Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly

Freek Verbeek
freek@vt.edu
Open Universiteit
Heerlen, The Netherlands
Virginia Tech
Blacksburg, USA

Nico Naus
nico.naus@ou.nl
Open Universiteit
Heerlen, The Netherlands

Binoy Ravindran
binoy@vt.edu
Virginia Tech
Blacksburg, USA

Abstract

We present an approach to lift position-independent x86-64 binaries to symbolized NASM. Symbolization is a decompilation step that enables binary patching: functions can be modified, and instructions can be interspersed. Moreover, it is the first abstraction step in a larger decompilation chain. The produced NASM is recompilable, and we extensively test the recompiled binaries to see if they exhibit the same behavior as the original ones. In addition to testing, the produced NASM is accompanied with a certificate, constructed in such a way that if all theorems in the certificate hold, symbolization has occurred correctly. The original and recompiled binary are lifted again with a third-party decompiler (Ghidra). These representations, as well as the certificate, are loaded into the Isabelle/HOL theorem prover, where proof scripts ensure that correctness can be proven automatically. We have applied symbolization to various stripped binaries from various sources, from various compilers, and ranging over various optimization levels. We show how symbolization enables binary-level patching, by tackling challenges originating from industry.

CCS Concepts

• **Security and privacy** → **Logic and verification**; • **Software and its engineering** → Software maintenance tools; Semantics.

Keywords

disassembly; binary analysis; formal methods

ACM Reference Format:
Freek Verbeek, Nico Naus, and Binoy Ravindran. 2024. Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3658644.3690244

1 Introduction

Decompilation of binaries has been a heavily studied research topic for decades [2, 7, 13, 15, 20, 24, 25, 37]. A typical scenario is lifting a binary to a source-code-like representation, enabling human understanding of the semantics and structure of the binary. This is often necessary for legacy binaries whose sources are not wholly or partially available, but whose security vulnerability analysis is still a necessary task due to the need to rapidly patch program errors, especially those that can be used to create security exploits. Decompilation to the LLVM Intermediate Representation (IR) has been studied as well [40], with the major advantage being that the "distance" between the original binary and the produced IR is less, enabling a more trustworthy decompilation. Mature and industrially developed tools exists, such as IDA-PRO and Ghidra (for reverse engineering) and McSema [11] (enabling application of LLVM level tools on binaries), among others [13, 36, 37].

However, with some notable exceptions, very few decompilation efforts focus on producing an IR that is both *recompilable*, *patchable* and *validatable*. Recompilability implies that the produced IR can be compiled back into a binary. Lifting approaches that target source-code-like representations typically produce non-recompilable and even unsound code. The lifted code has no clear semantics, has holes in it, and is non-executable. Recompilability was never an intended requirement of such decompilation suites: their intention is human-in-the-loop reverse engineering. Patchability means that it is possible to do transformations. This typically requires *symbolization*: instruction addresses as well as the addresses of global variables, external functions and data sections need to be replaced with labels [37, 38]. Finally, validatability ensures that it is possible to check whether the produced IR is a *semantically sound* representation of the original binary.

The motivation behind decompilation with these three properties is multifold. First, it enables a decompile-patch-recompile workflow. Symbolization ensures that at recompile-time instructions and sections can be layed out by the recompiler, a prerequisite for making any modification such as inserting instructions or replacing functions (more details follow in Section 2). Second, it aids in the trustworthiness of the lifted IR. Recompilability allows one to obtain an IR that is executable, and that can therefore be tested. We argue that even if an IR is formally proven to be a correct representation of the original binary, there still is much value in testing. Testing, especially regression testing, shows that the IR truly executes correctly in the original binary's production contexts.

This paper presents a lifter from position-independent (PIE) x86-64 binaries to Netwide Assembler (NASM) [8]. We have choosen NASM as the IR since it is well-known, publicly and open-source maintained, and is recompilable using standard off-the-shelf compilation tools (`nasm` + `gcc`). Moreover, it is an assembly dialect that supports symbolization. With proper options enabled, NASM can be compiled only when it is fully symbolized; the compiler does not

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690244

allow pointers to be immediate values. Most importantly, we argue that it is an IR that is *as close as possible* to the original binary (ensuring validatability), while still enabling patchability. See Figure 1: straightforward disassembly produces an IR extremely close to the original binary, but does not allow modifications. NASM is only one step further: its nearness to the original binary allows validation, but we demonstrate that patching is possible at this level. LLVM IR has more distance from the original binary: it has an unbounded number of variables, function scoping, rudimentary types, and is architecture agnostic. Validating the soundness of lifted LLVM IR with respect to an original binary is an unsolved open problem.

Figure 1: Distances of IRs used in decompilation to the original binary.

We present an approach to validate whether the lifted NASM is *sound*. We define a lifter to be sound, if it produces an IR that can be recompiled into a binary semantically equivalent to the original. We thus validate whether the original binary $\mathcal{B}_0$ and the recompiled binary $\mathcal{B}_r$ run in lockstep fashion (see Figure 2). Such a strong property can only be met due to the nearness of NASM to $\mathcal{B}_0$; a larger decompilation distance typically destroys this property. Notably, this approach is entirely orthogonal to both the NASM lifter as well as the recompiler, i.e., it only considers the two binaries (not the IR) to prove that they run in lockstep fashion. The proofs are formalized in Isabelle/HOL [28] and fully automated.

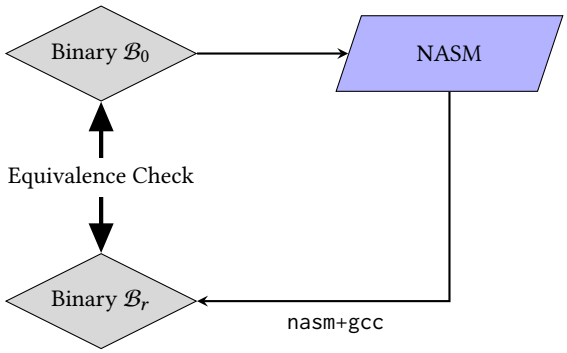

Figure 2: Lifting and recompilation.

Even though the state-of-the-art provides a multitude of tools dedicated to reverse engineering and decompilation of binaries (more discussion follows in Section 7), soundness and recompilability are relatively understudied; they are discussed mostly in the work on McSema [11], Ramblr [37] and FoxDec [35]. Ramblr and its early iteration Uruburos [38] are – to the best of our knowledge – the first to present results on recompilability of symbolized assembly, discussing overhead of running time and memory usage. They execute test cases on their recompiled binaries as well. The key novelty of our work with respect to these prior works, is that our lifted IR comes with a certificate that enables an *automated* and *formal* proof of soundness. In other words, our research contribution is a methodology for lifting binaries to a symbolized IR with a formal proof of soundness.

Section 6 discusses the application of lifting to symbolized NASM on a set of binaries. We discuss how recompiled binaries are tested to behave similar to the original ones. We compare running times and memory usage, and show with experimental results that the overhead is negligible. Moreover, we discuss the manual effort involved in applying a patch to a binary. We discuss verification times, which is roughly four hours for binaries of 20 000 instructions. The scope of binaries is discussed in detail in Section 3; it amounts to PIE x86-64 ELF binaries compiled from valid C code.

As demonstrative case studies, we deploy NASM lifting in the solution to challenge problems set in a DARPA research program. The challenges, discussed further in Section 6, include:

- Modify the binary `wget` so that it only includes HTTPS; any unsecure HTTP should be removed.
- Modify the behavior of OpenSC (a set of software tools and libraries for smart cards) by inserting a check to only load libraries whose file names match a predefined list.
- Achieve a rudimentary form of compartmentalization [16, 26, 32] by restricting the memory writes of a function to its own stack frame.

We have addressed all these challenges *at the binary level*, e.g., we have made modifications to the `wget` binary to produce a new binary `wget_`.

To summarize, this paper contributes:

(1) The first lifting tool for lifting PIE x86-64 ELF binaries to symbolized NASM;
(2) An approach to formal validation of recompiled binaries;
(3) A demonstration of use-cases for binary patching enabled by symbolized NASM lifting;
(4) Experimental results comparing original and recompiled binaries.

Example 1. *Figure 3 presents an example of NASM lifted from an x86-64 binary. A couple of transformations have happened. Some* `rip`*-relative addresses have been transformed into address computations relative to the beginning of their section. For example, the address computed by the second instruction belongs to the* `.data` *section of the binary. Second, a label* `L1` *has been introduced since it is a possible jump target. The code loads a function pointer into register* `rax` *twice: once an internal function symbolized to label* `L2`*, and once an external (dynamically linked using the NASM keyword* `wrt ..plt`*) function* `puts`*. It calls either of these functions indirectly, based on the initial value of register* `edi`*.*

All implementations and formal proofs have been made publicly available:

https://doi.org/10.5281/zenodo.12721325

The next section discusses use cases for lifting binaries to symbolized NASM. Section 3 discusses assumptions, scope and limitations. Section 4 then provides an overview of how symbolization is achieved. Section 5 presents validation, discusses the soundness definition, as well as threats to validity. In Section 6, we discuss results and applications. We describe an example of a lift-patch-recompile

```
1068:  mov   eax,QWORD PTR [rip+0x2fa2]
106e:  mov   QWORD PTR [rip+0x2fbc],eax
1074:  lea   rax,[rip+0x115]
107b:  cmp   edi,0x3
107e:  jle   [rip+0x7]
1080:  mov   rax,QWORD PTR [rip+0x2f51]
1087:  lea   rdi,[rip+0xf7f]
108e:  call  rax
```

(a) x86-64 Assembly

```
    mov rax, qword [stderr]
    mov qword [.bss + 0x10], rax
    lea rax, [L2]
    cmp edi, 0x3
    jle L1
    lea rax, [puts wrt ..plt]
L1: lea rdi, [.rodata + 0xd]
    call rax ; Resolved: [puts,L2]
```

(b) Symbolized NASM

Figure 3: Assembly code and the lifted NASM.

workflow step-by-step. Related work is discussed in Section 7 before we conclude with discussion in Section 8.

2 Use Cases of Lifting to Symbolized NASM

We discuss a couple of examples of use cases enabled by lifting to recompilable and symbolized NASM. Each of these use cases has been performed, on binaries in the order of complexity of CoreUtils such as `tar`, `wget` and `wc`.

Binary *patching* is achieved by lifting a binary, making a semantical modification, and then recompiling it. Examples are:

- Instruction or function *insertion/removal*: without symbolization, the layout of instructions is fixed. One can replace a single instruction with another instruction of the same byte-length, and even achieve a form of trampolining that way [12], but other modifications are hard. Removal of dead code is feasible using static analysis [30], but simply adding a new text- or data section is hard or impossible. Symbolization ensures that the recompiler can redo the entire layout. Therefore, it is easy to insert functions, intersperse instructions between existing ones, remove dead code, or add new data sections that were not in the original one.
- Function *updates*: for the same reason as above, one can update functions as well. One can replace functions that are deprecated with an error message or a reimplementation. Reimplementation can be done at the source code level. If one wants to replace a function `f` with a new implementation, then function `f` can be removed from the binary and declared as external. Then, one can write a new C function `f`, compile it to an object file, and add that object file to the linker during recompilation.

Binary *hardening* can be achieved by de-then-recompiling a binary, without doing semantical modifications. Examples are:

- The x86 `endbr64` instruction [34] is a security mechanism introduced by Intel that offers hardware protection against Return-oriented Programming (ROP, [31]) and Jump/Call-oriented Programming (JOP/COP, [4, 6]) attacks. Simply put, every function entry should start with an `endbr64` instruction, enabling Intel's CET technology to implement a shadow stack that offers protection against such vulnerabilities. By lifting to a symbolized IR, one can automatically insert `endbr64` instructions and thereby harden the binary.
- When a binary makes use of dynamic linking (as is typical in binaries), the linker writes – at linking time – addresses of external functions into relocation entries. In older versions of ELF files, the section containing these relocation entries are found in a writable data section (as the linker needs to write). This is a vulnerability, as a pointer to executable code is in a writable segment. Newer ELF files contain a `.data.rel.ro` section that is read-only after dynamic relocations have been applied. We harden old versions of binaries so that they make use of `.data.rel.ro` sections.
- Binary-level debugging information is extremely useful for analyzing low-level crashes or assertion violations. A stripped binary comes with no debugging information. By recompiling with `gcc`'s debugging option `-g` we can insert debugging information, allowing to easily find out the exact path leading to a crash where previously that was hard.

3 Discussion on Assumptions

The main challenge in symbolization of assembly is identifying which values are pointers, assessing where they point to, and use that information to decide how to symbolize. In general, this requires binary-level pointer analysis, a notoriously hard problem to solve accurately [14]. The fundamental problem lies in classifying immediate values: if the value falls within the range of possible addresses of instructions or data, then it is undecidable whether that value constitutes a pointer or not.

That problem, however, does *not* present itself in the case of PIE executables. Pointer computations can be arbitrarily complex. However, we argue that for PIE executables, the computation of the *base* of a pointer is limited to a set of specific cases. Each of these cases can be symbolized, and as such each pointer can be symbolized. We have identified the cases in Figure 4.

A pointer base is introduced when reading in `rip`, returning some statically known immediate value i. It may be introduced by reading the stack pointer (typically register `rsp`) or the Thread-Local-Storage (TLS) pointer (typically segment register `fs`). It can be introduced by an external function, or it can be introduced by having been passed to the current function as parameter through some register or somewhere in memory (p denotes some part of the initial state, be it a register or memory). A pointer base can also be introduced by reading from a part of the memory that contains an immediate value i that has been written there during runtime

Base	≡	IP i	Instruction Pointer	`lea rax, [rip+0x15]`
	|	SP	Stack Pointer	`mov qword ptr [rsp - 0x30], rax`
	|	TLS	Thread-Local-Storage Pointer	`mov rax,qword ptr fs:0x28`
	|	EX f	Generated by external function	`call malloc`
	|	IN p	Initial value	`mov qword ptr [rdi], rax`
	|	REL i	Relocation	`dq 00 40 00 00 00 00 00 00`
	|	RET	Return address	`call/ret`
	|	TBL i	Jump table entry	

Figure 4: Pointer Bases in PIE binaries, with examples of instructions that introduce pointers with such bases.

linking by the OS (a relocation). The return address is a pointer as well. Finally, pointers can be introduced by a series of instructions that combine non-writable data from a jump table with the current `rip`. Such a jump table does not necessarily contain immediate pointer values itself, but data that combined with a certain `rip` value through some instructions produce an immediate value i. Different compilers can have different implementations.

Assumption 1. *Every pointer in the binary is computed with as positive addend a* Base*, i.e., its computation has the form* $b \pm a_0 \pm a_1 \pm \ldots$ *for some* Base b *and some possibly empty set of addends* $a_0, a_1, \ldots$.

Note that this assumption does not hold for non-PIE executables. There, an immediate address may be used directly, without containing a base such as above.

Lifting, symbolization and then recompilation produces a binary in which pointers will have different values. Soundness of this process can only be assured when the actual values of pointers at runtime do not influence the behavior of the binary. For example, the following source code snippet has behavior depending on the *value* of a pointer:

```
int global;
void main () {
  printf("%p\n", &global);
}
```

This program may print a different value after symbolization and recompilation, since the data section may have been moved. We cannot prove that the observable behavior of the binary obtained by compiling this code will be preserved after lifting, symbolization and then recompilation, since the printed value may be different.

Assumption 2. *The observable behavior of the binary is independent of the values of pointers.*

The crucial consequence of these assumptions is that *if all base-introductions are symbolized*, then *the binary is symbolized*. The intuition is that the only addend of a pointer that must be symbolized is its base. All other computations to it, and any behavior that ensues the base-introduction, remains unmodified. This explains how symbolization can be done even in the presence of unresolved indirections. Consider an instruction `jmp rax`. Register `rax` holds a pointer. Even if statically it cannot be established where this pointer may point to at runtime, as long as the instructions that compute the base of the pointer have been symbolized, the instruction will jump to the right address after symbolization.

Assumption 3. *The set of reachable instructions can be overapproximated given a manually supplied set of function entries.*

Any decompilation effort assumes that all reachable instructions have somehow been disassembled. That, however, is an undecidable problem, due to indirections and mixing of data and instructions. Our approach assumes a list of function entries, from which the binary is explored through recursive traversal. The next section discusses how such a list can be obtained interactively. If the list is incomplete, then this will be detected at recompile-time. Note that the reachability problem heavily exacerbates when dealing with C++ binaries due to exceptional control flow. For that reason, we consider those binaries out of scope.

To intuitively summarize, our approach works for x86-64 PIE binaries in the ELF format that have been obtained by compiling code that adheres to the C standard.

4 Symbolization

Prerequisites. A *CFG* is a graph with as vertices basic blocks, i.e., lists of disassembled instructions. Let $\mathbb{W}_{64}$ denote 64-bit words. We assume the existence of a function Δ of type $\mathbb{W}_{64} \mapsto \{\text{CFG}\}$. This function takes as input an *entry address* (be it the entry address of the binary, or the entry address of some function within the binary), and produces a set of CFGs. It recursively traverses the binary, building a CFG. If an internal function call is encountered, this may produce a separate CFG. We explicitly acknowledge that function Δ is typically *incomplete*, i.e., executing this function even on the main entry point of the binary will not produce all functions, nor cover all instructions. It suffices if it covers as much of the binary as possible. We have used the disassembler of Verbeek et al. [35] as it comes with a formal proof of soundness of the produced CFGs (modulo unresolved indirections). Moreover, we assume the existence of a set of entry addresses ε of type $\{\mathbb{W}_{64}\}$.

Relocations are a mechanism to achieve PIE binaries. A binary (even when stripped) contains a relocation table with entries of the form $a_0 \rightarrowtail a_1$, with a_0 and a_1 of type $\mathbb{W}_{64}$. Here a_0 is an address somewhere in a `data` section of the binary (typically a `.bss` section). This means that during runtime linking, the OS will write the relative value of a_1 into memory at relative address a_0. In other words, at runtime, $*a_0 == a_1$.

The main algorithm of symbolization is to apply function Δ to all entry addresses in ε, traverse all produced CFGs, and symbolize each instruction. Whenever an instruction introduces a pointer base, it needs to be modified only in the three cases where the base

contains some immediate i. We first describe how immediates are symbolized, and then how jump tables are treated.

Symbolization of Immediates. Let i of type $\mathbb{W}_{64}$ be an immediate value occurring in some Base b. The following steps can be undertaken, with examples relative to Figure 3.

- Value i is the address of some instruction within the currently known set of CFGs. In this case, that instruction is labeled with fresh label `L` and value i is replaced with label `L`. Example: line `107e`.
- According the binary's symbol table, value i is a pointer to memory storing the address of an external function `f`. In this case, an operand `qword ptr[i]` is replaced with `f wrt ..plt`. This will ensure that external functions can be loaded dynamically in the recompiled binary using a procedure linkage table. Example: line `1080`.
- According to the symbol table, value i is a pointer to an external object `O`. In this case, value i is replaced with label `O`. Example: line `1068`.
- The binary's relocation table contains a relocation of the form $i \rightarrowtail a_1$. A fresh label `L` is inserted into the `data` section and value i is replaced with label `L`. Note that a_1 is itself an immediate pointer and is symbolized according to these rules as well.
- Value i is within the range of addresses encompassed by a `data` section (be it `.rodata`, `.bss`, ...). A fresh label `L` is inserted into the `data` section and value i is replaced with label `L`. Example: line `106e`.

It may be the case that none of these cases apply. In Figure 3, if the code snippet is symbolized as-is, then line `1074` will produce a *dangling pointer*. In practice, this indicates that set ε is incomplete (without exception, this was the case for any dangling pointer found during symbolization of all applications). We thus add the dangling pointer to ε and rerun symbolization, after which the dangling pointer has been eliminated. This may produce new dangling pointers, but without exception we have not needed to repeat more than twice in any of the applications.

Symbolization of Jump Tables. We explain the treatment of jump tables by example (see Figure 5). We would like to stress that compilers may implement jump tables in different ways, and that our approach is independent of compiler-specific implementations. The code first loads the address of the jump table into register `rbp`. Then, at some point, it checks whether some index (in `rcx`) is bounded by 8. It then reads a 4-byte jump table entry. The read data is not yet the pointer: that is obtained by adding the address of the jump table itself to the read data.

Our approach traverses the binary backwards, starting at the unresolved indirection at line `d18e`. It tries to find a conditional jump that bounds an index (line `d182`). If found, then it initializes symbolic states that store no information other than that the index has a concrete value. In the example, it would generate eight states with each a value for register `ecx`, starting at line `d184`. During symbolic execution, it then encounters a register with an unknown value (`rbp`). It again traverses backwards to see if it can concretize that register as well. Ultimately, if this process can ensure that *all* registers used in the computation of the indirection are concrete,

```
d102: lea    rbp,[rip+0x5fa87]
...
d17f: cmp    ecx,0x8
d182: ja     d1fd
d184: mov    edx,ecx
d186: movsxd rdx,DWORD PTR [rbp+rdx*4]
d18b: add    rdx,rbp
d18e: jmp    rdx
```

Figure 5: Example Implementation of Jump Table

then the indirection can be computed. This produces an observation that:

```
@d184 :  ecx  0     1     2     3     4     5     6     7
@d18e :  rdx  d198  d210  d230  d198  d2c8  d310  d340  d360
```

Symbolization now inserts an instruction at line `d184`, that moves the value of `ecx` to memory. In the produced IR, we add a new 8 byte data section to the binary to that end. The following instructions are then executed as normally. At line `d18e`, register `rdx` is then discarded, and the jump is replaced with instructions that based on the stored value in that special data section jump to the right address. The reason that we still execute all instructions up to the jump is because it may be the case that they modify state other than computing the jump. The inserted instructions that replace the jump are such that they do *not* modify any part of the state – not even flags – other than register `rdx` and `rip`.

5 Verification

As Figure 2 shows, our approach produces a recompiled binary $\mathcal{B}_r$ from an original binary $\mathcal{B}_0$. This section discusses how we verify an equivalence relation between both binaries. Besides formal verification, we also apply testing to both binaries, as they are both executable. Section 6 discusses the testing setup and results.

The binaries $\mathcal{B}_0$ and $\mathcal{B}_r$ are similar, but not equivalent. Instructions as well as data sections have been transposed. A *transposition* Γ maps addresses from $\mathcal{B}_0$ to addresses in $\mathcal{B}_r$. We thus aim to formalize that binaries $\mathcal{B}_0$ and $\mathcal{B}_r$ are semantically equivalent "modulo transposition". Ideally, one would formalize a relation over states that returns true if and only if one state is the transposition of the other. Such relation could be used to show a bisimulation [1] between the two binaries. This approach, however, is infeasible as it is impossible to know if, e.g., a value stored in a register is a pointer that needs to be transposed or a regular value. We thus formalize the desired equivalence differently, i.e., by transposing *the semantics*, not the state. For example, the semantics will contain a function that reads from a memory address. The transposed semantics will first transpose that memory address before executing the read.

Let Σ denote the type of binary-states: states with registers, flags and unstructured memory. Let the semantics of executing a single instruction i be defined by a transition relation $\longmapsto^i$ of type $\Sigma \times \Sigma \mapsto \mathbb{B}$. The semantics of a binary $\mathcal{B}$ can be defined inductively, by identifying a set of valid initial states (e.g, the instruction pointer is set to the entry-point of the binary and the sections are loaded into memory), and the per-instruction transition relation. This

produces transition relation $\longmapsto^{\mathcal{B}}$ for binary $\mathcal{B}$. Given transposition Γ, notation $\longmapsto_\Gamma$ will denote the transposed semantics. One could then formalize equivalence as:

$$\longmapsto_\Gamma^{\mathcal{B}_0} = \longmapsto^{\mathcal{B}_r}$$

In words, the transition relation of the recompiled binary is *exactly equal to* the transposed transition relation of the original binary.

However, a second complication arises due to the fact that sometimes instructions have been inserted, e.g., to symbolize jump tables. These instructions are choosen in such a way that the net effect of each basic block is the same, even when in the middle of a basic block different instructions are executed. We therefore define binary-equivalence by requiring that the two control flow graphs are isomorphic (denoted $\doteq$), and that all isomorphic basic blocks have semantics that are exactly equal. A basic block is a sequential list of instructions, and thus we define the semantics of a basic block $[i_0, i_1, \ldots]$ as follows:

$$s_0 \longmapsto^{[i_0,i_1,\ldots]} s' \equiv \exists s_1, s_2, \ldots \cdot s_0 \longmapsto^{i_0} s_1 \longmapsto^{i_1} \ldots s'$$

Definition 1. *Let Γ be a transposition. Let g_0 and g_r be the CFGs, and let β_0 and β_r be the sets of basic blocks of $\mathcal{B}_0$ and $\mathcal{B}_r$. Binaries $\mathcal{B}_0$ and B_r are* Γ-similar*, notation $\overset{\Gamma}{\simeq}$, if and only if:*

$$\mathcal{B}_0 \overset{\Gamma}{\simeq} \mathcal{B}_r \equiv \begin{cases} g_0 \doteq g_r \\ \forall b_0 \in \beta_0, b_r \in \beta_r \cdot b_0 \doteq b_r \implies \longmapsto_\Gamma^{b_0} = \longmapsto^{b_r} \end{cases}$$

In words, the CFGs are isomorphic and for each isomorphic pair of basic blocks (b_0, b_r) the transposed semantics of basic block b_0 are exactly equal to the semantics of basic block b_r.

5.1 Transposed Semantics

A crucial element in defining instruction semantics is memory accesses. These can be defined using generic `Load` and `Store` operations (see Figure 6). A memory region is denoted $[a, si]$ storing an address a and a size (in bytes) si. Consider the instruction `Load` r_d $[a, si]$. If a is an immediate address, that address is transposed. If it is a register, then the address is read from that register but not transposed, since it can only have been written to that register after transposition to begin with.

$$\frac{a = \Gamma(imm) \qquad v = \mathsf{read}(a, si, s)}{s \longmapsto_\Gamma^i s(\mathsf{regs}(r_d) \coloneqq v)}\ i = \texttt{Load}\ r_d\ [imm, si]$$

$$\frac{a = s.\mathsf{regs}(r_s) \qquad v = \mathsf{read}(a, si, s)}{s \longmapsto_\Gamma^i s(\mathsf{regs}(r_d) \coloneqq v)}\ i = \texttt{Load}\ r_d\ [r_s, si]$$

$$\frac{a = \Gamma(imm) \qquad v = s.\mathsf{regs}(r_s)}{s \longmapsto_\Gamma^i \mathsf{write}(a, si, v, s)}\ i = \texttt{Store}\ [imm, si]\ r_s$$

$$\frac{a = s.\mathsf{regs}(r_d) \qquad v = s.\mathsf{regs}(r_s)}{s \longmapsto_\Gamma^i \mathsf{write}(a, si, v, s)}\ i = \texttt{Store}\ [r_d, si]\ r_s$$

Figure 6: `Load` and `Store` semantics, based on functions `read` and `write` that perform memory access in little-endian fashion.

Other than memory accesses, a crucial element of instruction semantics is in the operations executed by the instructions. The different types of operations are numerous, ranging from floating-point operations, to cryptographic operations, to byte-level swaps, arithmetic, logic, etc. However, note that we do not need a formal semantics for all these operations. All we need to prove, is that the *same operations* are applied to the *same values*. Basic operations, such as arithmetic, logic and flags are given a formal semantics, but the majority of operations are modeled through uninterpreted functions.

Combined, this produces a transition relation $\longmapsto_\Gamma^i$. The untransposed transition relation $\longmapsto^i$ can be derived by taking as Γ the identity function.

5.2 Proving Semantical Equivalence

Deciding Γ-similarity requires three ingredients:

(1) Recovering control flow graphs;
(2) Performing symbolic execution over basic blocks;
(3) Proving equivalence between two symbolic states.

Figure 7 provides an overview of the verification approach. Starting with binary $\mathcal{B}_0$, the binary is lifted to NASM, and recompiled to binary $\mathcal{B}_r$. Both binaries are then lifted to *low P code* using the Ghidra decompiler. Low P code is an IR specifically developed for Ghidra. The use of Ghidra provides two advantages:

- Low P code reduces the complexity of the CISC x86-64 instruction set to a RISC-style instruction set of only 64 instructions. It is therefore relatively easy to formalize the language in a formal environment and to build a symbolic execution engine for it.
- We argue that the verification effort should be orthogonal to the lifter that created the recompiled binary. For example, if during verification we reconstruct CFGs with the same tool as used during NASM creation, then a bug in that CFG-reconstruction may be missed even if the verification effort succeeds. We thus use Ghidra during verification, a state-of-the-art externally developed binary analysis framework, instead of reusing internally developed tools.

During lifting, a mapping γ_0 is produced that is used to construct the mapping Γ needed to prove Γ-similarity. The lifter does not have any information of binary $\mathcal{B}_r$ at its disposal, as it does not yet exist. The only information it can produce is mapping γ_0 that maps addresses of $\mathcal{B}_0$ (thus: addresses from within the original binary) to *labels* in binary $\mathcal{B}_r$. Only after recompilation it can be seen what addresses these labels will correspond to. Thus, after compilation a mapping γ_r is produced from labels to addresses in $\mathcal{B}_r$. Mapping Γ is then obtained by functional composition:

$$\Gamma \equiv \gamma_r \circ \gamma_0$$

The low P code combined with mapping Γ produce a *certificate*. The certificate is a set of theories formulated in Isabelle/HOL [10, 17, 28], such that if all lemma's in the certificate can be proven, then binaries $\mathcal{B}_0$ and $\mathcal{B}_r$ are Γ-similar. To this end, we use Ghidra to construct two CFGs. These include all basic blocks. Per isomorphic pair of basic blocks (b_0, b_r) , we generate a lemma that formulates:

$$\longmapsto_\Gamma^{b_0} = \longmapsto^{b_r}$$

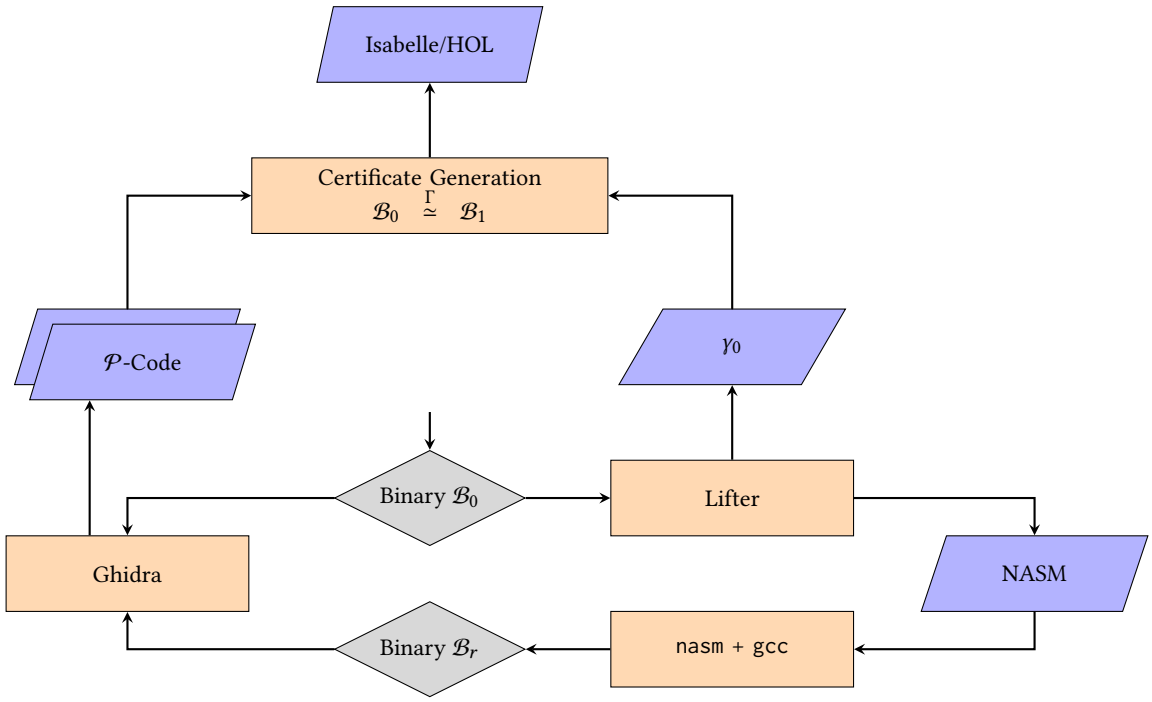

Figure 7: Verification Flowchart

The certificate also contains a proof tactic to automate the proofs. This tactic have been implemented using the Eisbach language for defining proof methods [22]. The tactic first symbolically executes both basic blocks, producing two symbolic states. These symbolic states are expressed as a series of state-updates. In order to prove their equivalence, the tactic will consider these state changes ony-by-one, and prove that they occur to the same state-part, and with the same symbolic value.

5.3 Threats to Validity

Even if two binaries are formally proven to be Γ-similar, there may be issues preventing them from exhibiting the exact same behavior when executed. We identify and discuss three possible causes, and their mitigations.

Alignment may cause execution to differ even if the exact same instructions are executed on the exact same data. There exists x86 instructions that behave differently depending on how they memory region they access is aligned. An example is the `MOVDQA` instruction which can causes a segmentation fault when its region is not properly aligned. Consider two hypothetical binaries both consisting of the same single `MOVDQA` instruction. These are Γ-similar, however, one may crash where the other may succeed. We mitigate this issue by transferring the alignments from the original binary to NASM, so that the recompiled binary has the same alignments as the original one.

Bugs in Ghidra may enable a scenario in which Γ-similarity is proven for two semantically different binaries. Consider, hypothetically, a scenario in which Ghidra does not produce complete CFGs, i.e., it omits basic blocks that are reachable during execution. This scenario may realistically happen, e.g., if code is only reachable through some indirection. In such a scenario, it may happen that Γ-similarity is proven only over a subset of the reachable part of the binaries. We mitigate this issue by supplying Ghidra with information on which functions have been de- and recompiled, which sometimes will aid Ghidra in discovering reachable instructions. However, other issues in CFG reconstruction may occur. For example, Ghidra may not correctly resolve an indirection. We argue this issue cannot ever be mitigated in full: any disassembler, be it Ghidra, IDA-PRO, Radare2, or angr, may suffer from this issue.

External functions may behave differently after recompilation. We prove Γ-similarity only over the instructions within the binaries. External functions, i.e., functions loaded at linking time, are not considered. It is proven that the recompiled binary calls external functions *with the same name*. For example, it is proven that both binaries will execute an instruction `call printf`. However, the recompiler decides how to link external symbols to their implementations. It may be the case that during recompilation, one needs to manually supply the libraries with which to link. The same holds for external objects such as `stdout` or `__progname`.

The above issue can be seen as a specific case of a more generic threat to validity. The task of the recompiler mostly consists of linking different object files and producing a PIE executable (e.g., implementing a PLT section so that external functions are executed

properly). Various other tasks commonly executed during compilation (e.g., optimizations or insertion of security mechanisms such as stack protectors) are not performed as the exact instructions to be executed are given and fixed. In general, if the recompiler performs its task differently from the original compiler, the two binaries may behave differently. A specific example is that if the recompiler links to different external functions than the original compiler, this can cause an issue even if the binaries are formally proven to be Γ-equivalent.

6 Experimental Results and Applications

Table 1 provides an overview of binaries that have been lifted and recompiled. We have lifted binaries from the Parsec benchmark suite [3], various CoreUtils binaries, the FDLIBM library (software implementations of various mathematical functions), openSSL, and OpenSC. The binaries were compiled through their own build mechanisms, i.e., we did not control which compiler was used or which compiler settings such as optimization level.

#Instructions	Name	Part of	Success
771	blackscholes	Parsec	
36 181	ferret	Parsec	
1810	vips	Parsec	
891	vips.mergup	Parsec	
787	vips.find_mosaic	Parsec	
30 970	du	CoreUtils	
14 127	gzip	CoreUtils	
3043	hexdump	CoreUtils	
9915	sha512sum	CoreUtils	
18 309	sort	CoreUtils	
6346	tar	CoreUtils	
6272	wc	CoreUtils	
81 768	wget	CoreUtils	
14 317	fdlibm	FDLIBM	
113 551	ssh	OpenSSL	No
12 188	OpenSC	OpenSC	
351 246			

Table 1: Lifted and Recompiled Binaries

We discuss the following questions relative to this table:

Q1. *Can it be confirmed through testing that the recompiled binary behaves the same as the original?*

Q2. *Does recompilation incur overhead in terms of running time or memory usage?*

Q3. *Did verification/testing expose issues?*

Q4. *What manual effort is involved in executing a patch?*

Q5. *How long does verification take?*

Q1 & Q2 (testing and observing the recompiled binary):

We have tested for *observable behavior* as well as *trace equivalence*. Testing for observable behavior consists of running a binary and observing whether the output is as expected. Testing for trace equivalence consists of running a binary and logging a trace of which instructions are executed at runtime, as well as the data read by the instructions.

Observable behavior: The CoreUtils library comes with a professionally maintained manually developed test-suite tailored to the binaries at hand. This test suite has a large degree of coverage of both normal as well as unexpected behavior. For all recompiled CoreUtils binaries, we have replaced the original binaries with the recompiled ones and run the test suite as-is. All test-cases are successful. With respect to Parsec, we have run the programs on the inputs provided by the benchmarks and confirmed equal output. For the FDLIBM library, we have replicated the testing set up of Verbeek et al. [36] which generates a large number of random floating-point numbers in various ways (e.g., random bitstrings, or sampling from likely values). Each mathematical function (e.g., `sin` or `log`) is applied to all random numbers and the output is logged for comparison to the original. No differences have been found.

Trace equivalence: With `gdb` we have logged traces. A trace consists of a list of trace elements, where each trace element contains 1.) which instruction is executed, and 2.) the data stored in the source operands before execution of the instruction. Since that data may consist of pointer values, the traces will differ. We therefore print data only if it does not "look like" a pointer. At runtime, pointers typically have large values. By printing data only if its integer interpretation is less than a certain cap (e.g., 100 000), one obtains two traces that are *textually equivalent*. One can use a textual `diff` to compare the traces. The only differences found are in `nop` operations, and instructions due to reimplementations of indirections. Of course, theoretically the use of such a cap is unsound: there may be non-pointer values larger then the cap. However, we argue that this approach provides substantial increase in trustworthiness in addition to both verification and testing for observable behavior.

In all cases, no detectable difference in either running time or memory usage has been encountered. The Parsec benchmark suite specifically is intended for benchmarking running times. This result is as expected: the recompiled binary executes virtually the same instructions as the original. Only in case of indirections, computations on how these indirections are computed may differ. The difference is at most four of five instructions per indirection, which has negligible impact.

Q3 (issues during symbolization):

Testing has exposed an issue related to the standard C library function `error`. Initially, it was assumed that this function always terminates. That is not the case, it may terminate based on the value of the first parameter. In cases where the function was used without immediate termination, we could observe difference in the exit value of the binary. The issue has been resolved.

All recompiled binaries executed correctly, with as notable exception the `ssh` binary. Both the observable behavior as well as traces diverge for some executions. We have been able to run the recompiled binary successfully, however, in some cases where an error message is to be produced by the binary, behavior diverges, We have traced the issue to a specific part of a .data section that initially contains just the value 0, but is at linking time initialized with a value by the OS in the original binary, whereas the same does not happen for the recompiled binary. We suspect this is due to specific compilers flags used during compilation of the original

binary.

Q4 (applying a patch):

We here describe the manual steps involved in a realistic semantical patch; a challenge proposed in the context of a DARPA research program. The challenge is to modify the binary `wget` so that it only includes HTTPS and disallows HTTP. This required several patches: we have removed a command-line option, and introduced checks whether an opened connection is HTTP or not. We here describe the simplest patch for sake of explanation.

The first step is lifting, where the only manual effort consists of dealing with dangling pointers (see Section 4). The next step is to find *hooks*, i.e., the points where it makes sense to insert a new function. Through manual analysis, we have identified a point just before a function return where a check can be inserted whether the connection is HTTP. Inserting a function call just before the `pop`s before a `ret` is convenient, as there we assume that the local stack frame can be discarded and callee-saved registers may be used. This even allows calling functions, without destroying any part of the state that is not allowed to be destroyed according to the calling convention. We identify that the value currently stored in `rax` provides the connection type as an integer. That value is thus stored, and passed to the patch as first parameter in register `rdi` (see Figure 8a). The patch is called, and register `rax` is restored, before continuing execution as normal.

Within NASM, function `myPatch` is declared as external. The NASM can be compiled to an object file `wget_.o`. Function `myPatch` can be written in C (see Figure 8b) and compiled to another object file `myPatch.o`.

Recompilation then is performed by the following command:

```
gcc  -g -m64 -nostartfiles
     -o wget_
     wget_.o myPatch.o
     -lgnutls -lidn2 -lnettle
```

The options on the first line are fixed for recompiling NASM. They instruct `gcc` to not use the standard system startup files when linking, as all of this machinery is already present in the NASM file. The second line names the executable. The third line tells `gcc` to link both object files.

The main manual effort is then in identifying which libraries need to be linked to resolve all external functions. If one omits a library, recompiling will not succeed. For the `wget` case study, we manually select three libraries that need to be installed before recompilation succeeds.

As second example, we sketch the contours of an approach to achieve compartmentalization. The challenge is to ensure that a given function can only write to a given part of the memory (in this example, it should only write to is own stack frame). The function belongs to a server that expects data from a client and writes it to a buffer. Our approach is to first identify which memory writes occur where to a base is added some dynamically computed addend. An example of such a memory write is an instruction with operand `rbp + rdx - 0x60`. The base is register `rbp` which stores the frame pointer, but an unknown value `rdx` is added. Since the base is the frame pointer, all that needs to happen is to check that the address does not point to above the current stack frame (see Figure 9). This check is achieved by interspersing instructions before the memory write happens. The patch requires inserting instructions within the original function, as well as adding a new part labelled `overflow_detect`.

The memory write originally actually was unsafe. It originated from the following line of C code:

```
while ((buff[n++] = getchar()) != '\n');
```

Here, `buff` is a local array. The inserted check jumps to label `overflow_detect` as soon as the above code would overwrite the return address.

The above pattern can be generically applied to any memory operand. It is, however, expensive in terms of running time. Mitigations are possible. First, in this specific example the check can be achieved more efficiently by simply comparing register `rdx` to value `0x60`. That does not require the use of a temporary register. It may be the case that the flags are not read after the original move, so that storing/restoring the flags is unnecessary as well.

In general, one wants to 1.) do a pointer analysis which overapproximates the set of writes that are to be patched (e.g., writes with an unknown base, or with an unknown addend), 2.) find an efficient implementation for checking whether the write occurs within a pre-specified region of memory, and 3.) intersperse that implementation into the binary. We have done these steps manually for a small example: more automation would be interesting for future research.

Q5 (verification times):

Figure 10 shows verification times. All results have been obtained on an M1 Pro machine with 32GB of memory. We would like to stress that we make heavy use of parallelization. Within the verification process, Isabelle/HOL is capable of running all generated theory files in parallel [39]. Each function corresponds to a theory file, so all functions can be verified in parallel. This is necessary: we run into memory issues for the larger examples if we load and verify the entire certificate at once. We thus split up the certificate into chunks and to prove each chunk in a separate process.

7 Related work

Decompilation has been a subject of research that has been around almost as long as compilers have. The work done by Cifuentes [7] forms the basis of most modern decompilers. She describes a decompilation pipeline that decouples the input architecture and the output language from a universal decompilation module, that performs operations like dataflow, type and control flow analysis. Most modern decompilation and reverse engineering tools have adopted this architecture.

A key observation is that generally decompilation is focused on providing *human understanding* of the binary, rather than *machine understanding*. Tools such as IDA-PRO[1], Ghidra[2], Binary Ninja[3],

[1] https://hex-rays.com/ida-pro/
[2] https://ghidra-sre.org/
[3] https://binary.ninja

```
mov   eax 0x4,
cmove eax, ebx
;; BEGIN MANUALLY INSERTED
push  rax
mov   rdi,rax
call  myPatch
pop   rax
;; END MANUALLY INSERTED
pop   rbx
pop   rbp
...
ret
```

(a) Patched Assembly

```
void myPatch (int conn_type) {
  if (conn_type == SCHEME_HTTP) {
    puts ("PATCH: exiting because of" \\
          "insecure HTTP connection.");
    exit(1);
  }
}
```

(b) Source of Patch

Figure 8: Inserting a patch.

```
; BEGIN MANUALLY INSERTED
pushfq                          ; Store flags
push rax                        ; Store a temp register
lea rax, [rbp + rdx - 0x60]     ; Load value of operand into temp register
cmp rax, rbp                    ; Compare operand to frame pointer
jae overflow_detect             ; If above or equal, then error
pop rax                         ; Restore temp register
popfq                           ; Restore flags
; END MANUALLY INSERTED
mov byte [rbp + rdx - 0x60], CL ; The original instruction
```

Figure 9: Patch: restricting a memory write

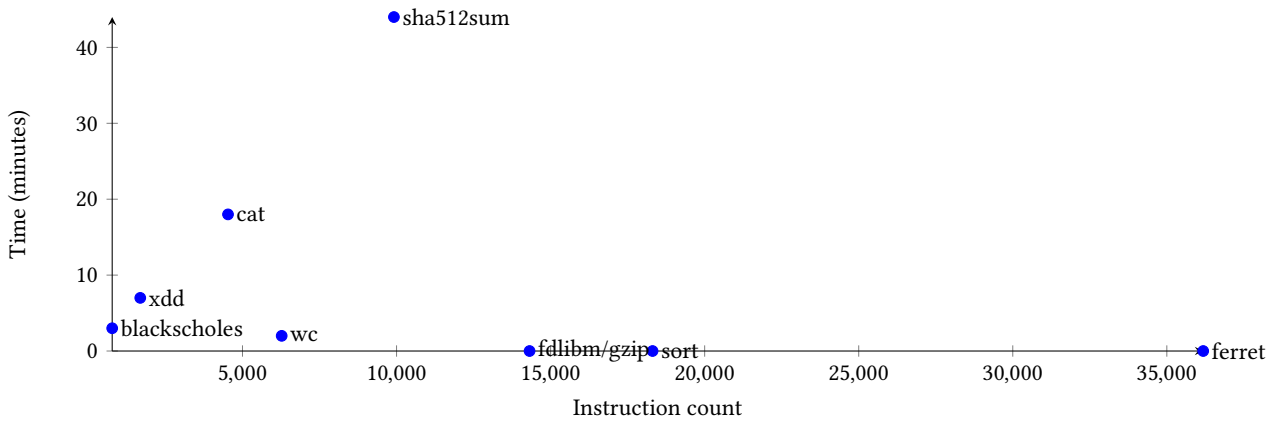

Figure 10: Verification Times

Radare2[4], RetDec [20], and SmartDec [13] each are focused on producing a "source-like" representation of the original binary. Various undecidable problems cause these decompilers to apply heuristics, best-effort guesses or domain-specific knowledge during the lifting process, or to introduce holes (undefined or unspecified behaviors). These problems include variable recovery, type inference, control flow recovery, and function-interface recovery. The result is generally code that is as humanly readable as possible, but not machine-readable. The purpose of these tools is human-in-the-loop reverse engineering and binary analysis.

Recompilable IRs. Machine-readable means in this context that the code at least is compilable. McSema translates x86-64 binaries to the LLVM IR, and was arguably the first to produce a recompilable higher-level IR. IDA-PRO is used for CFG generation (or other tools can be plugged in), and from there individual instructions of the

[4]https://rada.re/n/

Tools	Target IR	Recompilability	Patchability	Validatable
IDA-PRO Binary Ninja Radare2 …	Human-readable source			
McSema	LLVM IR	✓		
Ramblr	Assembly	✓	✓	
FoxDec	C	✓	✓	
This work	NASM	✓	✓	✓

Table 2: Inexhaustive Overview of Related Disassemblers and Decompilers

basic blocks are translated to LLVM. Effectively, McSema implements a *simulation* of the original binary. It provides a datastructure called `State` which simulates registers, a stack, and heap memory. For example, a simple arithmetic instruction is translated to calling an appropriate function that manipulates the current `State` object. Functions need wrappers that map a McSema context to the native context and conversely. We argue that the overhead induced by this approach is large. In addition, this approach makes it hard to construct an argument for soundness, which is not mentioned as a desired characteristic to begin with. McSema does not do any form of symbolization. For example, when recompiling, one has to fix sections to specific addresses.

Patchable IRs. Symbolized assembly was studied by Wang et al. and implemented in Ramblr [37]. Ramblr focuses on reassemblable assembly code, first put forward in [38]. The fact that symbolization enables patchability is posited there. Wang et al. are, to the best of our knowledge, the first to apply testing to recompiled binaries, and to report on numbers such as running time overhead (similar to this paper, the overhead is negligible). Similar to this paper, they confirm through testing that binaries have the same observational behavior. Ramblr essentially is based on *content classification*, i.e., differentiating bytes in the binary as either code or data. Moreover, it executes an intra-function data dependence analysis to determine whether values are used as a pointer later on. Both these problems are undecidable, which they explicitly admit: it is "an empirical solution that works on many binaries whose integer distributions roughly follow the pattern as presented". We stress here that even though this paper does not require a data dependence analysis such as Ramblr, our approach does not escape the infeasibility of content classification either (see Assumption 3 in Section 3 and the treatment of dangling pointers). For both approaches, instructions in the binary may be missed if they are erroneously considered unreachable. Ramblr makes assumptions similar to the assumptions discussed in Section 3. The key difference between our approach and Ramblr is the production of a certificate that allows a formal proof of correctness. Moreover, Ramblr does not scale, as also confirmed by the authors of BinPointer [19] who could only successfully apply it to micro-benchmarks.

Validatable IRs. Machine-readability is a prerequisite for the generated IR to be *mechanically proven to be sound*. A relatively small part of research in decompilation focuses on arguments for soundness, be it formal or pencil-and-paper. Schwartz et al. [5] present Phoenix, an x86 decompiler to C that guarantees that the structural analysis portion of their decompiler is sound. This is achieved by only using translation schemes that are semantics preserving. Their work does not provide an explicit definition of soundness, nor a proof (neither mechanized or pencil-and-paper). Verbeek et al. [36] present FoxDec, a decompiler from x86-64 to C. They employ formal methods to guarantee that their decompilation results are sound. In addition, the generated C code is recompilable. Their work has some severe restrictions: indirections are not allowed, as are local arrays or passing local pointers. Moreover, their technique introduces an overhead in terms of running time which is sometimes as large as a factor 48.

Both these works share the common property that components of the decompilation chain have been formally proven correct (e.g., control flow recovery). In contrast, our work treats both the de- and the recompiler as a black-box. It argues that soundness means that the recompiled binary can be proven equivalent to the original. Akin to the dichotomy between *verifying a compiler* [21] and doing *translation validation* [27, 29, 33], the approach to verifying (steps of) a decompiler can be distinguished from "decompilation validation" (our approach).

Translation validation has been used to check whether a higher-level representation and a compiled executable are semantically equivalent. However, translation validation considers the higher-level artifact to be the ground truth; the source code provides information on what the lower-level artifact must do. In decompilation, the lower-level artifact is the ground truth. As a concrete example, a typical problem in binary analysis is resolving indirections. If the higher-level artifact is the ground truth, then resolving indirections is possible by examining the source code to see, e.g., what all cases of a `switch` statement are. However, if the lower-level artifact is the ground truth, then indirections need to be resolved by establishing binary-level invariants, a notoriously hard problem in verification [9, 35]. We thus argue more generically that *decompilation validation* is a different problem than *compilation validation*.

To summarize, we argue that this paper presents the first approach to lift binaries to an IR that is recompilable, patchable, and formally validatable.

8 Conclusion & Discussion

This paper presents an approach to lifting PIE x86-64 executables to symbolized NASM. We demonstrate that this approach enables use-cases related to binary-level patching. The lifted NASM comes with a formal certificate, that can be loaded into the Isabelle/HOL theorem prover. If the certificate is proven correct, the original and the recompiled binary are Γ-equivalent.

Reflecting on the role of formal verification in this context, we argue that on the one hand it drastically increases trustworthiness of the lifted IR. This increases trustworthiness of downstream tools as well, like binary verification, binary patching, security analyses and further decompilation steps. On the other hand, it is not a panacea. The nature of binaries is complex, as the execution of a binary is an interplay between the binary itself and the operating system. This work focuses on proving that the same computations are executed, in the same order, and on the same data. However, truly proving that two binaries are equivalent would require a full formal model of the ELF format [18], or more generically, a full formalization of the application binary interface (ABI) between binary and OS [23]. The complexity of the ELF standard may prevent a complete full formal model to ever be completed, but it would be interesting nevertheless to develop formal models of components such as runtime linking and process initialization.

In the approach presented in this paper, the correctness criterion of an IR is that the IR can be recompiled back to the original binary (modulo symbolization). We argue that for the subsequent lifting steps in a larger decompilation chain, such a correctness criterion is too restrictive. For example, once memory regions are abstracted to variables, the recompiler has the freedom to decide how to implement these variables. It no longer needs to recompile back to the same implementation as the original. We argue that to develop a complete formally verified decompiler from binary to source code, the first step from binary to IR is virtually impossible to prove correct. For that reason, this paper presents a "decompilation validation" approach to validate that IR. However, subsequent decompilation steps applied to this IR may each be formally verified themselves. We thus envision a formally verified decompiler where the first step from binary to NASM is achieved through validation, but subsequent steps at higher levels of abstraction are achieved through verified decompilation steps.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which have greatly improved the paper. This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028. Research reported in this paper was also performed in connection with DARPA contract number W912CG-23-C-0024. The views and conclusions in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In Rastislav Bodik, editor, *Compiler Construction*, pages 250–254, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] 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, 2008.

[4] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[5] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 353–368. USENIX Association, 2013.

[6] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

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

[8] Sivarama P Dandamudi. Installing and using NASM. *Guide to Assembly Language Programming in Linux*, pages 153–166, 2005.

[9] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2020.

[10] Jeremy Dawson. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science*, 250(1):55–70, 2009.

[11] Artem Dinaburg and Andrew Ruef. McSema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[12] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[13] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356, October 2011.

[14] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, The University of Texas at Austin, 2016.

[15] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[16] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

[17] Florian Kammüller, Markus Wenzel, and Lawrence C Paulson. Locales a sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs' 99 Nice, France, September 14–17, 1999 Proceedings 12*, pages 149–165. Springer, 1999.

[18] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The missing link: explaining elf static linking, semantically. *ACM SIGPLAN Notices*, 51(10):607–623, oct 2016.

[19] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 169–180, 2022.

[20] Jakub Křoustek, Peter Matula, and P Zemek. RetDec: An open-source machine-code decompiler. In *July 2018*, 2017.

[21] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[22] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for isabelle. *Journal of Automated Reasoning*, 56:261–282, 2016.

[23] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*, 2012.

[24] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[25] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic – improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.

[26] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[27] George C Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, 2000.

[28] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer Science & Business Media, 2002.

[29] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4*, pages 151–166. Springer, 1998.

[30] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. BinTrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 482–501. Springer, 2019.

[31] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[32] Nick Roessler and André DeHon. Scalpel: Exploring the limits of tag-enforced compartmentalization. *J. Emerg. Technol. Comput. Syst.*, 18(1), sep 2021.

[33] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 471–482, 2013.

[34] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–11, 2019.

[35] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 934–949, 2022.

[36] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound C code decompilation for a subset of x86-64 binaries. In *Proceedings of the 18th International Conference on Software Engineering and Formal Methods*, SEFM 2020, September 2020.

[37] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security*, NDSS'17, 2017.

[38] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[39] Makarius Wenzel. Parallel proof checking in isabelle/isar. *PLMMS*, pages 13–29, 2009.

[40] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.