soundness: Invariants are established via a standard forward propagation algorithm [18]. For loops, the current invariant is iteratively weakened until a fix-point is reached. To establish whether two regions are necessarily overlapping, the Z3 theorem prover is used [13]. The regions are necessarily overlapping if them being separate is unsatisfiable.

Example 3. Variable analysis produces the following for the running example:

```
Block {$lv_0 := 1, lv_1 := 0, lv_2 := p_0 \ldots$}
Loop
  Block {$b := lv_1 > lv_2, \ldots$}
  If $b$ Then Block {$lv_0 := lv_0 * 6, \ldots$} Else Break Fi
Pool
Block {$ret := lv_0$]}
```

During the loop $\texttt{rbp} = \texttt{rsp}_0 - 8$. As result, the region $[\texttt{rbp}, 8]$ in the blocks in the loop were necessarily overlapping with the regions $[\texttt{rsp} - 8, 4]$ and $[\texttt{rsp} - 4, 4]$ in the first block. All these regions were thus merged into one variable lv_0. Variable b has been introduced as a Boolean variable for the carry flag. Moreover, the function signature maps registers `edi` to parameter p_0, and the function returns a value via variable $\mathit{ret} = \texttt{rax}$.

2.5 Step 5: References

After variable analysis, there can still be symbolic expressions that are references to variables. Step 5 identifies such expressions and replaces them. For example, if region $[\texttt{rsp}_0 - 4, 4]$ has been substituted with variable lv, symbolic expression $\texttt{rsp}_0 - 4$ is replaced by symbolic expression $\&lv$. In case the region was not encountered, a fresh variable lv_f is introduced and the region is translated as a reference to that fresh variable. Since the size cannot be established, it is assumed to be 8 bytes. The code is annotated with a possible unsoundness.

References to global variables pose a problem with respect to automation. Consider the occurrence of an immediate 4257968, and assume that this immediate value is within the address range of the data sections of the binary. This may or may not be the address of some global variable. By default, this is considered *not* to be a reference to a global variable. However, the code is annotated with a message indicating that an immediate occurred whose value indicates that it is likely to be a global variable pointer. The user can then manually modify a config file, ensuring that 4257968 is translated to $\&gv$.

2.6 Step 6: C code generation

The abstract code produced by variable analysis is the base for C code generation. The main difference between the abstract code and C is *types*. For the abstract code, we know the size of each variable, but not the type.

Consider the C expression $a - b/c$. Its semantics depend on whether the variables are floats, signed, or unsigned ints. The $+$, and $-$ operator behave the same for unsigned and signed ints, but division does not. The semantics of the operators are thus dependent on the types of the variables. In the abstract code, this is not the case. The operators are well-defined, e.g., the operator in the symbolic expression is the result of a `div`, `idiv`, or `fdiv` assembly instruction (division of signed, unsigned, floating point). This means that we do not need type information for variables.

We thus rely on *type punning* to translate variables. Consider a 64-bit variable. Depending on which operator is applied to it, it must be accessed in different ways. This is achieved using a `union`. When translating an operator, we first establish the types expected by the operator. For example, an `fdiv` assembly instruction expects two doubles. A `sub` expects (un)signed ints (note that subtraction is equal for unsigned/signed arithmetic). We arbitrarily pick unsigned out of the two options. We then translate the operands and *pun* them – if necessary – to the types expected by the operator. Punning one type to another means casting without modifying its contents.

Example 4. The running example produces the following C:

```
var64_t f(var32_t p0) {
  var64_t lv0.u = 1; var32_t lv1.u = 0, lv2 = p0;
  while (1) {
    bool b = lv1.s > lv2.s;
    if (b) lv0.s = lv0.s * 6; else break;
  }
  return lv0;
}
```

In contrast to using type punning and unions to derive C, it is also possible to use type inference [31]. The major advantage of using type inference with respect to our approach is that the produced C code is much more readable and humanly understandable. The drawback is that it is not always possible (i.e., it is undecidable). Type punning thus produces in more cases code that is recompilable, at the cost of readability.

3 Applications

We have applied C code extraction to binaries containing functions of the FDLIBM library. We have considered both non-optimized code (`O0`) and fully optimized (`O3`). FDLIBM provides mathematical floating-point functions such as `sin`, `fmod` and `log` that satisfy the IEEE754 standard. We have chosen FDLIBM as a case-study for the following reasons:

Complexity: The functions provide highly optimized implementations of advanced mathematical functions. They contain advanced flow control, including nested loops and if-statements, goto's, switches, and both in- and external function calls. They execute a plethora of floating-point, signed and unsigned arithmetic operations and frequently cast between types. Advanced pointer-arithmetic is used, both on arrays and on addresses of variables (e.g., splitting up the high and low part of a 64-bit floating point variable). Type punning is used, even in the original (vanilla) C code.

Testability: Even though the implementation is complex, the interfaces to the functions are all simple. They generally are side-effect free functions that take either one or two floating-points parameters and return a floating point. Some functions do produce side effects, such as writing certain results into an array that is passed as a parameter. For all functions, these are well-documented. The functions are typically fast, allowing execution of millions of test-cases.

The FDLIBM library comprises 84 functions and 8.543 lines of C code. 23 of these functions are simple wrappers that only call other functions. We exclude them from our evaluation, since they do not do anything interesting on their own. One function, `__ieee754_lgamma_r`, uses indirect branching, which is not supported. The text sections of the remaining 61 functions comprise 13.744 (10.221) lines of assembly code (respectively `O0` and `O3`), not including any data sections. For each of these 61 functions, we have decompiled C code. In this case study, all warnings on pointers to global variables were trustworthy, i.e., they actually were pointers and not constants. At four places, the decompiled C code has been modified due to fixed-size local arrays. The decompiled C code has been recompiled. For each function f we thus have the vanilla function f_v in the original binary and the function f_r in the recompiled binary.

Test setup Testing consists of running both functions f_v and f_r on randomly generated values and comparing their results. For each function, we have run 100.000.000 test cases. All experiments in this section were run on a 1.2 GHz Intel Core m3 machine with 8 GB RAM. The test setup first requires generation of random values of type `double`. We build a random-generator that at the top-level randomly (but uniformly) calls one of five different random double generators. Four of the random double generators randomly pick a double from 1.) the range $[-\texttt{2.0}, \texttt{2.0}]$, 2.) the range $[-\texttt{20000.0}, \texttt{20000.0}]$, 3.) the range of subnormals, 4.) a set of specific values such as plus- and minus infinity, plus- and minus zero and one, plus- and minus NaN, and values such as maximal and minimal doubles. The fifth random double generator produces a bit pattern from 64 random Bools and casts it to a double. The purpose of this test setup is thus to test random values, likely values and corner cases.

Results We measure running times of the vanilla and the recompiled version for a comparison. For each function, an array of 50.000 elements is initialized with random test data for 50.000 test cases. Two arrays are allocated to store output data. Subsequently, a loop is run that calls the vanilla function f_v for each test case and writes its output to one of the output arrays. Then, a loop is run for the recompiled function f_r. Of both loops, the running times are measured. After having run both loops, the output data is compared for equality, to ensure that the functions produced the same results. This is then repeated until a total of 100.000.000 test cases have been run, accumulating the running times. The reason to do it in batches of 50.000 is to prevent measurement inaccuracy. One batch of 50.000 function calls typically takes about 5.000 microseconds, which is accurately measurable.

Figure 2 shows running times for 50 out of 61 functions. The remaining 11 functions are each called by at least one of the 50 functions and therefore not shown. Let t_v and t_r denote respectively the running times of all test cases on the vanilla and recompiled functions. The graph shows the ratio $\frac{t_r}{t_v}$. The series `O0/O0` is the case where the vanilla binary had been compiled without optimizations, and the decompiled code is compiled without optimizations as well. The regenerated code is on average a factor 1.5 slower then the vanilla. There are two notable exceptions: `__ieee754_pow` and `cbrt`. The recompiled version of the first is, for all four series in Figure 2, about a factor 40 times slower. This function consists of a series of if-statements (no loops). Its CFG consists of a largely irreducible CFG of 93 basic blocks and 144 edges, leading to an exponential blow-up during control flow extraction. The second is a factor 9.9 slower in this series.

If, however, the decompiled code is compiled with optimizations, then the decompiled code is generally either faster than the vanilla, or is at most 50% slower. This is shown by series `O0/O3` and `O3/O3` (respectively for a non-optimized and fully optimized original binary). As expected, the series `O3/O0`, which decompiles code from an optimized binary and recompiles it without optimizations, produces decompiled code that is significantly slower than its original.

3.1 Use-cases of C Decompilation

Binary Improvement: Decompiling as C code allows us to recompile a program with higher optimizations or a different compiler in order to get performance gains. We used the PARSEC [6] Blackscholes benchmark which is representative of modern compute-intensive data analytics applications. We compile the vanilla source code with `gcc` 6.3 and the default optimization level, i.e. `O0`. We then decompile that binary and recompile the resulting code varying the compiler and the level of optimizations. We use `gcc` 6.3 and `clang` 3.8, the default versions available on Debian 9, as well as `gcc` 8.3 and `clang` 9.0 which are

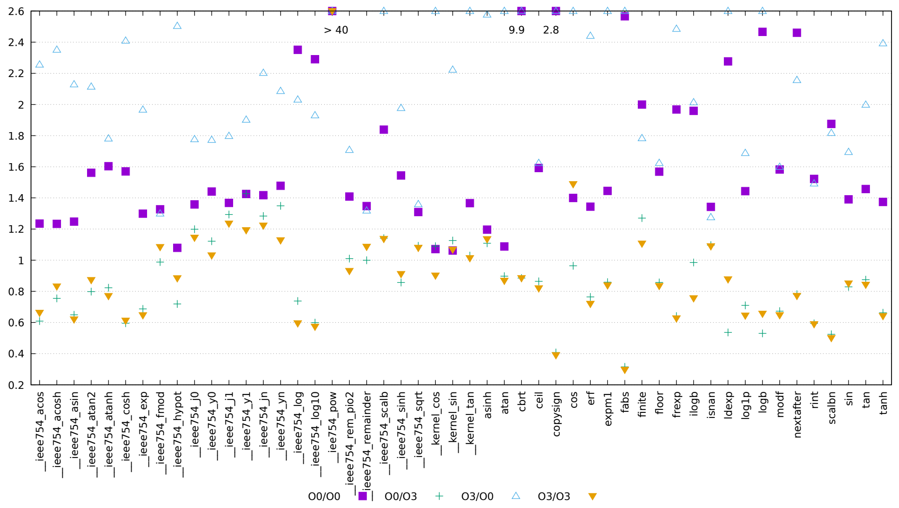

Fig. 2. Ratios of execution speed, capped of at a factor 2.6. Series x/y means that the original binary had been compiled with optimization x, and the recompiled C is compiled with optimization y. For series `O0/O0`, factors higher than 2.6 have been labelled. Excluding `__ieee754_pow`, the average factor for series `O3/O0` is 2.3 with a maximum of 6.6.

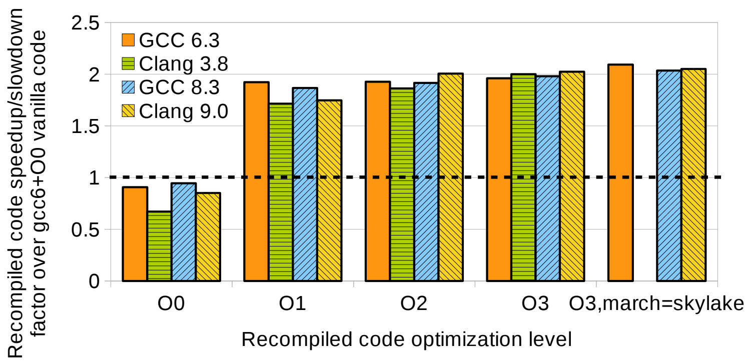

Fig. 3. Original and recompiled performance of PARSEC Blackscholes.

the latest versions released. We varied the optimization level from `O0` to `O3`, and added a fourth case with both `O3` and `march=skylake` which produces binaries optimized for the particular CPU used for this test: a Xeon E3-1270 v5 clocked at 3.6 GHz. Blackscholes's largest data set was used (native size).

We compare the speedup/slowdown of the recompiled code over the original Blackscholes code compiled with `gcc` 6.3/`O0`. As one can observe, the capacity to decompile and recompile with `O1` or higher can bring significant speedups. Across all compilers, the average speedup is 1.81x for `O1`, 1.93x for `O2`, 1.99x for

`O3` and 2.01x for `O3` with `march=skylake`. Note that `clang` 3.8 with this last option produced invalid code so no result is presented for that particular case.

Binary Patching Coccinelle is a C transformation tool use for software patching [36]. It takes as input a *semantic patch* and one or a set of C files, and produces a patch. As example, we take a binary that contains a function allocating an array on the heap with `malloc()`, then initializing it and performing some computations within that array. The function returns without calling `free()` on that array which is a memory leak. We write a Coccinelle semantic patch formulating that the array manipulated by that function should be freed before it returns. We decompile C from the binary, apply the patch, and recompile, producing a patched binary.

Binary Porting We have run the x86 binary containing the FDLIBM library on an ARM64 Cavium ThunderX server machine, by recompiling the decompiled C, and successfully reran all test cases. Moreover, we have taken an x86 binary containing the Blackscholes benchmark and run it as a unikernel. Unikernels [29] are minimal and single-purpose VMs tailored for cloud execution and presenting numerous benefits such as low image size, memory footprint, and fast instantiation times. Porting an application to unikernel models typically requires access to the application's sources [35]. Using FoxDec we can decompile an application for which the sources are not available, and recompile it as a unikernel.

We decompiled PARSEC Blackscholes and recompiled it as a unikernel, using the HermitCore [27] unikernel model. To demonstrate unikernel benefits, we measured the image size, boot time, and memory footprint of the resulting VM. We compared these numbers to a regular Linux VM (a minimal Ubuntu 16.04 from Vagrant repositories), which can be used to deploy in the cloud an application whose sources are not available. The image size, boot time, and memory footprint for the recompiled unikernel vs. the Linux VM were resp 2.1MB vs. 781MB, 20ms vs. 26ms, and 12MB vs. 87 MB. We came to similar data for a Docker container [30] (resp. 116MB, 1500ms, 2MB).

Binary Analysis Frama-C is a tool suite dedicated to source code analysis of C software. It can be used for, among others, program slicing, test-case generation, and verification. We have applied Frama-C to the binary of the word-count program. It indicated one loop with a possible buffer overflow. Manual analysis showed that this was a false negative.

4 Discussion and Limitations

This section summarizes issues related to soundness and automation and discusses limitations.

Let a be the abstract code produced after Step 3 (symbolic execution). Soundness of these steps provides as a corollary:

$$\text{is\_path}_{\mathsf{bin}}(\pi) \iff \exists \pi' \;\cdot\; \text{is\_path}_{\mathsf{ac}}(\pi', a) \wedge \forall \sigma \;\cdot\; \text{exec}(\pi, \sigma) = \text{eval}_{\mathsf{path}}(\pi', \sigma)$$

Executions of the binary are represented by the symbolized abstract code. Steps 4 to 6 concern the translation to C. Sound variable analysis ensures that the assignments in the symbolized state blocks are executed on the proper variables.

Table 1. Decompilation tools. FM = Based on Formal Methods; RC = recompilability

Name	Output	FM	RC	Types	Supports	Soundness
FoxDec	C	✓	partial	punning	x86-64 with SIMD	yes
Phoenix	C		partial	yes	32 bit: x86	yes
Ghidra	C		no	partial	All	
RetDec	C / Python		partial	yes	32 bit: x86, ARM, MIPS	
Ramblr	Assembly		yes		x86-64	
DiL	HOL	✓	no		32 bit: x86, ARM, MIPS	
IDA PRO	Assembly		no		All	
Hex-Rays	Pseudo C		no	partial	All	
SmartDec	C++		partial	yes	x86-64	
McSema	LLVM IR		yes		All	

Step 5 is possibly unsound, but C code annotations are provided. Step 6 is a matter of translating symbolized expressions to their C equivalent. Universally, if the resulting C code does not contain annotations, it is sound.

The subset of supported binaries is limited to binaries without indirect branching, variadic functions, self-modifying code, `setjmp` / `longjmp`, and concurrency.

Some parts of our C code decompilation are x86-64 specific. The symbolic execution engine is based on semantics for x86 instructions. Variable analysis is largely generic, but it requires knowledge on which registers are used to relate local variables to (in x86: `rsp` and `rbp`). CF extraction and C code generation are generic. We thus argue that implementing this approach for other architectures is a matter of engineering.

5 Related Work

Decompilation, and C decompilation in particular, has been an active research field for decades [20,11,38,12,43,9]. Table 1 provides an overview of some of the available tools. The table does not show disassemblers, such as CapStone or BAP [8]. Column RC provides information on recompilability. Ramblr and McSema provide decompilation into a language lower-level then C, but with recompilation. The bulk of the C code decompilation tools produce pseudo C or C that requires manual inspection for it to be sound and recompilable. The notable exception is Phoenix, discussed below [9]. Column Types provides an overview of how types are dealt with, if applicable. As discussed in the previous section, we use type punning to enable recompilability, at the cost of readability.

To the best of our knowledge, the only existing C decompilation tool that targets soundness is Phoenix [9]. Phoenix uses control-flow recovery and type recovery to produce structured and readable C code. The key difference between FoxDec and Phoenix is that for Phoenix, soundness is not defined, nor is an argument provided on why the approach is sound. For example, an algorithm is provided for control-flow recovery without a soundness criterion or proof. In contrast, this paper is based on formally proven correct control-flow recovery.

Second, Phoenix uses type recovery, producing much better and readable code. However, they themselves state that this leads to soundness failures, whereas type punning does not introduce any soundness issues. Finally, Phoenix works for x86 in 32-bit and does not support floating-point operations. We have not been able to install Phoenix for a direct comparison.

Very recently, Ghidra has been released by the US National Security Agency (NSA) [1]. It is an open-source framework that aims at analyzing malicious code. It provides a robust C multi-architectural decompilation framework. Moreover, it supports indirect branching and its C generation relies not on punning and unions. However, the code it produces is not necessarily sound or recompilable.

RetDec (Retargetable Decompiler) provides an end-to-end binary-to-C decompilation suite [15,25]. Their work considers binaries from various 32-bit architectures and provides decompilation via LLVM. Their output is either C or a Python-like language. Their work has some unique characteristics. First, the use of LLVM enables the application of LLVM based tools. Second, they have put a great effort in producing humanly readable source code. Thirdly, they combine their work with type inference [31], producing C code with arrays and structs. Finally, they provide an *unpacker* as a preprocessor [26], that is able to take a packed binary (e.g., malware) and make it suitable for decompilation.

Ramblr [42] is part of the angr binary analysis framework [40], which provides binary analysis tools such as CFG recovery. The focus of Ramblr is on symbolized disassembly, i.e., deriving assembly with symbolic labels instead of immediate addresses. This allows binary patching, since the assembly can be reassembled.

A special kind of decompilation is *decompilation-into-logic* (DiL) [32,33]. DiL embeds disassembled machine code into Higher-Order-Logic. Blocks are given formal pre- and postconditions and loops are translated to recursive functions. DiL enables formal verification of binaries in a theorem prover.

Various commercial tools for binary analysis and decompilation exists. IDA-PRO [16] supports all mainstream architectures and is build on years of research into binary analysis. An extension of IDA-PRO is Hex-Rays [19]. Hex-Rays extracts humanly readable C-like pseudocode text. It is extremely fast, providing a result virtually on-the-fly for many functions. Other commercial tools include SmartDec [17], which targets C++ code, and McSema [14] which extracts LLVM.

6 Conclusion

This paper presents FoxDec: a C decompilation framework that is based on formal methods and that aims at soundness and recompilability. We show that C decompilation allows the application of source-level tools on binaries. We apply, e.g., a C code patching tool to patch a binary, or we apply a C code verification framework to verify the binary. Moreover, C decompilation can be useful for binary porting. We show this by porting an x86-64 binary to a unikernel and to an ARM machine. We have shown that FoxDec provides decompiled C code with little overhead in terms of execution speed with respect to the original binary. FoxDec will be made available online under an open-source license.

In the near future, we want to strengthen the formal relation between the decompiled C code and the binary. Instead of proving formal correctness of each individual step, we aim at producing a formal certificate that provides a theorem prover with all the information necessary to establish a simulation relation between the binary and C code. Moreover, we want to focus on formally verified type inference algorithms. The result would be a way to largely automatically retrieve formally proven correct C code from a subset of x86-64 binaries.

Acknowledgments. This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-17-1-2297, N00014-16-1-2104, and N00014-18-1-2022.

References

1. National Security Agency. *Ghidra*, 2019. https://www.nsa.gov/resources/everyone/ghidra/.
2. Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, 2016.
3. Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.
4. Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.
5. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.
6. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.
7. Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malware signatures. 2007.
8. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.
9. David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. 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)*, pages 353–368, 2013.
10. Edouard Bugnion, Jason Nieh, and Dan Tsafrir. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.
11. Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.
12. Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 228–237. IEEE, 1998.
13. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

14. Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.
15. Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.
16. Justin Ferguson and Dan Kaminsky. *Reverse engineering code with IDA Pro.* Syngress, 2008.
17. Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: approaching C++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.
18. Steven M German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE transactions on Software Engineering*, (1):68–75, 1975.
19. Ilfak Guilfanov. Decompilers and beyond. *Black Hat USA*, 2008.
20. Matthew S Hecht and Jeffrey D Ullman. Characterizations of reducible flow graphs. *Journal of the ACM (JACM)*, 21(3):367–375, 1974.
21. Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *ACM SIGPLAN Notices*, volume 51, pages 237–250. ACM, 2016.
22. R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.
23. M. A. B. Khadra, D. Stoffel, and W. Kunz. Speculative disassembly of binary code. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, Oct 2016.
24. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
25. Jakub Křoustek. *Retargetable Analysis of Machine Code.* PhD thesis, PhD thesis, Brno, FIT BUT, 2014.
26. Jakub Křoustek and Dušan Kolár. Preprocessing of binary executable files towards retargetable decompilation. In *8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13)*, pages 259–264, 2013.
27. Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, page 4. ACM, 2016.
28. Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4):461–472, 2013.
29. Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4):461–472, 2013.
30. Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
31. Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.
32. M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures – an application of decompilation into logic. In *Formal Methods in Computer-Aided Design*, pages 1–8, Nov 2008.

33. Magnus O Myreen, Michael JC Gordon, and Konrad Slind. Decompilation into logic – improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.
34. Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
35. Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE19), 2019.*, 2019.
36. Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 247–260, New York, NY, USA, 2008. ACM.
37. Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Semantic patches, documenting and automating collateral evolutions in Linux device drivers. In *Ottawa Linux Symposium (OLS 2007)*, Ottawa, Canada, 2007.
38. Todd A Proebsting and Scott A Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *COOTS*, pages 185–198, 1997.
39. Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 181–195. ACM, 2019.
40. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.
41. Sardar Muhammad Sulaman, Alma Orucevic-Alagic, Markus Borg, Krzysztof Wnuk, Martin Höst, and Jose Luis de la Vara. Development of safety-critical software systems using open source software – a systematic map. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 17–24. IEEE, 2014.
42. Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.
43. Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *International Static Analysis Symposium*, pages 170–183. Springer, 2007.
44. Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, 2009.
45. Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.