BIRD: A Binary Intermediate Representation for formally verified Decompilation of x86-64 binaries*

Daniel Engel[1][0009−0004−0989−3869], Freek Verbeek[1,2][0000−0002−6625−1123], and Binoy Ravindran[2][0000−0002−8663−739X]

[1] Open University, 6419 AT Heerlen, The Netherlands
{daniel.engel,freek.verbeek}@ou.nl
[2] Virginia Tech, VA 24061 Blacksburg, United States
binoy@vt.edu

Abstract. We present *BIRD: A Binary Intermediate Representation for formally verified Decompilation of x86-64 binaries*. BIRD is a generic language capable of representing a binary program at various stages of decompilation. Decompilation can consist of various small translation passes, each raising the abstraction level from assembly to source code. Where most decompilation frameworks do not guarantee that their translations preserve the program's operational semantics or even provide any formal semantics, translation passes built on top of BIRD must prove their output to be bisimilar to their input. This work presents the mathematical machinery needed to define BIRD. Moreover, it provides two instantiations — one representing x86-64 assembly, and one where registers have been replaced by variables — as well as a formally proven correct translation pass between them. This translation serves both as a practical first step in trustworthy decompilation as well as a proof of concept that semantic preserving translations of low-level programs are feasible. The entire effort has been formalized in the Coq theorem prover. As such, it does not only provide a mathematical formalism but can also be exported as executable code to be used in a decompiler. We envision BIRD to be used to define provably correct binary-level analyses and program transformations.

Keywords: Formal Methods, Decompilation, Static Analysis

1 Introduction

Verification of software on the binary level has numerous advantages: the trusted code base (TCB) is reduced [13], and applicability is widened to software where

* This is the author's version of the work posted here per the publisher's guidelines for your personal use. Not for redistribution. The final authenticated version is published in the Proceedings of the 17th International Conference on Tests and Proofs (TAP 2023), Leicester, United Kingdom, July 18-19, 2023.

source code is not available. The latter may occur in the context of legacy systems, third-party proprietary software, or software that was (partially) handwritten in assembly. However, binary-level verification is notoriously difficult: at this low level of abstraction, there are no variables, no structured control flow, no typing information, no a priori function boundaries, etc. Methods typically are either interactive or tailored towards specific low-level properties. Mostly, binary-level verification consists of static analysis tools that are based on heuristics, rather than based on a formal foundation.

Hypothetically, if one could decompile binaries to high-level code then a large body of research in formal methods at the source code level becomes directly applicable. This would require a formally proven *sound* decompiler: a decompiler whose output is shown to be semantically equivalent to the original binary. Even for the big players in this field [11,18,23,21], decompilation is considered to be more of an art form than an exact science. Typically, decompilers do not produce a semantically equivalent program and a human-in-the-loop is needed to interpret the decompiler's hints to do reverse engineering based on experience [4,5,25].

We argue for the need for formally verified decompilation. Such an approach should consist of numerous small *translation steps* that each lift the program to a representation with a higher level of abstraction. Each such step should be accompanied by a mathematical proof of correctness that shows the step to be semantics-preserving.

In this paper, we present *BIRD*: a Binary Intermediate Representation for formally verified Decompilation. It serves as the data structure on which provably semantics-preserving translations on the level of assembly programs can be implemented. BIRD is a generic, optionally SSA-based language that can represent an x86-64 program at various stages of decompilation. The concept of *storage cells* abstracts over registers and their aliasing behavior, and variables that have strong non-aliasing semantics. *Annotations* can be used to augment storage cells with additional information that may be needed for them to operate correctly. An annotation can carry low-level information like the original bit-pattern or higher-level data like typing information. Lastly, *labels* can be multi-byte values as in the original binary or more abstract identifiers like in assembly dialects such as relocatable Netwide Assembler (NASM) . Figure 1 (which is elaborated in Section 5) shows how BIRD can be used as an intermediate representation (IR) during decompilation. Each of the rectangular boxes contains a representation of the original binary; each of the arrows constitutes a translation step. BIRD is sufficiently generic to model all these representations. It requires – by construction – all translation steps to be semantics-preserving.

We then show two example instantiations for BIRD: 1. the original x86-64 assembly as found in the binary after disassembly (storage cells are registers, no annotations are needed), and 2. the *early BIRD* language in which registers are replaced by variables. In Figure 1, the contributions of this paper are marked in bold. We aim to provide both an IR allowing translation steps in decompilation

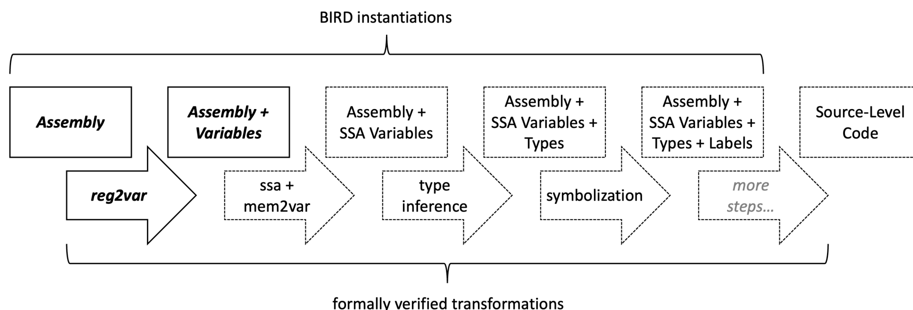

Fig. 1: Overview of an example micro-step decompiler using BIRD

to be tackled in a formal and semantics-preserving fashion, as well as a practical first step toward formally proven correct decompilation.

All definitions are implemented in the Coq theorem prover [6]. With this formal approach, the TCB shrinks to our definitions of the semantics of assembly and the core of Coq itself. The Coq code[1] can be exported to Haskell, making the translation executable [17].

2 Related Work

While not directly applicable to the tasks of decompilation, the Static Single Assignment (SSA) form [7] IRs used in modern compilers have served as inspiration for the design of BIRD. One of the most popular such IRs is the *LLVM IR* [14] which is used in real-world compilers such as *clang*. LLVM is implemented in C++ and does not focus on formally proven transformations, but work by Zakowski et al. formalizes an executable semantics for a subset of the IR in Coq [26]. The Multi-Level Intermediate Representation (MLIR) [15] aims to support different requirements in a unified framework. Similar to BIRD Instantiation, MLIR supports user-defined dialects. As such MLIR is very extensible, but giving formal semantics to it is difficult. The *CompCert* project [16] investigates compilation of C code in a formally proven correct way. Our work mirrors their graph structure for SSA RTL [1] in that the nodes are instructions that are annotated by phi-nodes.

In the field of decompilation, most work focuses on retrieving an approximation of a high-level program that may produce the input binary. *Ghidra* [18] is a large scale reverse engineering suite developed by NSA's Research Directorate. It offers support for a wide variety of assembly dialects which it translates into high P-Code and then into C. Recent work by Naus et al. [19] provides formal semantics for an augmented version of P-Code and shows that the current version cannot be given an executable semantics. Ghidra is fundamentally unable

[1] Made available here: `https://doi.org/10.5281/zenodo.7928215`

to produce an output of which it can be formally proven that it preserves the semantics of the input program. The *Binary Analysis Platform* (BAP) [2] aims to decompile binaries in order to analyze them. Similar to our work, it transforms assembly code into an intermediate language on which state-of-the-art program analyses can be executed. Currently, x86, ARM, MIPS and PowerPC are supported by BAP. As noted by the authors, the lifting process cannot be proven correct as "the semantics of the x86 ISA is not formally defined". Instead, the authors aim to catch bugs through randomized testing. The Interactive Disassembler (IDA) [11] is a disassembler for a large variety of executable formats, including MS-DOS, EXE, ELF, etc. It lifts these binaries into assembly-level programs, but with additional plugins, C code can be generated [9]. No formal argument is given on why this C code represents the same program as the original binary. *Binary Ninja* [23] is another reverse-engineering platform which lifts a range of assembly dialects into several internal IR to analyze them and produce decompiled code. Similarly to the other large decompilation tools, no formal semantics are defined for these languages and as such, no soundness can be proven.

Recent work in the field aims at guaranteeing that the decompiled output is recompilable and semantically equivalent to the input. Schulte et al. [22] use an evolutionary search through a large database of "big code" to arrive at a high-level program. This output can then be recompiled to measure how many bytes are equivalent with the input. The authors report that on a test bed of 19 programs, 10 could be decompiled by this technique to full byte equivalence, the remaining programs matched to $> 80\%$. *Phoenix* [3] uses a more conventional approach based on semantics preserving structural analysis to arrive at an output whose control flow graph (CFG) is provably equivalent to the input's CFG. However, no formal criteria are defined on what constitutes a "semantics preserving" analysis, nor are there formal arguments on why their transformations are correct.

Dasgupta et al. [8] provide formal semantics for more than 774 different x86 instructions. It is implemented in the $\mathbb{K}$ framework to define a correct-by-construction deductive verifier. As such, it can be used to analyze assembly programs directly and perform provable correct transformations on these programs. It cannot represent a higher-level language than x86, thus transformations that lift the abstraction (such as variable recovery) are not possible within this framework. Kennedy et al. [12] provide formal semantics for a subset of x86 in the Coq theorem prover. They define macros to write higher-level assembly-like code directly inside Coq and assemble it into bytes. Parts of this translation are proven to be correct. This formalization also focuses on having x86 as the highest level language and thus cannot support higher abstractions. However, due to being formalized within Coq, its value types serve as a practical foundation on which BIRD is built.

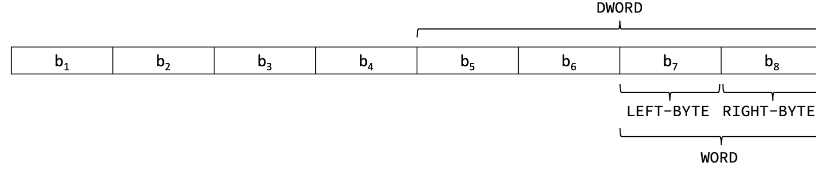

Fig. 2: Word parts of a full QWORD

3 Formalization

All the mathematical structures presented in this section are implemented in the Coq theorem prover. For the sake of simplicity, we will not distinguish between the different universes of Coq's type theory and use $\mathbb{T}$ to mean a type of any level and $\mathbb{P}$ to mean the type of propositions. Relations of type $R : T_1 \to \ldots \to T_n \to \mathbb{P}$ use the set-theoretical notation $R \subseteq T_1 \times \ldots \times T_n$. We make use of Coq's standard library and write $\mathbb{B}$ for booleans, $\mathbb{N}$ for natural numbers, $\mathbb{L}(T)$ for lists of T and $\mathbb{O}(T)$ for optionals of T. The `Some` constructor of optionals is left implicit and $\emptyset$ is the empty element of monoids like lists and optionals. We also use the *bits* library [12] which defines the primitive type for `bytes` (8 tuple of $\mathbb{B}$) together with its operations. We do not use *bits*' bigger types (16, 32, 64 tuple of $\mathbb{B}$) as they are inconvenient for byte-granular operations. Instead, we define $\mathbb{V}_n$ as n tuples of `bytes`. The type $\mathbb{S} :=$ `BYTE` | `WORD` | `DWORD` | `QWORD` contains the valid sizes for values (1, 2, 4, 8 bytes respectively). Word parts $\mathbb{WP} :=$ QWORD | DWORD | WORD | RIGHT-BYTE | LEFT-BYTE correspond to the parts on which a register can be accessed (Figure 2 visualizes these access patterns).

3.1 Generic BIRD

All definitions for the generic IR *BIRD* are polymorphic in the types

- *Storage cells* ($\mathcal{C}$) which serve as the primitive objects into which values can be written. They are themselves writable and they form the basis for addresses.
- *Cell annotations* ($\mathcal{A}$) which express additional information for the cells such as read/write patterns or data types.
- *Labels* ($\mathcal{L}$) which are the locations at which data and instructions can be found. We require labels to be isomorphic to $\mathbb{V}_8$ so that their values can be stored in memory and pointer arithmetic can be performed.

Additionally, the semantics requires functions over storage cells and their state (Γ, to be defined later) to be provided to describe how they are accessed.

- A function of type $PartFunc := (\mathcal{C} \times \mathcal{A}) \to \mathbb{WP}$ to assign each annotated cell a word part to describe how they are narrowed and widened by reads and writes.

– A function of type $\mathit{ReadFunc} := \Gamma \rightarrow \mathcal{C} \rightarrow \mathbb{V}_8$ to describe how a raw cell is read from its state.
– A function of type $\mathit{WriteFunc} := \Gamma \rightarrow \mathcal{C} \rightarrow \mathbb{V}_8 \rightarrow \Gamma$ to describe how a raw cell is written into its state.

Definition 1 (BIRD). *A binary intermediate representation for decompilation (BIRD) is a tuple of these generic elements.*

$$\mathit{bird} := \langle \mathcal{C} : \mathbb{T}, \mathcal{A} : \mathbb{T}, \mathcal{L} : \mathbb{T}, p : \mathit{PartFunc}, r : \mathit{ReadFunc}, w : \mathit{WriteFunc} \rangle$$

For all of the following elements, we define shorthand instantiations. For example, given a BIRD $ir = \langle \mathcal{C}, \mathcal{A}, \mathcal{L}, p, r, w \rangle$, the type of programs in ir is defined $\mathit{Prog}(ir) := \mathit{Prog}(\mathcal{C}, \mathcal{A}, \mathcal{L})$.

Syntax BIRD programs are organized in overapproximating CFGs over instructions. The edges represent possible transitions from one instruction to another. The nodes in the graph are generic labels. Each label is assigned an instruction and any number of phi-assignments.

Instructions ($\mathit{Instr}(ls)$) are dependently typed in their program's labels ls. We categorize instructions into plain operations (`OP`), stack operations (`PUSH`, `POP`), and control flow operations (`JMP`, `CALL`, `RET`). Plain operations contain an opcode (Op), a number of sources ($\vec{\mathit{Src}}$) to read from, a number of destinations ($\vec{\mathit{Dst}}$) to write to, and optionally a successor label ($\in ls$). The successor label points to the next instruction to be executed, if no successor label is given, the instruction terminates the program. The stack operations contain the source to be pushed onto the stack or the destination into which the stack top is popped. They also contain a pair of cells to read the current and write the updated stack pointer value. The jump instruction contains a condition (Cond) deciding whether to jump to one of the `true` labels ($ls_\top$) or to the `false` label ($l_\bot$, the next instruction in the assembly), and a source from which the jump target is read. Similar to the jump, the call contains a return label (l_r) which is pushed on the stack and a number of possible callee labels (ls_c) and a source from which the call target is read. Like the stack instructions, both the call and the return contain a pair of cells to update the stack pointer. The return contains a list of possible return labels (ls_r). A phi-instruction (Φ) consists of exactly one destination *cell* to which it writes and any number of *sources* from which one is read.

Sources (Src) are either an immediate qword value, a cell from which some bytes are read, an expression over cells for which the value is computed, an address from which some bytes are read from memory or the current value of the instruction pointer. Destination (Dst) are only cells or addresses. Expressions (Expr) are formed over cells, qword offsets, cell scalings ($\in \{1, 2, 4, 8\}$) and arithmetic operations thereof. All elements of an expression are optional and can be left empty. Addresses (Addr) are expressions together with a size describing the number of bytes to be read. Absolute addresses ($\mathit{AbsAddr}$) are labels and as such are isomorphic to qwords.

Program	$Prog$	$::= \langle ls : \mathbb{L}(\mathcal{L}), entry \in ls, Code_{\Phi}(ls), Code_{Instr}(ls)\rangle$
Node in program	$Node(p)$	$::= \langle l : \mathcal{L} \mid l \in p.ls\rangle$
Instruction mapping	$Code_{Instr}(ls)$	$::= \langle l : \mathcal{L} \mid l \in ls\rangle \to Instr(ls)$
	$Code_{\Phi}(ls)$	$::= \langle l : \mathcal{L} \mid l \in ls\rangle \to \Phi$
Instructions	$Instr(ls)$	$::= \mid \texttt{OP}\langle Op, \vec{Src}, \vec{Dst}, l \in ls \cup \{\emptyset\}\rangle$ – *plain ops*
		$\mid \texttt{PUSH}\langle Src, \langle \mathcal{C}, \mathcal{C}\rangle, l \in ls\rangle$ – *stack ops*
		$\mid \texttt{POP}\langle Dst, \langle \mathcal{C}, \mathcal{C}\rangle, l \in ls\rangle$
		$\mid \texttt{JMP}\langle Cond, Src, l_{\bot} \in ls, ls_{\top} \subseteq ls\rangle$ – *cf ops*
		$\mid \texttt{CALL}\langle Src, \langle \mathcal{C}, \mathcal{C}\rangle, l_r \in ls, ls_c \subseteq ls\rangle$
		$\mid \texttt{RET}\langle\langle \mathcal{C}, \mathcal{C}\rangle, ls_r \subseteq ls\rangle$
Operation codes	Op	$::= \texttt{MOV} \mid \texttt{ADD} \mid \texttt{CMP} \mid \ldots \mid \texttt{XCHG}$
	$Cond$	$::= \texttt{JMP} \mid \texttt{JZ} \mid \texttt{JNZ} \mid \ldots \mid \texttt{JC}$
Phi node	Φ	$::= \mathbb{L}(Dst \text{ `='} \ \mathbb{L}(Src))$
Operands	Src	$::= Addr \mid \mathcal{C} \times \mathcal{A} \mid Expr \mid \mathbb{V}_8 \mid \texttt{rip} \pm \mathbb{V}_8$
	Dst	$::= Addr \mid \mathcal{C} \times \mathcal{A}$
Addresses	$Addr$	$::= \mathbb{S} \text{ `}\texttt{- PTR}\text{' `}\texttt{[}\text{' } Expr \text{ `}\texttt{]}\text{'}$
	$Expr$	$::= \mathcal{C} \pm (1\|2\|4\|8) * \mathcal{C} \pm AbsAddr$

Fig. 3: Syntax definitions for the generic BIRD language. Everything is polymorphic in $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{L}$.

A dependently typed mapping from all labels of a program to their instructions is called code ($Code_{Instr}$), the dependent mapping to the phi instructions is called phi-code ($Code_{\Phi}$). The type of nodes ($Node(p)$) describes all labels that are in a program p and similar to (phi-)codes, it is indexed by the labels. A program ($Prog$) is a tuple of labels and the (phi-)code for the (phi-)instructions of these labels, and an entry label at which program execution starts. Figure 3 summarizes this syntax.

Notably, programs are not organized in functions to form a call graph, they only contain instructions to form a control flow graph. At this level of representation, function boundaries have not yet been established and a `RET` is not guaranteed to return to its caller. As a consequence, exploits such as return-oriented programming are expressable in this format.

Semantics Figure 4 describes the formal semantics of BIRD. Before these can be explained, we first introduce the constituents used to define these semantics: *reading*, *writing* and *denotations*. Reading and writing from the state requires evaluation of the expressions used in operands of instructions, and up-/downcasting of values. We thus first define these.

Definition 2 (Up- Downcast). *A qword* $q : \mathbb{V}_8$ *can be downcasted to any* $\mathbb{V}_n$ *with a word part* p *by extracting the correct bytes (notation* $\Downarrow_p b$*). For example,* $\Downarrow_{\text{LEFT-BYTE}} \langle b_1, \ldots, b_8\rangle = b_7$*. Symmetrically, the upcasting operator (notation* $\Uparrow_p b$*) transforms a* $\mathbb{V}_n$ *to a* $\mathbb{V}_8$ *by filling all other bytes with 0. For example,* $\Uparrow_{\text{WORD}} \langle b_1, b_2\rangle = \langle 0, 0, 0, 0, 0, 0, b_1, b_2\rangle$.

Definition 3 (Update). *Given a word part $p : \mathbb{WP}$, an old value $o : \mathbb{V}_8$ and a new value $n : \mathbb{V}_8$, the update (notation $o \otimes_p n$) computes an updated $\mathbb{V}_8$ value. Intuitively, this corresponds to the writing behavior of x86 registers.*

$$
\begin{aligned}
\langle o_1, \ldots, o_8 \rangle \otimes_{\text{LEFT-BYTE}} \langle n_1, \ldots, n_8 \rangle &:= \langle\ o_1,\ o_2,\ o_3,\ o_4,\ o_5,\ o_6,\ n_7,\ o_8\ \rangle \\
\langle o_1, \ldots, o_8 \rangle \otimes_{\text{RIGHT-BYTE}} \langle n_1, \ldots, n_8 \rangle &:= \langle\ o_1,\ o_2,\ o_3,\ o_4,\ o_5,\ o_6,\ o_7,\ n_8\ \rangle \\
\langle o_1, \ldots, o_8 \rangle \otimes_{\text{WORD}} \langle n_1, \ldots, n_8 \rangle &:= \langle\ o_1,\ o_2,\ o_3,\ o_4,\ o_5,\ o_6,\ n_7,\ n_8\ \rangle \\
\langle o_1, \ldots, o_8 \rangle \otimes_{\text{DWORD}} \langle n_1, \ldots, n_8 \rangle &:= \langle\ 0,\ 0,\ 0,\ 0,\ n_5,\ n_6,\ n_7,\ n_8\ \rangle \\
\langle o_1, \ldots, o_8 \rangle \otimes_{\text{QWORD}} \langle n_1, \ldots, n_8 \rangle &:= \langle\ n_1,\ n_2,\ n_3,\ n_4,\ n_5,\ n_6,\ n_7,\ n_8\ \rangle
\end{aligned}
$$

Definition 4 (Denotation). *The denotation for the operators Op is given by a* computation *for the actual value (or multiple computations for instruction like `XCHG`), and an* effect *for the change to the flags. Given an $op : Op$, the denotation (notation $[\![\ op\ ]\!]$) returns these two functions. For example, the computation of the `add` instruction computes the sum of both values, ignoring the carry flag. The side effect writes the carry bit into `CF` and populates `ZF`.*

$$
\begin{aligned}
[\![\ \texttt{ADD}\ ]\!] \ := \ \langle\ & \lambda v_1, v_2 \mapsto v_1 + v_2, && \text{– computation} \\
& \lambda v_1, v_2, \sigma \mapsto \mathit{let}\ \langle \mathit{sum}, \mathit{carry} \rangle := v_1 + v_2 && \text{– side effect} \\
& \qquad \mathit{in}\ \sigma[\texttt{CF} \leftarrow \mathit{carry}][\texttt{ZF} \leftarrow \mathit{sum} = 0]\ \rangle
\end{aligned}
$$

For a condition $c : Cond$, the denotation returns a predicate. For example, the predicate for `JMP` is always true, the one for `JZ` returns the `ZF`.

$$
[\![\ \texttt{JMP}\ ]\!] := \lambda\sigma \mapsto \mathit{true} \qquad [\![\ \texttt{JZ}\ ]\!] := \lambda\sigma \mapsto \sigma[\texttt{ZF}]
$$

Definition 5 (State). *The semantics of BIRD programs is given over states (Σ). States consist of a state for cells (Γ), a state for the memory (Θ), a state for flags (Ξ) and the label of the next instruction to be executed ($\boldsymbol{rip}$). The cell-state maps each cell to a qword value where individual bytes can be extracted using the cell's word part, the memory-state maps each absolute address to one byte, bigger regions can be read and written to by multiple read and write applications, the flags-state maps the flags (`ZF`, `OF`, ...) to one bit.*

$$
\Sigma := \langle \Gamma, \Theta, \Xi, \boldsymbol{rip} : \mathcal{L} \rangle \quad \Gamma := \mathcal{C} \to \mathbb{V}_8 \quad \Theta := \mathit{AbsAddr} \to \mathbb{V}_1 \quad \Xi := \mathit{Flag} \to \mathbb{B}
$$

For a full state $\sigma = \langle \gamma, \theta, \xi, \boldsymbol{rip} \rangle : \Sigma$, reading takes a $s : Src$ and returns the correct $\mathbb{V}_n$ for the source by dispatching to a substate.

$$
\sigma[s] := \begin{cases}
v & \text{if } s \text{ is an immediate } v \\
e|_{\text{read}(\gamma)} & \text{if } s \text{ is an expression } e \\
\langle b_0, \cdots, b_{s-1} \rangle & \text{if } s \text{ is an address } \langle s, e \rangle,\ b_i = \gamma(e|_{\text{read}(\gamma)} + i) \\
\Downarrow_{\text{part}(c,a)} (\text{read}(\gamma, \text{c})) & \text{if } s \text{ is an annotated cell } \langle c, a \rangle \\
\boldsymbol{rip} + d & \text{if } s \text{ is a rip relative } \boldsymbol{rip} + d
\end{cases}
$$

Writing is defined similarly, but cells are updated based on their old value

$$\sigma[d \leftarrow v] := \begin{cases} \langle \gamma, \theta', \xi, \boldsymbol{rip} \rangle & \text{if } s \text{ is an address } \langle s, e \rangle \\ & \quad \text{where } \theta' = \theta[e|_{\mathrm{read}(\gamma)+i} \mapsto v_i], i \in \{0, .., 7\} \\ \langle \gamma', \theta, \xi, \boldsymbol{rip} \rangle & \text{if } s \text{ is an annotated cell } \langle c, a \rangle \\ & \quad \text{where } \gamma' = \mathrm{write}(\gamma, c, v') \\ & \quad\quad \text{and } v' = \sigma[\langle c, a \rangle] \otimes_{\mathrm{part}(c,a)} (\Uparrow_{\mathrm{part}(c,a)} v) \end{cases}$$

Here, the expression evaluation ($e|_f$) takes an expression e and a cell evaluation function $f : \mathcal{C} \to \mathbb{V}_8$ to compute the value of e.

$$[b \pm s * i \pm d]|_f := f(b) \pm s \cdot f(i) \pm d$$

The semantics for executing nodes in a program is given by a small step relation $p \vdash \langle \sigma, k \rangle \Rightarrow \langle \sigma', k' \rangle \subseteq \mathit{Prog} \times \mathit{Node}(p) \times \Sigma \times \mathbb{O}(\mathit{Node}(p)) \times \Sigma$ which describes executing node k in state σ to arrive in state σ' where k' needs to be executed next. The execution of the phi instructions is given by the relation $p \vdash \sigma\ \Phi_k^{k'}\ \sigma' \subseteq \mathit{Prog} \times \Sigma \times \Sigma \times \mathit{Node}(p) \times \mathit{Node}(p)$ which assigns the phi's nth source to its destination if k is the nth predecessor of k'. Figure 4 shows the rules for these relations. The transitive closure has one case for terminal and one for nonterminal nodes. The nonterminal rule implements the transitivity and is interleaved with the execution of the phi instructions. For two nodes s and k where k is the nth predecessor of s, the phi rule executes all phi assignments by evaluating to their nth source. An `OP` instruction is executed by first running the operation's main effect and then the side effect to populate the flags. If the instruction has no successor labels, program execution halts. Depending on the evaluation of a `JMP`'s condition, it jumps to the next instruction $k_\perp$ or one of its jump targets $ks_\top$. A `PUSH` first writes its target into the memory at the current stack pointer value s_{sp} and then sets the new stack pointer d_{sp} to the old value minus the size of the target. A `POP` does the opposite by first incrementing the stack pointer and then writing the value read from memory to its target. A `CALL` essentially pushes the label of the next instruction onto the stack and then jumps to its target. Similarly, a `RET` pops the label of the next instruction from the stack and then jumps to it.

Definition 6 (Program semantics). *The semantics of a whole program is given by the transitive closure of the entry node's small step semantics.*

$$p \vdash \sigma_1 \Downarrow \sigma_2 := p \vdash \langle \sigma_1, p.\mathit{entry} \rangle \Rightarrow^* \langle \sigma_2, \emptyset \rangle$$

Translation A translation is a function between programs of two BIRDs together with a semi-decider on program equivalence. When this semi-decider returns true then the input and output of the translation must be bisimilar. A translation step then returns the translated program if it is bisimilar to the input program or nothing if no bisimilarity could be proven.

$$\textsc{terminal}\;\frac{p \vdash \langle \sigma_1, k \rangle \Rightarrow \langle \sigma_2, \emptyset \rangle}{p \vdash \langle \sigma_1, k \rangle \Rightarrow^* \langle \sigma_2, \emptyset \rangle}
\qquad
\textsc{nonterminal}\;\frac{\begin{array}{c} p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_2, k_2 \rangle \\ p \vdash \sigma_2\ \Phi_{k_2}^{k_3}\ \sigma_3 \\ p \vdash \langle \sigma_3, k_3 \rangle \Rightarrow^* \langle \sigma_4, k_4 \rangle \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow^* \langle \sigma_4, k_4 \rangle}$$

$$\textsc{phi}\;\frac{\begin{array}{c} \text{is}-\text{pred}_n(k, s) \qquad phi = \text{phi}(s) \\ \sigma_2 = \text{run}-\text{phis}(n, phi, \sigma_1) \end{array}}{p \vdash \sigma_1\ \Phi_k^s\ \sigma_2}$$

$$\begin{array}{l} \text{run}-\text{phis}(n, ps, \sigma) := \\ \qquad \text{foldr}(\text{run}-\text{phi}(n), \sigma, ps) \\ \text{run}-\text{phi}(n, \langle d, s \rangle, \sigma) := \\ \qquad \sigma[\langle d, \textsc{qword} \rangle \leftarrow \sigma[\langle s_n, \textsc{qword} \rangle]] \end{array}$$

$$\textsc{op}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{OP}(op, \vec{s}, \vec{d}, k_2) \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_2] \qquad \langle \vec{f}, e \rangle = [\![op]\!] \\ \sigma_3 = \sigma_2[\vec{d} \leftarrow \vec{f}(\sigma_2[\vec{s}])] \\ \sigma_4 = e(\sigma_3[\vec{s}], \sigma_3) \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_4, k_2 \rangle}
\qquad
\textsc{jump}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{JMP}(c, s, k_\perp, ks_\top) \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_\perp] \qquad f = [\![c]\!] \\ f(\sigma_2) \implies k_2 \in ks_\top \wedge k_2 = \sigma_2[s] \\ \neg f(\sigma_2) \implies k_2 = k_\perp \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_2, k_2 \rangle}$$

$$\textsc{push}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{PUSH}(s, s_{sp}, d_{sp}, k_2) \\ v = \sigma_2[s] \qquad n = \text{size}(s) \qquad \sigma_1[d_{sp}] \geq s \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_2] \\ \sigma_3 = \sigma_2[n\text{-}\texttt{PTR}[s_{sp}] \leftarrow v] \\ \sigma_4 = \sigma_3[d_{sp} \leftarrow \sigma_3[s_{sp}] - n] \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_4, k_2 \rangle}
\qquad
\textsc{pop}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{POP}(d, s_{sp}, d_{sp}, k_2) \\ v = \sigma_3[n\text{-}\texttt{PTR}[s_{sp}]] \qquad n = \text{size}(d) \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_2] \\ \sigma_3 = \sigma_2[d_{sp} \leftarrow \sigma_2[s_{sp}] + n] \\ \sigma_4 = \sigma_3[d \leftarrow v] \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_4, k_2 \rangle}$$

$$\textsc{call}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{CALL}(d, s_{sp}, d_{sp}, k_r, ks) \\ \sigma_1[s_{sp}] \geq 8 \qquad k_2 = \sigma_2[d] \qquad k_2 \in ks \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_r] \\ \sigma_3 = \sigma_2[\texttt{QWORD-PTR}[s_{sp}] \leftarrow k_2] \\ \sigma_4 = \sigma_3[d_{sp} \leftarrow \sigma_3[s_{sp}] - 8] \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_4, k_2 \rangle}
\qquad
\textsc{ret}\;\frac{\begin{array}{c} \text{instr}(k_1) = \texttt{RET}(s_{sp}, d_{sp}, ks) \\ k_2 = \sigma_1[\texttt{QWORD-PTR}[s_{sp}]] \qquad k_2 \in ks \\ \sigma_2 = \sigma_1[\texttt{rip} \leftarrow k_2] \\ \sigma_3 = \sigma_2[d_{sp} \leftarrow \sigma_2[s_{sp}] + 8] \end{array}}{p \vdash \langle \sigma_1, k_1 \rangle \Rightarrow \langle \sigma_3, k_2 \rangle}$$

Fig. 4: Formal semantics for BIRD programs. From top to bottom: The transitive closure for (phi-)instruction semantics, the phi semantics, and the small step semantics for instructions.

Definition 7 (Bisimulation). *Given two BIRDs ir_1, ir_2 and two programs $p_1 : Prog(ir_1), p_2 : Prog(ir_2)$, the programs are considered* bisimilar *($p_1 \sim p_2$) if there exists a relation $R \subseteq \Sigma(ir_1) \times \Sigma(ir_2)$ such that R and R^{-1} are* simulations. *R is a simulation, if for every $\langle \sigma_1, \sigma_2 \rangle \in R$ and $\sigma_1' : \Sigma(ir_1)$*

$$p_1 \vdash \sigma_1 \Downarrow \sigma_1' \implies \exists \sigma_2', p_2 \vdash \sigma_2 \Downarrow \sigma_2' \wedge R(\sigma_1', \sigma_2')$$

Definition 8 (Translation). *Given two birds ir_1, ir_2, we define translations and translation steps*

$$\text{translation} := \langle \tau : \mathit{Prog}(ir_1) \to \mathit{Prog}(ir_2), (\equiv) \subseteq \mathit{Prog}(ir_1) \times \mathit{Prog}(ir_2) \rangle$$
$$\text{translation}-\text{step}(\langle \tau, \equiv \rangle) := \lambda p \mapsto \ \ \mathit{if}\ p \equiv \tau(p)\ \ \mathit{then}\ \tau(p)\ \ \mathit{else}\ \emptyset$$

where translations *have the requirement*

$$\forall p, p \equiv \tau(p) \implies p \sim \tau(p)$$

The composition of two steps $step_1, step_2$ is defined as the monadic bind.

$$step_1 \circ step_2 := \lambda p \mapsto \mathit{if}\ step_1(p) = p'\ \ \mathit{then}\ step_2(p')\ \ \mathit{else}\ \emptyset$$

Note that if a translation step returns the translation of an input program then the translation is bisimilar to the input. This, combined with transitivity of bisimulation, ensures that if the composition of two steps returns the translation of an input program then the composed translation is bisimilar to the input.

3.2 Instantiation 1: X86

For X86 assembly, the storage cells are registers and no annotations (a.k.a the `unit` tuple) are needed. Reading and writing is implemented with the aliasing semantics of registers.

Definition 9 (Register). *Registers are the general-purpose registers $Reg ::= \texttt{rax} \mid \texttt{eax} \mid \ldots \mid \texttt{r15b}$. A register's word part is given as follows*

$$\text{part}_{reg}(\texttt{rax}, _) := \text{QWORD} \qquad \text{part}_{reg}(\texttt{eax}, _) := \text{DWORD} \qquad \text{part}_{reg}(\texttt{ax}, _) := \text{WORD}$$
$$\text{part}_{reg}(\texttt{al}, _) := \text{RIGHT-BYTE} \qquad \text{part}_{reg}(\texttt{ah}, _) := \text{LEFT-BYTE} \qquad \ldots$$

Definition 10 (Register read, write). *Given a register state $\gamma : Reg \to \mathbb{V}_8$ and a register $r : Reg$, reading is the function application*

$$\text{read}_{reg}(\gamma, r) := \gamma(r)$$

and writing is the aliasing function update

$$\text{write}_{reg}(\gamma, r, v) := \text{W}_{regs}(\gamma, \text{aliases}(r), v)$$
$$\text{W}_{regs}(\gamma, rs, v) := \text{fold}((\lambda\gamma', r \mapsto \gamma'[r \mapsto v]), \gamma, rs)$$
$$\text{aliases}(\texttt{rax}) := [\texttt{rax}, \texttt{eax}, \texttt{ax}, \texttt{al}, \texttt{ah}], \ \ldots$$

Definition 11 (Well formed X86 state). *A register state $\gamma : Reg \to \mathbb{V}_8$ is* well formed *if all aliasing registers contain the same $\mathbb{V}_8$ value.*

$$\text{wf}(\gamma) := \forall r_1 r_2, \text{aliases}(r_1) = \text{aliases}(r_2) \implies \gamma(r_1) = \gamma(r_2)$$

Definition 12 (X86 BIRD). *The X86 language is the IR with registers, no annotations, absolute addresses as labels and the aforementioned aliasing semantics for registers.*

$$X86 := \langle Reg, \langle\rangle, AbsAddr, \text{part}_{reg}, \text{read}_{reg}, \text{write}_{reg} \rangle$$

3.3 Instantiation 2: Early BIRD

For the Early BIRD language, the storage cells are mutable variables and the annotations are the word parts corresponding to the word parts of the registers from which the variable originates. Reading and writing have no aliasing semantics.

Definition 13 (Mutable Variable). *A* mutable variable *is a string identifier* $Var := \texttt{string}$. *Its word part is determined entirely by its annotation*

$$\text{part}_{var}(_, a) := a$$

Definition 14 (Variable read, write). *Given a variable state* $\gamma : Var \to \mathbb{V}_8$ *and a variable* $var : Var$*, reading is again the function update and writing is the non-aliasing function update*

$$\text{read}_{var}(\gamma, var) := \gamma(var) \qquad \text{write}_{var}(\gamma, var, v) := \gamma[var \mapsto v]$$

Definition 15 (Early BIRD). *The Early BIRD language is the IR with variables, word parts as annotations, absolute addresses as labels and the aforementioned non aliasing semantics for variables*

$$EarlyBIRD := \langle Var, \mathbb{WP}, AbsAddr, \text{part}_{var}, \text{read}_{var}, \text{write}_{var} \rangle$$

4 X86 To Early BIRD

```
1 MOV R8D, EDI
2 LEA RAX, [R8 - 1]
3 INC R8B
```

(a) Original X86 code

```
1 MOV R8(dword), DI(dword)
2 LEA AX(qword), [R8 - 1]
3 INC R8(right byte)
```

(b) Corresponding Early BIRD code

Fig. 5: Example of the `reg2var` translation.

In this section, we define the translation `reg2var` where the function τ is overloaded for all syntactical elements. We then define a congruence relation $\cong$ between X86 states and Early BIRD states and extend τ to also translate states. These two definitions serve as the bisimulation relation and the method to compute new related states needed for definition 7. Based on these definitions, we show that τ always produces bisimilar states, thus the semi-decider on program equivalence is the constant `true` function. Figure 5 shows an example X86 program together with its `reg2var` transformation.

Definition 16. *For (annotated) registers,* τ *returns (annotated) cells. All aliasing registers are mapped to the same variable*

$$\tau(\textit{rax}) := \text{“}AX\text{''} \quad \tau(\textit{eax}) := \text{“}AX\text{''} \quad \ldots \quad \tau(\textit{r15w}) := \text{“}15\text{''} \quad \tau(\textit{r15b}) := \text{“}15\text{''}$$

and the resulting variables annotation is the word part of the registers

$$\tau(\langle r : \mathit{Reg}, a : \boldsymbol{unit}\rangle) := \langle \tau(r), \mathrm{part}_{reg}(r, a)\rangle$$

Sources are translated by translating the contained data, all other syntactical elements work similarly.

$$\tau(s) := \begin{cases} v & \text{if } s \text{ is an immediate } v \\ \tau(e) & \text{if } s \text{ is an expression } e \\ \langle s, \tau(e)\rangle & \text{if } s \text{ is an address } \langle s, e\rangle \\ \tau(\langle r, a\rangle) & \text{if } s \text{ is an annotated register } \langle r, a\rangle \\ \mathtt{rip} + d & \text{if } s \text{ is an rip relative } \mathtt{rip} + d \end{cases}$$

All nodes that appear in a program $p : \mathit{Prog}(X86)$ also appear in the program $\tau(p)$ because τ does not change the control flow structure, only the instructions. As such, we can implicitly cast any node from $\mathit{Node}(p)$ to $\mathit{Node}(\tau(p))$. In particular, the statements $p \vdash \langle \sigma_1, k\rangle \Rightarrow \langle \sigma_1', k'\rangle$ and $\tau(p) \vdash \langle \sigma_2, k\rangle \Rightarrow \langle \sigma_2', k'\rangle$ have meaningful semantics.

Definition 17 (Reg2Var congruence). *The relation $\cong$ between a X86 state σ_1 and an Early BIRD state σ_2 requires the memory, flags and* ***rip*** *of both states to be identical. For the cells, the values of all registers must match their translated counterparts.*

$$\begin{aligned} \sigma_1 \cong \sigma_2 := \forall(a : \mathit{AbsAddr}), \sigma_1[a] = \sigma_2[a] \quad & \wedge \forall(f : \mathit{Flag}), \sigma_1[f] = \sigma_2[f] \\ \wedge \forall(c : \mathit{Reg} \times \boldsymbol{unit}), \sigma_1[c] = \sigma_2[\tau(c)] & \wedge \sigma_1[\boldsymbol{rip}] = \sigma_2[\boldsymbol{rip}] \end{aligned}$$

This will be the bisimulation relation R for Definition 7.

Definition 18. *The translation for states $\tau : \Sigma(X86) \to \Sigma(\mathit{EarlyBIRD})$ keeps memory, flags and* ***rip*** *unchanged and evaluates all variables by forwarding to the corresponding register.*

$$\begin{aligned} \tau(\langle \gamma, \theta, \xi, \boldsymbol{rip}\rangle) &:= \langle\langle \gamma', \theta, \xi, \boldsymbol{rip}\rangle\rangle \\ \gamma' &:= \lambda(v : \mathit{Var}) \mapsto \gamma(r), \text{ where } \tau(r) = v \end{aligned}$$

For the rest of the section, we can assume such an r with $\tau(r) = v$ to exist as all variables in this transformation originate from registers. This will be the state σ_2' for Definition 7.

To show bisimilarity between any program p and its translation $\tau(p)$, we first need to show that state congruence preserves all values. First, we show that registers and their translation have the same value (Lemma 1), that expressions and their translation evaluate to the same value (Lemma 2) and that reading from a source is the same as reading from its translation (Lemma 3). We then show that writing preserves congruence (Lemma 4). Based on these, we show that executing (phi-)instructions preserves the small-step semantics (Lemmas 5,6

and 7), and extend this to the transitive closure (Lemma 8). Finally, we show that the semantics of the entire program is preserved (Corollary 1) and thus, $\cong$ is a bisimulation (Theorem 1). For the rest of the section, we always assume X86-states to be well-formed.

Lemma 1. *Given two states σ_1, σ_2, for all registers r and annotations a*

$$\sigma_1 \cong \sigma_2 \implies \sigma_1[\langle r, a\rangle] = \sigma_2[\tau(\langle r, a\rangle)]$$

Proof. *Unfolding the definitions, we need to show $\Downarrow_{\text{part}(r,a)} \gamma_1(r) =\Downarrow_{\tau(\langle r,a\rangle)} \gamma_2(\tau(r))$. By definition, $\text{part}(r, a)$ and $\tau(\langle r, a\rangle)$ are equal. By a case analysis on r, we need to show cases such as $\gamma_1(\texttt{EAX}) = \gamma_2(\text{“}AX'')$. This is true by the assumption $\sigma_1 \cong \sigma_2$.* □

Lemma 2. *Given two states γ_1, γ_2, for all expressions $e = [d \pm b \pm s * i]$*

$$\gamma_1 \cong \gamma_2 \implies e|_{\text{read}(\gamma_1)} = \tau(e)|_{\text{read}(\gamma_2)}$$

Proof. *Unfolding the definition of expression evaluation, we need to show $d \pm \gamma_1[b] \pm s \cdot \gamma_1[i] = d \pm \gamma_2[\tau(b)] \pm s \cdot \gamma_2[\tau(i)]$. By Lemma 1, we have $\gamma_1[b] = \gamma_2[\tau(b)]$ and $\gamma_1[i] = \gamma_2[\tau(i)]$.* □

Lemma 3. *Given two states σ_1, σ_2, for all sources s*

$$\sigma_1 \cong \sigma_2 \implies \sigma_1[s] = \sigma_2[\tau(s)]$$

Proof. *Case analysis over s. For an immediate v, $\tau(s) = v$ and $\forall \sigma, \sigma[v] = v$. For an expression e, Lemma 2 shows that evaluation of e and $\tau(e)$ yield the same result. For an address $\langle s, e\rangle$, s stays the same and the evaluation of e and $\tau(e)$ are equal. Reading from $\langle s, e\rangle$ in σ_1 and $\tau(\langle s, e\rangle)$ in σ_2 is the same by $\sigma_1 \cong \sigma_2$. For an annotated register $\langle r, a\rangle$, Lemma 1 shows the goal. For a* ***rip*** *relative* $\textbf{\textit{rip}} + d$, *equality follows from $\sigma_1 \cong \sigma_2$.* □

Lemma 4. *Given two states σ_1, σ_2, for all destinations d and values v*

$$\sigma_1 \cong \sigma_2 \implies \sigma_1[d \leftarrow v] \cong \sigma_2[\tau(d) \leftarrow v]$$

Proof. *Case analysis over d. For an annotated register $\langle r, a\rangle$, we only need to consider the cell state parts as all other state parts are unchanged by the write. We need to show $\text{write}_{reg}(\gamma_1, r, a, v) = \text{write}_{var}(\gamma_2, \tau(\langle r, a\rangle), \tau(r), v)$. By Lemma 1 we know that the old values of r in γ_1 and $\tau(r)$ in γ_2 are equivalent, thus updating with v results in the same value. Again, by Lemma 1, we know that reading from r and $\tau(r)$ after writing results in the same value. For an address $\langle s, e\rangle$, we only need to consider the memory state part. By Lemma 2, we know that e and $\tau(e)$ evaluate to the same value, thus the same memory update is performed. Also by Lemma 2, Reading after the the memory update yields the same value.* □

Lemma 5. *Given two states* σ_1, σ_2*, a list of source registers* s*, a destination register* d *and a natural number* n*, we have*

$$\sigma_1 \cong \sigma_2 \implies \text{run-phi}(n, \langle d, s\rangle, \sigma_1) \cong \text{run-phi}(n, \tau\langle d, s\rangle, \sigma_2)$$

Unfolding the definitions, we need to show

$$\sigma_1[d \leftarrow \sigma[\langle s_n, \text{QWORD}\rangle]] \cong \sigma_2[\tau(d) \leftarrow \sigma[\langle \text{map}(\tau, s)_n, \text{QWORD}\rangle]]$$

By Lemma 3, we know that $\sigma_1[\langle s_n, \text{QWORD}\rangle]$ *and* $\sigma_2[\langle \text{map}(\tau, s)_n, \text{QWORD}\rangle]$ *evaluate to the same* $v : \mathbb{V}_8$*. The remaining* $\sigma_1[d \leftarrow v] \cong \sigma_2[\tau(d) \leftarrow v]$ *follows from Lemma 4.* □

Lemma 6. *Given a program* $p : \mathit{Prog}(X86)$*, two nodes in the program* k_1, k_2*, two states* σ_1, σ_1'*, a state* σ_2*, and* $\sigma_2' := \tau(\sigma_1')$*, we have*

$$\sigma_1 \cong \sigma_2 \wedge p \vdash \sigma_1\ \Phi_{k_1}^{k_2}\ \sigma_1' \implies \tau(p) \vdash \sigma_2\ \Phi_{k_1}^{k_2}\ \sigma_2'$$

Proof. *Inversion on the phi step relation. We have an* n*, such that* $\text{is-pred}_n(k_1, k_2)$ *and* $\sigma_1' = \text{run-phis}(n, \text{phi}(k_2), \sigma_1)$*. We show* $\sigma_2' = \text{run-phis}(\text{n}, \tau(\text{phi}(\text{k}_2)), \sigma_2)$ *by an induction on* $\text{phi}(k_2)$ *and application of Lemma 5.* □

Lemma 7. *Given a program* $p : \mathit{Prog}(X86)$*, two nodes in the program* k_1, k_2*, two states* σ_1, σ_1'*, a state* σ_2*, and* $\sigma_2' := \tau(\sigma_1')$*, we have*

$$\sigma_1 \cong \sigma_2 \wedge p \vdash \langle k_1, \sigma_1\rangle \Rightarrow \langle k_2, \sigma_1'\rangle \implies \tau(p) \vdash \langle k_1, \sigma_2\rangle \Rightarrow \langle k_2, \sigma_2'\rangle$$

Proof. *We show the* OP *rule as an example. All other rules are similar. Doing an inversion, we get 1.* p, $\text{instr}(k_1) = \mathtt{OP}(op, \vec{s}, \vec{d}, k_2)$*, 2.* $\sigma_a = \sigma_1[\mathtt{rip} \leftarrow k_2]$*, 3.* $\langle \vec{f}, e\rangle = [\![\ op\ ]\!]$*, 4.* $\sigma_b = \sigma_a[\vec{d} \leftarrow \vec{f}(\sigma_a[\vec{s}])]$ *and 5.* $\sigma_1' = e(\sigma_b[\vec{s}], \sigma_b)$*. We need to show:*

- $\sigma_a \cong \sigma_a'$ *where* $\sigma_a' = \sigma_2[\mathtt{rip} \leftarrow k_2]$*. Follows by definition of writing to* $\mathtt{rip}$*.*
- $\sigma_b \cong \sigma_b'$ *where* $\sigma_b' = \sigma_a'[\tau(\vec{d}) \leftarrow \vec{f}(\sigma_a'[\tau(\vec{s})])]$*. Follows from Lemmas 3 and 4.*
- $\sigma_1 \cong \sigma_2'$ *where* $\sigma_2' = e(\sigma_b'[\tau(\vec{s})])$*. Follows from Lemmas 3 and 4.*

□

Lemma 8. *Given a program* $p : \mathit{Prog}(X86)$*, two nodes in the program* k_1, k_2*, two states* σ_1, σ_1' *a state* σ_2*, and* $\sigma_2' := \tau(\sigma_1')$*, we have*

$$\sigma_1 \cong \sigma_2 \wedge p \vdash \langle k_1, \sigma_1\rangle \Rightarrow^* \langle k_2, \sigma_1'\rangle \implies \tau(p) \vdash \langle k_1, \sigma_2\rangle \Rightarrow^* \langle k_2, \sigma_2'\rangle$$

Proof. *Induction over* $p \vdash \langle k_1, k_2\rangle \Rightarrow^* \langle \sigma_1, \sigma_1'\rangle$ *with* σ_2 *generalized. The* TERMINAL *base case follows by Lemma 7 and the* NONTERMINAL *inductive case follows from Lemmas 6, 7 and the induction hypothesis.* □

Corollary 1. *Given a program* $p : \mathit{Prog}(X86)$ *and the states* $\sigma_1, \sigma_1', \sigma_2$ *and* $\sigma_2' := \tau(\sigma_1')$*, we have*

$$\sigma_1 \cong \sigma_2 \wedge p \vdash \sigma_1 \Downarrow \sigma_1' \implies \sigma_1' \cong \sigma_2' \wedge \tau(p) \vdash \sigma_2 \Downarrow \sigma_2'$$

Proof. $\sigma_1' \cong \sigma_2'$ *follows by definition of* τ*, the rest by Definition 6 and Lemma 8.* □

Theorem 1. *Given a program* $p : Prog(X86)$*, we have* $p \sim \tau(p)$

Proof. *We need to provide a relation* $R \subseteq \Sigma(X86) \times \Sigma(EarlyBIRD)$*, such that both* R *and* R^{-1} *are simulations. We choose* $R := (\cong)$*.*

$$\forall \langle \sigma_1, \sigma_2 \rangle \in (\cong), \forall \sigma_1', p \vdash \sigma_1 \Downarrow \sigma_2 \implies \exists \sigma_2', \sigma_2 \cong \sigma_2' \wedge \tau(p) \vdash \sigma_2 \Downarrow \sigma_2'$$

We provide $\sigma_2' := \tau(\sigma_1')$*, the remaining*

$$\sigma_1 \cong \sigma_2 \wedge p \vdash \sigma_1 \Downarrow \sigma_1' \implies \sigma_1' \cong \tau(\sigma_1') \wedge \tau(p) \vdash \sigma_2 \Downarrow \tau(\sigma_1')$$

is exactly Corollary 1. The case for $(\cong)^{-1}$ *is similar.* □

The first step in the translation chain (Figure 1) is the `reg2var` step between the $X86$ and the $EarlyBIRD$ languages. Its translation is implemented by the τ function defined in this section. By Theorem 1, this translation step always produces an $EarlyBIRD$ program whose semantics are equivalent to the semantics of the original $X86$ program. Thus, the semi-decider for program equivalence between input and output of the τ function is the constant `true` function. By Definition 8, we obtain a provably correct translation step.

5 Conclusion

In this paper, we presented the mathematical framework *BIRD* to describe low-level programs in a generic assembly language, translations between different instantiations of that language, and the soundness criteria thereof. We presented two example instantiations: X86 and Early BIRD with their full formal semantics, and showed a translation step to replace the registers of the former with mutable variables of the latter and proofed soundness of this translation.

As such, BIRD is the first of its kind as it allows decompilation to be performed in an exact and provably correct way that requires no human intervention. Opposed to most of the competing platforms, it does not aim to produce human-readable code and instead produces machine analyzable code.

For usage in a real-world decompiler, the limited set of instructions presented in this paper can be extended by leveraging work such as [8,10,12,24]. By implementing the language definition and transformation in Coq, we are able to use its language extraction feature to produce around 500 lines of Haskell code to be used in our internal x86-64 decompilation suite.

Apart from the `reg2var` translation that was introduced here, we see three more steps as the immediate future work that can directly build upon BIRD. Figure 1 shows a hypothetical compiler that uses these steps to leverage the low-level assembly into fully typed, SSA-based programs.

ssa, mem2var SSA-form programs are often used by static analysis tools to better reason about mutable variables. In its current state, an Early BIRD program still uses mutable variables, but the generic definition supports the definition of immutable SSA variables too. Notably, phi nodes with their semantics based on incoming edges are defined as part of Figure 4. The register-based variables can already be SSA transformed, but there still remain memory accesses that make statically analyzing an assembly program hard. We argue that these memory accesses need to be replaced by SSA variables wherever possible.

symbolization The Early BIRD language uses $\mathbb{V}_8$ values as the labels for all instructions. As such, they can be the result of computations to implement indirect jumps. If one was to reorder instructions in the program, or insert new ones, these $\mathbb{V}_8$ labels would no longer match and indirect jumps would break. A symbolization step can introduce more abstract, position-independent labels. This would allow for semantics preserving, *structure altering* translations.

type inference The registers modeled so far (and thus the variables) are based on 1,2,4 and 8 byte integers. There is no information yet on signedness of values or whether or not they form more complex compound data structures, but such information can be used to guide program analyses [20]. An extension of the current framework may see the introduction of floating point instructions and SIMD registers.

Acknowledgements. 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.

References

1. Barthe, G., Demange, D., Pichardie, D.: A formally verified SSA-based middle-end: Static single assignment meets compcert. In: Proceedings of the 21st European Conference on Programming Languages and Systems. p. 47–66. ESOP'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_3
2. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/0.1007/978-3-642-22110-1_37
3. Brumley, D., Lee, J., Schwartz, E.J., Woo, M.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: USENIX Security Symposium (2013)
4. Burk, K., Pagani, F., Kruegel, C., Vigna, G.: Decomperson: How humans decompile and what we can learn from it. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2765–2782. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/burk
5. Canzanese, R., Oyer, M., Mancoridis, S., Kam, M.: A survey of reverse engineering tools for the 32-bit microsoft windows environment (01 2005)

6. Coq Development Team: The coq proof assistant, https://coq.inria.fr/, accessed: May 12, 2023
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (oct 1991). https://doi.org/10.1145/115372.115320
8. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1133–1148. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314601
9. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. IT Pro, No Starch Press (2008), https://books.google.de/books?id=BoFaZ1dB1HOC
10. Hamlen, K.W., Fisher, D., Lundquist, G.R.: Source-free machine-checked validation of native code in coq. In: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation. p. 25–30. FEAST'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338502.3359759
11. Hex Rays: Ida pro, https://hex-rays.com/ida-pro/, accessed: May 12, 2023
12. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: The world's best macro assembler? In: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming. p. 13–24. PPDP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2505879.2505897
13. Kumar, R., Mullen, E., Tatlock, Z., Myreen, M.O.: Software verification with ITPs should use binary code extraction to reduce the TCB. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. pp. 362–369. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_21
14. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665
15. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
16. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814, http://xavierleroy.org/publi/compcert-CACM.pdf
17. Letouzey, P.: Extraction in coq: An overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms. pp. 359–369. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_39
18. N. S. Agency: Ghidra, https://ghidra-sre.org/, accessed: May 12, 2023
19. Naus, N., Verbeek, F., Ravindran, B.: A formal semantics for P-Code. In: 14th International Conference on Verified Software: Theories, Tools, and Experiments (2022). https://doi.org/10.1007/978-3-031-25803-9_7

20. Palsberg, J.: Type-based analysis and applications. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (07 2001). https://doi.org/10.1145/379605.379635
21. PNF Software: Jeb decompiler, https://www.pnfsoftware.com/, accessed: May 12, 2023
22. Schulte, E., Ruchti, J., Noonan, M., Ciarletta, D., Loginov, A.: Evolving exact decompilation. In: Shoshitaishvili, Y., Wang, R.F. (eds.) Workshop on Binary Analysis Research. San Diego, CA, USA (Feb 18-21 2018). https://doi.org/10.14722/bar.2018.23008, https://www.ndss-symposium.org/ndss2018/bar-workshop-programme/
23. Vector 35 Inc.: Binary ninja, https://binary.ninja/, accessed: May 12, 2023
24. Verbeek, F., Bharadwaj, A., Bockenek, J., Roessle, I., Weerwag, T., Ravindran, B.: X86 instruction semantics and basic block symbolic execution. Archive of Formal Proofs (October 2021), https://isa-afp.org/entries/X86_Semantics.html, Formal proof development
25. Yakdan, K., Dechand, S., Gerhards-Padilla, E., Smith, M.: Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 158–177. IEEE Computer Society, Los Alamitos, CA, USA (may 2016). https://doi.org/10.1109/SP.2016.18
26. Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for LLVM IR. Proc. ACM Program. Lang. **5**(ICFP) (aug 2021). https://doi.org/10.1145/3473572