A Formal Semantics for P-Code

Nico Naus[1](✉), Freek Verbeek[1,2], Dale Walker[1], and Binoy Ravindran[1]

[1] Virginia Tech, Blacksburg, USA
{niconaus,freek,dalewalker,binoy}@vt.edu
[2] Open University of The Netherlands, Heerlen, The Netherlands

Abstract. Decompilation is currently a widely used tool in reverse engineering and exploit detection in binaries. Ghidra, developed by the National Security Agency, is one of the most popular decompilers. It decompiles binaries to high P-Code, from which the final decompilation output in C code is generated. Ghidra allows users to work with P-Code, so users can analyze the intermediate representation directly. Several projects make use of this to build tools that perform verification, decompilation, taint analysis and emulation, to name a few. P-Code lacks a formal semantics, and its documentation is limited. It has a notoriously subtle semantics, which makes it hard to do any sort of analysis on P-Code. We show that P-Code, as-is, cannot be given an executable semantics. In this paper, we augment P-Code and define a complete, executable, formal semantics for it. This is done by looking at the documentation and the decompilation results of binaries with known source code. The development of a formal P-Code semantics uncovered several issues in Ghidra, P-Code, and the documentation. We show that these issues affect projects that rely on Ghidra and P-Code. We evaluate the executability of our semantics by building a P-Code interpreter that directly uses our semantics. Our work uncovered several issues in Ghidra and allows Ghidra users to better leverage P-Code.

Keywords: Decompilation · P-Code · Formal semantics

1 Introduction

After more than 60 years of research, the field of decompilation is currently very mature. A plethora of open source and commercial decompilation tools exist [1,7,11–13]. They are widely used to recover source code from binaries and to detect software vulnerabilities.

Ghidra[1] is one of those tools, developed by the National Security Agency (NSA), and made public and open source a few years ago. Recent comparative studies show that it ranks among the top performing decompilers [5]. At the heart of Ghidra lies P-Code, an intermediate representation that exists at two levels of abstraction. The first is a direct one-to-many translation of the disassembled assembly instructions, called low P-Code. The second is high P-Code,

[1] https://ghidra-sre.org/.

A. Lal and S. Tonetta (Eds.): VSTTE 2022, LNCS 13800, pp. 111–128, 2023.
https://doi.org/10.1007/978-3-031-25803-9_7

which is the result of various transformations on the low P-Code from Ghidra's decompiler. Since we focus our work on high P-Code, we will simply refer to it as P-Code from now on.

From the P-Code, Ghidra constructs C code, which is the final decompilation result. Users can define their own analyses over high P-Code, customizing the decompilation process or extracting information from it. We have come across several projects that perform a wide range of analyses on P-Code. Verification [6], decompilation [14] and taint analysis [2], to name a few. However, the P-Code documentation is very limited and P-Code lacks any form of formal semantics. On top of that, P-Code has a notoriously subtle semantics, as will become clear in the rest of this paper. This limits the usability of P-Code for formal analysis, since there is no way to know if the analysis that is being performed is correct with respect to the language semantics.

In this paper, we develop a formal semantics for high P-Code. We base our work on Ghidra 10.1.4 released May 2022, which comes with P-Code documentation last updated September 5th, 2019 [9]. Ghidra does come with a P-Code interpreter, but only for low P-Code. This means that there is no ground-truth that we can base our high P-Code semantics on. We therefore start our investigation by looking at the P-Code documentation. Since this is rather limited, we additionally run experiments for P-Code instructions that are unclear. These experiments consist of compiling several C programs to binary, decompiling them using Ghidra, and comparing the high P-Code and decompiled C code to the original source code. During the development of a formal P-Code semantics, we uncover several issues in Ghidra, P-Code and the P-Code documentation. These issues all stem from inconsistencies in documentation, Ghidra's output via UI or API, and in some cases the inability to formulate executable P-Code semantics. As-is, P-Code cannot be given an executable semantics. To overcome the shortcomings of P-Code, we extend the language with additional information. For the extended P-Code semantics, we define a formal operational semantics. We argue that projects relying on P-Code are directly affected by these issues, and could benefit from a formal P-Code semantics. Since there is no interpreter for high P-Code available, we are only able to validate our semantics by writing an interpreter for high P-Code to show that our semantics are executable. We have shared our results with the NSA prior to publication, and they acknowledge all issues identified in this paper.

More specifically, we make the following contributions:

- An extended P-Code language.
- A formal syntax and semantics for extended P-Code.
- A P-Code interpreter written in Haskell.
- An overview of several bugs, issues and inconsistencies in Ghidra, P-Code and documentation.

The P-Code interpreter written in Haskell and the accompanying Ghidra script written in Java are publicly available[2].

[2] https://github.com/niconaus/pcode-interpreter
https://github.com/niconaus/PCode-Dump.

Section 2 first gives an introduction to Ghidra, its assembly translator SLEIGH and low & high P-Code. Section 3 describes and motivates our design choices. Section 4 describes the P-Code syntax and Sect. 5 gives its semantics. We list an overview of the bugs, issues and inconsistencies we have found, and a response from the NSA on these, in Sect. 6. Section 7 presents related work and Sect. 8 concludes.

2 Ghidra, SLEIGH and P-Code

Ghidra is an open-source reverse engineering tool, developed by the National Security Agency. It is capable of decompiling binaries of a wide variety of architectures. To do so, it first disassembles the binary using its custom disassembler SLEIGH, and performs several analysis steps. The end result is high-level C code, as well as detailed control flow information. A recent study compared other well known decompilers like IDA-pro [7] and Angr [13] to Ghidra, by evaluating 1760 binaries, using the correctly identified function starts (CFS) metric [5]. They found that Ghidra performs above average.

Address	x86-64	low P-Code	high P-Code
0x100000e70	push rbp	$Uea00:8 = RBP	$Ud100:1 = EDI s< 3:4
		RSP = RSP - 8:8	if $Ud100:1 goto 0x100000e95:1
		*RSP = $Uea00:8	
0x100000e71	mov rbp, rsp	RBP = RSP	
0x100000e74	mov dword ptr [rbp - 8], edi	$U3100:8 = RBP + -8:8	
		$Ubf00:4 = EDI	
		*$U3100:8 = $Ubf00:4	
0x100000e77	cmp dword ptr [rbp - 8], 2	$U3100:8 = RBP + -8:8	
		$Ubf80:4 = *$U3100:8	
		CF = $Ubf80:4 s< 2:4	
		$Ubf80:4 = *$U3100:8	
		OF = sborrow($Ubf80:4, 2:4)	
		$Ubf80:4 = *$U3100:8	
		$U29000:4 = $Ubf80:4 - 2:4	
		SF = $U29000:4 s< 0:4	
		ZF = $U29000:4 == 0:4	
		$U12e80:4 = $U29000:4 & 0xff:4	
		$U12f00:1 = popcount($U12e80:4)	
		$U12f80:1 = $U12f00:1 & 1:1	
		PF = $U12f80:1 == 0:1	
0x100000e7b	jle 20 <_isPrime+0x25>	$Ud000:1 = OF != SF	
		$Ud100:1 = ZF || $Ud000:1	
		if $Ud100:1 goto 0x100000e95:8	

Fig. 1. First few instructions in the decompilation of nearest prime

To illustrate the Ghidra decompilation pipeline, we take a look at the first few instructions of an x86-64 binary. Figure 1 lists the decompilation result for the first few addresses of our example binary. For readability's sake, we have harmonized notation, and simplified in- and output notation. Address 0x100000e70 is the entry point of a function. Ghidra's machine code translator SLEIGH takes the x86-64 assembly instructions as listed in the second column in Intel syntax, and translates them to low P-Code, listed in the third column. The exact meaning of the P-Code instructions will be left for the coming sections, and is not essential to understand at this point. As for notation, the $U prefix indicates local variables. Registers are identified by their name, and are assumed to have a fixed size. Addresses are given in a hexadecimal format, prefixed by

0x. Constants are given either as a decimal or hexadecimal prefixed by 0x. Both addresses and constants have a size indicated by the number after the colon.

P-Code is considered low before decompilation, and is merely a one-to-many translation from assembly instructions to P-Code. The instruction set of P-Code is much smaller than x86, and SLEIGH basically breaks up a complicated assembly instruction into simple P-Code ones. For example, the x86 compare instruction at address 0x100000e77 breaks down into 13 separate P-Code instructions. To break down the instructions, P-Code uses local variables. The x86 `push rbp` instruction at address 0x100000e70 for example, is broken down into three simple assignments, using the variable $Uea00:8 to hold the original value of RBP temporarily, until it has been stored in memory.

After disassembly and translation, Ghidra performs several decompilation analyses. This results in high P-Code, listed in the third column, which uses an instruction set almost identical to low P-Code. The analyses Ghidra performs, remove all instructions that have no effect on execution, resolves the stack and constructs a control flow graph, among other things. A more detailed description of Ghidra's decompilation analyses can be found in other literature [4]. From the high P-Code, Ghidra constructs the final decompilation result in C code.

By example, we have shown how the basic Ghidra pipeline operates. Users can inspect the final decompilation result, but there also exists a plethora of scripts that can be run to further analyse the final or intermediate P-Code result. It is also possible to define custom scripts, through Ghidra's extensive API [10]. Since these analyses can target high P-Code directly, it is important to have a correct semantics for it.

3 Design Choices

As mentioned in Sect. 1, P-Code documentation is limited. On top of that, P-Code as-is cannot be given a formal semantics, because crucial information is not included. To arrive at an executable P-Code semantics, we had to make the following design choices.

Conditional Branches. We observed that the P-Code generated for conditional branches is often incorrect. Three different situations occur. One, the conditional branch is correct. Two, the conditional branch incorrectly jumps to the fall-through address in the True case. Three, the conditional branch should jump to a different address than the fall-through in the False case. The control flow API always returned the correct out addresses in our experiments. We use this fact later on when we build our interpreter.

Phi-nodes. P-Code has a MULTIEQUAL instruction, which is better known as a phi-node in literature [3]. A phi-node's value depends on the address of the previous block that was executed. In other SSA languages using phi-nodes, like LLVM IR [8], every alternative value is guarded by an address, which is compared to the address of the previous block. You compare the address to the alternatives in the phi-node, to know which value will be selected. This is not the case in P-Code. Here, only the alternative values are listed. Looking at the documentation, no additional information is provided about the ordering of the alternatives.

To determine which alternative belongs to which control flow, we have ran several experiments. From these experiments, we know that the order of the alternatives coincides with the inbound edge ordering when requesting these though Ghidra's control flow API.

Varnodes. Inputs and outputs are encoded by the so called varnodes. They represent the arguments and destination of the instructions. The P-Code manual states that varnodes are either a register or a memory location, and that they consist of three components: address space, offset and size.

We use a slightly different view of varnodes, and regard the "address space" as a varnode type. We have come across six different types, and they consist of two components: a value and a size. Of those six, we can bring it down to four essential types of varnodes.

- $(\mathbf{R}\, r, l)$ register, identified by a register address r and size l.
- $(\mathbf{A}\, a, l)$ memory, with a the starting address and l the size of the memory region.
- $(\mathbf{C}\, c, l)$ constant value, with c the value and l the size.
- (v, l) local variable, with v the identifier and l its size.

In the case where P-Code models a Harvard architecture binary instead of the more common von Neumann architecture, the memory varnode is split up into a data varnode and a code varnode, to model the dedicated addresses in this architecture.

- $(\mathbf{A_C}\, a, l)$ memory containing code, with a the starting address and l the size of the memory region.
- $(\mathbf{A_D}\, a, l)$ memory containing data, with a the starting address and l the size of the memory region.

The register notation differs from the previous P-Code example, as listed in Fig. 1. Instead of listing register names, an address in the register space is used, together with a size. Using register addresses and sizes has the advantage that we do not explicitly have to take care of register aliasing.

Call & Return. For the CALL instruction, P-Code documentation lists: "This instruction is semantically equivalent to the BRANCH instruction. (...) The P-Code instruction does not implement the full semantics of the call itself; it only implements the final branch."

Looking at P-Code programs, it is immediately clear that this cannot be true. A function can have arguments and return a value, and this behavior is certainly not captured by a basic BRANCH instruction. What is more, not all varnode address spaces will be in scope when dispatching a function call. Local variables are cleared when a new function context is entered. Arguments are passed through the registers, and these registers are cleaned up when P-Code returns from a call. How this is done exactly depends on the calling convention that the original binary relies on, more on this later.

For the indirect call and return, documentation lists a similar description. They are both said to be equivalent to BRANCHIND. For indirect call, this is false, for the reasons described above. For return, we see that this cannot hold for two reasons. One, in high P-Code, a return instruction can hold a return value that is to be the result of calling the function. Furthermore, no address to return to is included in the instruction; we do not have a branch destination.

Fall-through. The P-Code outputted by Ghidra performs fall-through in a non-uniform way. Looking at the P-Code output for several decompiled binaries, there is no pattern to be found in the way fall-through is performed. The most natural way to perform fall-through is to let control flow jump to the next block listed in the output, but this is not always true. It can occur that control flow jumps to a later or earlier block instead. The addresses of those blocks are not logical either. In some cases, the address of the fall-through block is higher than the current block, in some cases it is lower. We therefore do away with fall-through all-together, instead requiring that every block ends in a branching instruction.

User defined instructions. P-Code allows the use of the so called "USERDEFINED" instruction. The goal of this instruction is to capture very complicated behavior that cannot be described in terms of existing P-Code instructions. Since the semantics are user-provided, we consider this instruction to be out of scope for this paper.

Indirect instruction. One final tricky aspect of P-Code is the use of the indirect instruction. We will denote an indirect instruction as $\mathrm{out} = \mathrm{in}_0 \hookleftarrow \mathrm{in}_1$. The intuition of this instruction is that the value of in_0 should be assigned to out, but may be influenced indirectly by an instruction elsewhere, which is described by in_1. If there was an indirect influence, this means that potentially any value may be assigned to out. This can happen for example when calling external functions, so there is no way of knowing what the value will be.

4 P-Code Syntax

This section lists the syntax of P-Code programs. We base the syntax on the notation used in the P-Code documentation. P-Code features some instructions that do not have a defined syntax. For those instructions, we have taken the liberty of choosing an appropriate representation ourselves. On top of that, there are several minor changes that we needed to make, in accordance with the design choices from Sect. 3.

Figures 2 and 3 lists our P-Code syntax and P-Code operators. We model a program as a mapping from addresses a to code blocks b.

Inputs and outputs are modeled by four different varnodes. Registers, memory, constants and variables. For registers, addresses are used instead of names. To give an example, (0x18,8), (0x18,4) and (0x19,1) refer to RBX, EBX and BH respectively.

Program	
$\text{p} ::= a \mapsto b$	Mapping from address to block
Block	
$\text{b} ::= i;b \mid t$	Sequence, terminator
Instruction	
$\text{i} ::= \text{out} = (o \mid s) \mid s$	Assignment, call
$\mid\ {*}\,\text{out} = \text{in} \mid \text{out} = {*}\,\text{in}$	Store, load
$\mid\ \text{out} = \texttt{zext(}\,\text{in}\,\texttt{)} \mid \text{out} = \texttt{sext(}\,\text{in}\,\texttt{)}$	Zero-extend, sign-extend
$\mid\ \text{out} = \texttt{float2float}\,\text{in}$	Float size conversion
$\mid\ \text{out} = \texttt{trunc(}\,\text{in}\,\texttt{)}$	Float truncation
$\text{s} ::= \texttt{call}\ [\text{in}_0]\,\text{in}_1 \ldots \text{in}_n$	(Indirect) function call
Operation	
$\text{o} ::= \text{in} \mid \text{in}\,\texttt{((}c,s\texttt{))} \mid \ominus\,\text{in} \mid \text{in}_0 \oplus \text{in}_1$	Copy, subpiece, Unary, binary operation
$\mid\ \phi(a_0,\text{in}_0)\ldots(a_n,\text{in}_n) \mid \text{in}_0 \hookleftarrow \text{in}_1$	Phi node (multiequal), indirect
$\mid\ \text{in}_0 +_p \text{in}_1 \times_p \text{in}_2 \mid \text{in}_0 +_p \text{in}_1$	Pointer calculation, simple pointer calc
$\mid\ \text{out}((c_0,l_0),(c_1,l_0)) = \text{in}$	Bit insert
$\mid\ \text{out} = \text{in}((c_0,l_0),(c_1,l_0))$	Bit extract
Terminator	
$\text{t} ::= \texttt{goto}\,\text{in} \mid \texttt{if}\ \text{in}_0\ \texttt{goto}\ \text{in}_1\ \texttt{else}\ \text{in}_2$	(In)direct branch, conditional branch
$\mid\ \texttt{return}\ [\text{in}_0]\,\text{in}_1$	Function return
Varnode	
$\text{in,out} ::= (\mathrm{R}\,r,l) \mid (\mathrm{A}\,a,l) \mid (\mathrm{C}\,c,l) \mid (v,l)$	Register, ram, constant, variable
$a,r,c,l \in \mathbb{N}$	Address, register, constant, length
$v \in$ set of names	

Fig. 2. P-Code language syntax definition

A P-Code block must be non-empty, can have zero or more instructions i, and must end in a terminator instruction t. This does not adhere to documentation or Ghidra produced P-Code, but is required to correct the fall-through issues that occur, as mentioned in Sect. 3. We distinguish between regular instructions and terminator instructions, to make it easier to construct a semantics later on.

Instructions i are either basic operations on data, that are assigned to an output, or a function call that may or may not return some value. The instruction out $= (o \mid s)$ represents basic operations o that are assigned to the output out, or a function call s with a return value. In some cases, the data operation depends on the size of the output, like sign-extend. These instructions are added at this level for that reason.

A function call is denoted by `call` $[\text{in}_0]\,\text{in}_1 \ldots \text{in}_n$, where in_0 is either a constant, for direct calls, or a register, memory location or variable, which indicates that this is an indirect function call. This notation is identical to the P-Code documentation, but differs from the P-Code that Ghidra produces. Instead, Ghidra explicitly differentiates between direct and indirect function calls using different instructions, and uses a memory-varnode in the case of a direct function call, instead of a constant value.

Operators

$\oplus ::= \equiv \mid \neq \mid < \mid <_{s}$	Integer: equal, not equal, (signed) less
$\mid \leq \mid \leq_{s} \mid + \mid - \mid \veebar \mid \wedge \mid \vee$	(Signed) less or equal, add, subtract, xor, and, or
$\mid \ll \mid \gg \mid \gg_{s} \mid \times$	Left shift, (signed) right shift, times
$\mid \% \mid \%_{s} \mid \div \mid \div_{s} \mid \texttt{::}$	(Signed) remainder, (signed) divide, piece
$\mid \texttt{carry} \mid \texttt{carry}_{s} \mid \texttt{borrow}_{s}$	(Signed) overflow or carry, signed overflow or borrow
$\mid \veebar_{b} \mid \wedge_{b} \mid \vee_{b}$	Bool: exclusive-or, and, or
$\mid \equiv_{f} \mid \neq_{f} \mid <_{f} \mid \leq_{f} \mid +_{f}$	Float: equal, not equal, less, less or equal, plus
$\mid -_{f} \mid \times_{f} \mid \div_{f}$	Subtract, times, divide
$\ominus ::= \texttt{!} \mid - \mid \sim \mid -_{f}$	Bool negate, Int: sign negate, negate. Float: negate
$\mid \texttt{abs} \mid \sqrt{} \mid \texttt{nan}$	Absolute value, square root, NaN test
$\mid \sqcap \mid \sqcup \mid \texttt{round} \mid \texttt{int2float}$	Round to $+\infty$, $-\infty$, closest integer, int to float
$\mid \texttt{popcount} \mid \texttt{cast} \mid \texttt{new}$	Popcount, casting operation, allocate

Fig. 3. P-Code language operator syntax definition

Basic operations o are all data operations that are independent of the output. Most of them are straight forward binary or unary operations. We will discuss two non-standard operations.

The P-Code documentation does not list a syntax for the phi-node, so we have chosen one ourselves. We enhance the phi-node to include the address that guards the value alternative.

Indirect $(\mathrm{in}_0 \hookleftarrow \mathrm{in}_1)$ is the second non-standard operation. This instruction indicates that either the value of in_0 will be returned, or the value is unknown, because an instruction pointed to by in_1 has altered it. A typical use case of this is when an external function is called with a pointer.

Finally, we have terminator instructions. These instructions are not singled out by the P-Code documentation, but as mentioned, by separating them, constructing a semantics becomes easier.

The conditional branch is an interesting case. We have seen that Ghidra does not produces consistent P-Code for this instruction. To work around this issue, we replaced the conditional branch instruction with our own, so we can later use the Ghidra API to get a hold of the correct addresses, and explicitly state both the true and false branch. This works since conditional branches are terminators and can thus only appear at the end of a block.

Lastly, the return instruction. This instruction includes an offset, and may hold a value to be returned. Looking at large pieces of P-Code, we found that the offset is not actually used in performing the return. For completeness sake we include this parameter in the syntax, but we will not give it any semantics. The same is true for the `cast` and `new` unary operators, these are merely placeholders to indicate that a value has been cast or new memory has been allocated. This information can then be used by subsequent analyses.

For space reasons, we omit a description of all basic operations, as well as unary and binary operators.

5 P-Code Semantics

This section presents a big-step semantics for P-Code. As described in the previous sections, defining a semantics for P-Code is not trivial. Based on our experiments and observations so far, we assume that the following holds for programs written in P-Code.

Local variables do not overlap. We assume that local variables do not overlap in the local variable address space. In other words, local variables occupy separate memory locations. This property of P-Code has been verified by decompiling large binaries and checking that no variables overlap.

No global variables. We assume that all declared variables are local. Strictly speaking, the P-Code documentation does not prohibit an address space that serves as global variables, but after running several experiments, this behaviour has not been observed. Programs can and do have global variables, but they are confined to memory and registers.

Call and Branch on constant. We assume that direct calls and direct branches are encoded by having a constant varnode as an argument. In all other cases, the call and branch will be interpreted as indirect.

Terminators only at block's end. We assume that terminator instructions like branch and return only occur at the end of a block. This assumption already shows up in the syntax listed in the previous section, but for completeness sake, we reiterate this fact here.

$\sigma = (\mathcal{M}, \mathcal{R}, \mathcal{V})$	State, containing the address of the previous block, the current block, memory, registers and variables
$\mathcal{M} = (\mathrm{A}\, a, l) \mapsto (\mathrm{C}\, c, l)$	Memory
$\mathcal{R} = (\mathrm{R}\, r, l) \mapsto (\mathrm{C}\, c, l)$	Register mapping
$\mathcal{V} = v \mapsto (\mathrm{C}\, c, l)$	Variable mapping

Fig. 4. Semantic objects

Figure 4 lists the semantic objects needed for evaluation; state, memory, registers and variable mapping. The memory mapping $\mathcal{M}$ takes an address a and size l and returns a constant varnode $(\mathrm{C}c, l)$. The register mapping $\mathcal{R}$ takes a register address r and a size l and returns a constant varnode $(\mathrm{C}c, l)$. Just a register identifier is not sufficient, since registers can alias. Three special registers are used to keep track of function return value, the address of the previous block and the address of the current block. These registers are denoted by "Ret", "Prev" and "Cur", assuming that a varnode representation that is separate from registers used by the program exists. Variable mapping $\mathcal{V}$ takes a variable identifier v and returns a constant varnode $(\mathrm{C}c, l)$. Figures 7 through 11 list the rules for the different semantic judgements we use. These rules use two auxiliary judgements, namely the evaluation of varnodes and state update, listed in Fig. 5 and 6 respectively.

$$\text{V-MEM } (\mathcal{M},\mathcal{R},\mathcal{V}),(\mathrm{A}\,a,l)\downarrow \mathcal{M}[(\mathrm{A}\,a,l)] \qquad \text{V-REG } (\mathcal{M},\mathcal{R},\mathcal{V}),(\mathrm{R}\,r,l)\downarrow \mathcal{R}[(\mathrm{R}\,r,l)]$$

$$\text{V-CONST } \sigma,(\mathrm{C}\,c,l)\downarrow(\mathrm{C}\,c,l) \qquad \text{V-UNIQUE } (\mathcal{M},\mathcal{R},\mathcal{V}),(v,l)\downarrow(\mathcal{V}[v],l)$$

Fig. 5. Evaluation of varnodes to values

$$\text{U-REG}\ \frac{(\mathcal{M},\mathcal{R},\mathcal{V}),(x,l_2)\downarrow \mathrm{val} \qquad l_1\equiv l_2 \qquad \mathcal{R}'=\mathcal{R}[(\mathrm{R}\,r,l_1)\mapsto \mathrm{val}]}{(\mathcal{M},\mathcal{R},\mathcal{V}),(\mathrm{R}\,r,l_1),(x,l_2)\uparrow(\mathcal{M},\mathcal{R}',\mathcal{V})}$$

$$\text{U-MEM}\ \frac{(\mathcal{M},\mathcal{R},\mathcal{V}),(x,l_2)\downarrow \mathrm{val} \qquad l_1\equiv l_2 \qquad \mathcal{M}'=\mathcal{M}[(\mathrm{A}\,a,l_1)\mapsto \mathrm{val}]}{(\mathcal{M},\mathcal{R},\mathcal{V}),(\mathrm{A}\,a,l_1),(x,l_2)\uparrow(\mathcal{M}',\mathcal{R},\mathcal{V})}$$

$$\text{U-VAR}\ \frac{(\mathcal{M},\mathcal{R},\mathcal{V}),(x,l_2)\downarrow \mathrm{val} \qquad l_1\equiv l_2}{(\mathcal{M},\mathcal{R},\mathcal{V}),(v,l_1),(x,l_2)\uparrow(\mathcal{M},\mathcal{R},\mathcal{V}[v\mapsto \mathrm{val}])}$$

Fig. 6. State update semantics

Varnodes are evaluated by judgements of the form $\sigma, \mathrm{in} \downarrow \mathrm{val}$, taking a state σ and varnode in, and returning the resulting value val, as listed in Fig. 5. The resulting value is again a varnode, of the form $(\mathrm{C}c, l)$, where c is a constant and l the size. Depending on the type of the varnode, this value is retrieved from memory, register mapping or variable mapping.

Updates to memory are handled by judgements of the form $\sigma, \mathrm{out}, \mathrm{in} \uparrow \sigma'$, taking a state σ, destination out and source in, returning the updated state σ', as listed in Fig. 6. The type of the destination is used to select the correct rule, and ultimately which part of the state to update. In general, P-Code always requires that destination and source size is equal. This is ensured by the update semantics.

At the top-level, we evaluate a block using the judgement $p, \sigma, b \longrightarrow_b \sigma'$, which takes a program p, state σ and entry block b and returns a new state σ', which is the result of completely executing the program. Figure 7 lists the semantic rules for block evaluation.

B-Seq first evaluates the instruction i, and uses the resulting state to evaluate the remainder of the block, b.

B-Term evaluates a block ending in a terminator, using the terminator semantics.

The terminator semantics is given in Fig. 8. It uses judgements of the form $p, \sigma, t \longrightarrow_t \sigma'$, taking a program p, state σ and terminator t as input and producing a resulting state σ'.

The terminator semantics is pretty straight forward. T-Branch evaluates its varnode in and looks up the next block in p, and then use the block semantics to evaluate it. These rules make use of the varnode evaluation semantics $\downarrow$ listed in Fig. 5. In the case of a conditional branch, in_0 is evaluated. If the condition returns decimal 1, of any size, we go to the true-branch. All other return values are regarded as false. Branching instructions also perform some bookkeeping on what the current and previous block addresses are.

$$\textsc{B-Seq}\ \dfrac{p, \sigma, i \longrightarrow_i \sigma' \qquad p, \sigma', b \longrightarrow_b \sigma''}{p, \sigma, i\,;b \longrightarrow_b \sigma''} \qquad \textsc{B-Term}\ \dfrac{p, \sigma, t \longrightarrow_t \sigma'}{p, \sigma, t \longrightarrow_b \sigma'}$$

Fig. 7. Block evaluation semantics

$$\textsc{T-CBranch-T}\ \dfrac{\begin{array}{c}(\mathcal{M}, \mathcal{R}, \mathcal{V}), \mathrm{in}_0 \downarrow (\mathrm{C}\,1, l_0) \\ \mathcal{R}' = \mathcal{R}[\mathrm{Prev} \mapsto \mathcal{R}(\mathrm{Cur}), \mathrm{Cur} \mapsto (\mathrm{C}\,a_1, l_1)] \\ p, (\mathcal{M}, \mathcal{R}', \mathcal{V}), p(a_1) \longrightarrow_b \sigma'\end{array}}{p, (\mathcal{M}, \mathcal{R}, \mathcal{V}), \texttt{if}\ \mathrm{in}_0\ \texttt{goto}\ (a_1, l_1)\ \texttt{else}\ (a_2, l_2) \longrightarrow_t \sigma'}$$

$$\textsc{T-CBranch-F}\ \dfrac{\begin{array}{c}(\mathcal{M}, \mathcal{R}, \mathcal{V}), \mathrm{in}_0 \downarrow (\mathrm{C}\,c, l_0) \\ c \neq 1 \qquad \mathcal{R}' = \mathcal{R}[\mathrm{Prev} \mapsto \mathcal{R}(\mathrm{Cur}), \mathrm{Cur} \mapsto (\mathrm{C}\,a_2, l_2)] \\ p, (\mathcal{M}, \mathcal{R}, \mathcal{V}), p(a_2) \longrightarrow_b \sigma'\end{array}}{p, (\mathcal{M}, \mathcal{R}', \mathcal{V}), \texttt{if}\ \mathrm{in}_0\ \texttt{goto}\ (a_1, l_1)\ \texttt{else}\ (a_2, l_2) \longrightarrow_t \sigma'}$$

$$\textsc{T-Branch}\ \dfrac{\begin{array}{c}(\mathcal{M}, \mathcal{R}, \mathcal{V}), \mathrm{in} \downarrow (\mathrm{C}\,c, l) \\ \mathcal{R}' = \mathcal{R}[\mathrm{Prev} \mapsto \mathcal{R}(\mathrm{Cur}), \mathrm{Cur} \mapsto (\mathrm{C}\,c, l_1)] \\ p, (\mathcal{M}, \mathcal{R}, \mathcal{V}), p(c) \longrightarrow_b \sigma'\end{array}}{p, (\mathcal{M}, \mathcal{R}, \mathcal{V}), \texttt{goto}\,\mathrm{in} \longrightarrow_t \sigma'} \qquad \begin{array}{l}\textsc{T-Return} \\ \dfrac{\sigma, (\mathrm{Ret}, s), (x, s) \uparrow \sigma'}{p, \sigma, \texttt{return}\ [\mathrm{in}](x, s) \longrightarrow_t \sigma'}\end{array}$$

Fig. 8. Terminator evaluation semantics

T-Return handles a return statement. To pass the return value to the caller, we use the Ret register. Updating the state as such is taken care of by the memory update function ↑.

Figure 9 lists the partial instruction semantics. Judgements have the form $p, \sigma, i \longrightarrow_i \sigma'$, taking a program p, state σ and instruction i, and returning the resulting state σ'. For space reasons, we only list I-Assign, I-AssCall, I-Store and I-Load here.

I-Assign uses the operation semantics to evaluate o, which returns the value that should be assigned to out. The output out is updated by ↑, depending on the type of varnode that out is; memory, register or variable.

I-AssCall relies on the call semantics to perform the call, and the resulting value is again used to update out, and the final state is returned.

I-Store evaluates the destination, converts it to a varnode of type address, and updates accordingly.

I-Load evaluates the input and then treats it as a memory address, and looks up the final value in memory. The number of bytes to be retrieved from memory is dictated by the output varnode.

The calling semantics is one of the more perculiar aspects of P-Code. All arguments that are passed though registers in the original binary, are now passed

$$\textsc{I-Assign}\quad \frac{\sigma, o \longrightarrow_o \text{val} \qquad \sigma, \text{out}, \text{val} \uparrow \sigma'}{p, \sigma, \text{out} = o \longrightarrow_i \sigma'} \qquad \textsc{I-AssCall}\quad \frac{p, \sigma, s \longrightarrow_s \sigma', \text{val} \qquad \sigma', \text{out}, \text{val} \uparrow \sigma''}{p, \sigma, \text{out} = s \longrightarrow_i \sigma''}$$

$$\textsc{I-Store}\quad \frac{\sigma, \text{out} \downarrow (\mathrm{C}\, c, l) \qquad \sigma, (\mathrm{A}\, c, l), \text{in} \uparrow \sigma'}{p, \sigma, *\,\text{out} = \text{in} \longrightarrow_i \sigma'} \qquad \textsc{I-Load}\quad \frac{\begin{array}{c}\sigma, \text{in} \downarrow (\mathrm{A}\, a, l') \qquad \sigma, (\mathrm{A}\, a, l) \downarrow \text{val}' \\ \sigma, (x, l), \text{val}' \uparrow \sigma'\end{array}}{p, \sigma, (x, l) = *\,\text{in} \longrightarrow_i \sigma'}$$

Fig. 9. Partial instruction evaluation semantics

$$\textsc{Call-AMD64-ABI}\quad \frac{\begin{array}{c}(\mathcal{M}, \mathcal{R}, \mathcal{V}), \text{in} \downarrow (\mathrm{C}\, c, l) \qquad (\mathcal{M}, \mathcal{R}, \mathcal{V}), \text{in}_i \downarrow (\mathrm{C}\, c_i, l_i) \\ \mathcal{R}' = \mathcal{R}\left[\begin{array}{c}(0x38,l_0)\mapsto(\mathrm{C}\, c_0,l_0),(0x30,l_1)\mapsto(\mathrm{C}\, c_1,l_1),(0x10,l_2)\mapsto(\mathrm{C}\, c_2,l_2),(0x8,l_3)\mapsto(\mathrm{C}\, c_3,l_3) \\ ,(0x80,s_4)\mapsto(\mathrm{C}\, c_4,l_4),(0x88,l_5)\mapsto(\mathrm{C}\, c_5,l_5),\mathrm{Prev}\mapsto\mathcal{R}(\mathrm{Cur}),\mathrm{Cur}\mapsto(\mathrm{C}\, c,l)\end{array}\right] \\ p, (\mathcal{M}, \mathcal{R}', \emptyset), p(c) \longrightarrow_b \mathcal{M}', \mathcal{R}'', \mathcal{V}') \\ \mathcal{R}''' = \mathcal{R}''\left[\begin{array}{c}(0x38,l_0)\mapsto\mathcal{R}[(0x38,l_0)],(0x30,l_1)\mapsto\mathcal{R}[(0x30,l_1)],(0x10,l_2)\mapsto\mathcal{R}[(0x10,l_2)],(0x8,l_3)\mapsto\mathcal{R}[(0x8,l_3)] \\ ,(0x80,l_4)\mapsto\mathcal{R}[(0x80,l_4)],(0x88,l_5)\mapsto\mathcal{R}[(0x88,l_5)],\mathrm{Prev}\mapsto\mathcal{R}(\mathrm{Prev}),\mathrm{Cur}\mapsto\mathcal{R}(\mathrm{Cur})\end{array}\right] \\ \sigma = (\mathcal{M}', \mathcal{R}''', \mathcal{V})\end{array}}{p, (\mathcal{M}, \mathcal{R}, \mathcal{V}), \texttt{call}\ [\text{in}]\ \text{in}_0\ \text{in}_1\ \text{in}_2\ \text{in}_3\ \text{in}_4\ \text{in}_5 \longrightarrow_s \sigma, \mathcal{R}''[\text{Return}]}$$

Fig. 10. Example of a function call evaluation rule

as function arguments in the call instruction. However, the called function does still retrieve them from registers. How this is done precicely depends on the original calling convention. Figure 10 lists an example of a call rule for a binary that uses the AMD64-ABI calling convention.

Judgements are of the form $p, \sigma, s \longrightarrow_s \sigma', \text{val}$, taking a program p, state σ, call statement s and returning the resulting state σ' and value val. In this case, a call can have at most six arguments, adhering to the specific calling convention.

The semantics completely deviates from the P-Code documentation. Section 3 discusses this issue, here we stick to a description of the semantics.

The Call-rule first resolves the address of the function to be called. As mentioned in Sect. 4, if in is a constant, we have a direct call. In all other cases, we have an indirect call, and the varnode evaluation semantics takes care of resolving the address. We evaluate all arguments to the call, which are then assigned to the appropriate registers. To execute the called function, the block semantics is used. We look up the block, and evaluate it under the current memory, the registers containing the arguments, and an empty local variable mapping $\emptyset$. Returning from the call, the local variable mapping is disregarded, registers are cleaned up, and the return value is retrieved from the registers. This value, along with the new state are then returned.

Figure 11 lists a few of the operation evaluation rules. Judgements are of the form $\sigma, o \longrightarrow_o \text{val}$, taking a state σ and operation o, returning the resulting value

$$\textsc{O-Copy}\ \frac{\sigma, \mathrm{in} \downarrow \mathrm{val}}{\sigma, \mathrm{in} \longrightarrow_o \mathrm{val}} \qquad \textsc{O-Phi}\ \frac{a_i \equiv \mathcal{R}(\mathrm{Prev}) \qquad (\mathcal{M}, \mathcal{R}, \mathcal{V}), \mathrm{in}_i \downarrow \mathrm{val}}{(\mathcal{M}, \mathcal{R}, \mathcal{V}), \phi(a_1, \mathrm{in}_1) \ldots (a_n, \mathrm{in}_n) \longrightarrow_o \mathrm{val}}$$

$$\textsc{O-Indirect-val}\ \frac{\sigma, \mathrm{in}_0 \downarrow \mathrm{val}}{\sigma, \mathrm{in}_0 \hookleftarrow \mathrm{in}_1 \longrightarrow_o \mathrm{val}} \qquad \textsc{O-Indirect-ND}\ \frac{c, l \in \mathbb{N}}{\sigma, \mathrm{in}_0 \hookleftarrow \mathrm{in}_1 \longrightarrow_o (\mathrm{C}\, c, l)}$$

Fig. 11. Partial operation evaluation semantics

val. For space reasons, we omit all basic operations, as well as unary and binary operations. These operations are all straight-forward and standard.

O-Copy evaluates the varnode and return its value.

O-phi evaluates the phi-node by finding the address a_i that is equal to the address of the last block. The selected input is evaluated and its value returned.

O-Indirect-val and -ND rules handle the indirect instruction. Here, we basically have two options. Either the value is unaffected, and we can return it, or the instruction pointed to by the second argument has altered the first in some way, in which case any value of any size can be returned.

The above syntax and semantics assume that the P-Code models the von Neumann architecture. Extending them to deal with Harvard architecture that has dedicated code and data memory sections is straight-forward, but omitted from the description here for space reasons.

5.1 P-Code Interpreter

To validate that the semantics above are executable, we have built a P-Code interpreter in Haskell. The source code is publicly available, and consists of a parser, type definitions and the interpreter itself[3]. Ghidra does not come with a script to dump P-Code, so we have created a script with that functionality[4]. The interpreter is intended to be used in combination with this Ghidra script, since it corrects the P-Code output of Ghidra with respect to the conditional branch, fall-through and phi-nodes.

We encountered several interesting issues in order to get to a working interpreter. First of all, we had to bridge the gap between what we think the syntax and semantics of P-Code should be, ideally, and what Ghidra actually produces for us. We assumed that both call and indirect call use the same instruction, and we merged the branch and indirect branch instruction. In the P-Code produced by Ghidra, these are all separate instructions. The direct call and direct branch use a memory varnode where we prefer to use a constant. For the MULTI-EQUAL (phi-node), conditional branch instruction and the fall-through mechanism, Ghidra's output does not contain enough information to come to an executable semantics, as described above. We augment the P-Code dumping

[3] https://github.com/niconaus/pcode-interpreter.

[4] https://github.com/niconaus/PCode-Dump.

script to include the additional information required. Second, we assumed several properties to hold for P-Code programs, as outlined in the beginning of this section.

Finally, it is important to note that although the semantics is executable, it is not practical to do so. Any realistic program will produce many INDIRECT instructions, which introduces non-determinism into the program. Execution of P-Code containing these instructions will therefore return many different alternative outcomes, and may be propagating unknown values, not returning a meaningful result. The point of the P-Code interpreter is merely to validate the property of our semantics that it is in fact executable. In our interpreter, we have chosen to regard INDIRECT instructions as deterministic, assuming that their value has not been changed by the indicated side-effect.

6 Changes to Ghidra and P-Code

Based on our findings, we recommend the following changes to be made to Ghidra, P-Code and its documentation.

6.1 P-Code

Currently, the phi-node, or MULTIEQUAL as P-Code calls it, is incomplete. It only contains a list of alternative values, but not the control flow address that guards it. We suggest to adopt the definition introduced in Sect. 4, which includes both the address of the previous block and the value associated.

6.2 Ghidra/SLEIGH

We recommend the following changes to be made to Ghidra and its machine code translator SLEIGH.

CBRANCH Our experiments show that Ghidra can return erroneous destinations for the CBRANCH instruction. It does have the correct information available, as we have validated by requesting the true-branch and false-branch destination via the Ghidra API instead of P-Code. It is clear that there is a bug in the way Ghidra produces P-Code. As mentioned before, one of three situations occur. The conditional branch is correct, and so is the fall-through. The conditional branch is incorrect, either the true-address or the fall-through branches to the wrong address.

Fall-through The P-Code outputted by Ghidra performs fall-through in a non-standard way in certain cases. For assembly languages, the fall-through address is the next instruction listed, or in our case, the next block listed. The P-Code that Ghidra returns sometimes breaks with this standard, and fall-through goes to the next block listed, which might have a completely different address, or in some rare cases, to a completely different block all together.

P-Code rendering The P-Code displayed in Ghidra's GUI uses a different syntax than the one given in the P-Code documentation. The low P-Code in the listing view uses the capital letters notation, where tools like the graph AST do use the regular syntax. Readability would be greatly improved if the same, preferably the regular, syntax is used.

6.3 Documentation

The P-Code documentation included with Ghidra has not been updated for several years. We suggest to make the following changes to greatly improve the quality of the documentation, both in correctness and completeness.

High and Low P-Code The P-Code Reference Manual attempts to cover both low and high P-Code with one description for each instruction, and then tagging on extra information for the high P-Code case. We recommend splitting up documentation in high and low P-Code.

Varnodes In documentation, varnodes are described as containing three elements: address space, address and size. From our experiments, we see that this view does not work in practice. Constants are also encoded as varnodes, where the address is used as a constant value instead. When performing a call (also see below), some address spaces are preserved, some are reset for the scope of the call. In this paper we have used the address space field as the type of the varnode, and this seems to be a better fit. We suggest one of two things to be done. One, this view is adopted by documentation, including a list of the different types of varnodes and how they behave in for example a function call. Or two, the CALL and RETURN instructions are updated to include address space scoping.

CALL, CALLIND, RETURN As mentioned in Sect. 3, documentation states that CALL, CALLIND and RETURN are equivalent to BRANCH, BRANCHIND and BRANCHIND respectively. From our experimental results, we see that this is not the case. Function arguments are transferred via registers, local variables are reset, a value can be returned, and after a call, register cleanup is performed and local variables restored. We don't see an issue with these instructions themselves, more with the way they are explained. This also ties in with the first point made on the difference between low and high P-Code. Call, indirect call and return behave completely different in low and high P-Code, and deserve a better documentation.

Small inconsistencies The documentation contains many small inconsistencies which should be cleared up. For example, the syntax reference introduces two different notations for SUBPIECE that are not in the P-Code Operation reference, and that we have not found in our experiments.

6.4 Response from Ghidra Developers

We have reached out to the Ghidra development team at NSA with our findings and the above recommendations. They have confirmed our findings and acknowledged all issues we found. As for the conditional branch, they refer to a GitHub issue where this problem is also identified[5]. Their stance is that although it is semantically incorrect, they do not consider this to be a bug. The destination of the conditional jump is preserved from low to high P-Code, which they deem more important than the correctness of the instruction itself.

7 Related Work

Research that makes use of Ghidra's results is scarce, due to the fact that Ghidra has only been publicly available for a few years. Below is a survey of several interesting projects that use Ghidra and P-Code to perform program analysis.

GhiHorn is an SMT based path analysis tool that uses Ghidra and P-Code [6]. Their goal is to determine if a path exists to a certain program point, and how the program should be instantiated to reach this point. Their approach relies on Ghidra's control flow API to construct flow from block to block. For the individual blocks, they use a custom made transformer from P-Code to Z3 expressions. The documentation provided is limited, and GhiHorn does not seem to deal with the more intricate details of P-Code, such as phi-nodes, indirect and the call/return mechanism. It would be interesting to see where this approach leads in the future, when the tool matures further.

A recent master thesis describes work on decompiling binaries into LLVM IR using Ghidra [14]. The binary is loaded into Ghidra, and the decompiled P-Code is then translated to LLVM IR. Although this work does not provide a semantics for P-Code, it does relate LLVM IR's semantics to that of P-Code. We've looked though the source code for this project, and compared the relational semantics to our P-Code semantics. In most cases, translation to LLVM IR seems to be a more straight-forward affair, since issues like the call/return mechanism carry over directly. One big limitation of this work is again that more difficult P-Code concepts like phi-nodes, the nondeterministic indirect instruction, floating point operations and pointer calculations are not supported. Looking at the translation for conditional branches, we see that this work is susceptible to the error that we discovered in Ghidra.

Ghidra has also been used to develop a static taint analysis [2]. The author uses external lists containing sources and sinks, and uses a taint policy that defines how a taint is introduced and propagated. Unfortunately, source code for this project is no longer available. The group is working on a new version and has pulled the code in the mean time. It would have been very interesting to see what P-Code semantics they employ.

A caveat of all of these approaches is the fact that none of them do any kind of verification or have any formal theory on their approaches. As we have seen from our experiments, the P-Code semantics is not straight-forward. Having a formal semantics has the potential to improve these and future efforts on decompilation and binary analysis.

[5] https://github.com/NationalSecurityAgency/ghidra/issues/2736.

8 Conclusion

We have presented a formal semantics for Ghidra's P-Code. By developing this semantics, we have uncovered several undocumented properties of P-Code, as well as some inconsistencies and one serious bug in the way that Ghidra builds the conditional branch instruction. To arrive at an executable P-Code semantics, we have made several extensions to the language. We have validated that our semantics is executable by building an interpreter for P-Code in Haskell. The semantics and issues described have been acknowledged by the NSA. We have performed a survey of binary analysis projects that leverage Ghidra and P-Code, and have seen several that are directly affected by the issues we uncovered.

Acknowledgements. We would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped to greatly improve the paper.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under contract N6600121C4028 and Agreement No. HR.00112090028, and the US Office of Naval Research (ONR) under grant N00014-17-1-2297.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific, or ONR.

References

1. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: a binary analysis platform. In: Computer Aided Verification–23rd International Conference, CAV 2011, Snowbird, UT, USA, 14–20 July 2011. Proceedings, pp. 463–469 (2011)
2. Cole, E.: Static taint analysis of binary executables using architecture-neutral intermediate representation (2019)
3. 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 (1991)
4. Eagle, C., Nance, K.: The Ghidra Book: The Definitive Guide. No Starch Press, California (2020)
5. Shaila, S., Darki, A., Faloutsos, M., Abu-Ghazaleh, N., Sridharan, M.: Disco: combining disassemblers for improved performance. In: RAID 2021: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, 6–8 October 2021, pp. 148–161 (2021)
6. Gennari, J.: Ghihorn: Path analysis in Ghidra using smt solvers. Carnegie Mellon University's Software Engineering Institute Blog, 18 October 2021. http://insights.sei.cmu.edu/blog/ghihorn-path-analysis-in-ghidra-using-smt-solvers/
7. Hex-Rays, S.: Ida pro disassembler (2022)
8. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 20–24 March 2004, San Jose, CA, USA, pp. 75–88 (2004)
9. National Security Agency: P-Code Reference Manual, September 2019
10. National Security Agency: Ghidra API help (2021)

11. PNF Software: Jeb decompiler (2022). https://www.pnfsoftware.com
12. Radare org: Radare2 (2022). https://github.com/radareorg/radare2
13. Shoshitaishvili, Y., et al.: SOK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, 22–26 May 2016, pp. 138–157. IEEE Computer Society (2016)
14. Toor, T.: Decompilation of Binaries into LLVM IR for Automated Analysis. Master's thesis, University of Waterloo (2022)