AALBORG UNIVERSITET

ds104f19:

Assembler typesystem

AALBORG UNIVERSITY
STUDENT REPORT

Software
Aalborg University
http://www.aau.dk

Title:
Assembler typesystem

Theme:
Security

Project Period:
Fall Semester 2019

Project Group:
ds104f19

Participant(s):
Kenneth Husum Stick

Supervisor(s):
René Rydhof Hansen

Copies: 0

Number of Pages: 18

Date of Completion:
February 26, 2020

Abstract:

This rapport propose a syntax, semantic and typesystem for x86-64, with discussion of how to handle problems that arice from reuse of registers. How to handle the memory, the stack and registers is also discused.

Part I

Introduction

1. Introduction

The intention with this report is to make a system that can add type information to binary files so they become easier to reverse engineer, this is done by making a syntax, semantic and typesystem for a subset of the x86-64 instruction set. As there is no types in assembler a new set of types have to be made both to fix the information that is available in the binary and what types that would be preferable when reverse engineering. The analysis will have to be context aware as x86-64 only implement the core language and leave other implementation details vague so they are not uniform over different systems.

1.1 Resume

The previous rapport proposed a system to add type information to x86-64 by making a system that added type constraints too the language. The report looked into how unix device drivers worked, with focus on how that information can be used to restore the type information that is lost when compiling. If knowledge of types were restored there would be fewer binary bloobs that contains data. The focus on driver was chosen as their are an integral part of model operation systems, where the plug and play approach gives new security challenges, drivers for most devices have to be available for this system to work, but any errors in these can be abused by adversary's.
Not all drivers make their source code available for external security analysis, this leaves only the binary files. To reverse engineer any program require a time as most of the program have to be analysed manually, any tools that can do some of this analysis make the entire process faster. The first part of this process is to make a type system that is both general enough to fit with the typeless x86-64 and specific enough to give more information, as most drivers are written in c or c++ these languages was chosen as the base for typesystem, the type hierarchy 1.1 show how types interact with each other an arrow a1 -> a2 means that in every case where a2 can be used a1 can also be used, this is show best by $\top$ as it can be used in all cases.
A system for type inference was proposed where every instruction would add a set of constrains, based on the knowledge that was available, all constraints would be coalesced into a set of constraints on the programs, that when solved would give a upper and lower bound on the type of every variable, this approach had a problem with scoping as register dont work like variables in other programming languages that often only hold one type, register will be over written several times over the course of a execution.
How information would be propagated through function call was looked at, as the program state when a function is called is that state that is handed over to the function, with the exception of setting up a new stack frame, it would just be needed to keep the program state and not do anything else.
Functions can be seen as inline code this approach will not require any additional work, but if a function is use several times it will be analysed many times, this problem can be countered by making a function summary, where the function is only analysed once and sum down to a single constrain. If done manually external well defined function would not be analysed but the know functions summary will be used saving computation time, this could also allow for custom function summary's so if there is a common function that is use several time by the binary it can be manually analysed and a function summary can be made.

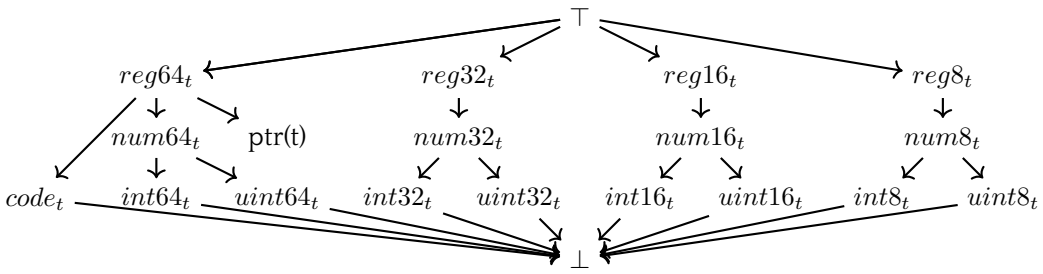

Figure 1.1: Subtypes

2. Language

2.1 Drivers

Drivers is use as an abstraction of the functionality of hardware, it can either be written as a specialised driver or a general driver. An example of a general driver is the VMMOUSE driver in unix, it can be used by most mouse, an example of a specialised driver is the NVIDIA graphic drivers they either work of a single model or a family of models. Drivers can either run the kernel mode or user mode, if it is possible user mode is recommended as errors here can be handled much easier then if they run in kernel mode, some driver need functionality from kernel mode and therefor is needed run as kernel module. Kernel modules is part of the kernel that can be loaded and unloaded at will, before kernel modules all drivers had to be compiled into the kernel, if a new device needed to be added the kernel need to be recompiled with that driver. Kernel modules added usability to consumer computers, but as the kernel can be modified at run time it also added security risks. With kernel modules operation system got a list of drivers and what devices they support, every time a new device is added, a search is done through this list for a driver that support this device, when one is found it is loaded into the kernel. The unix kernel got strict rules about how code can be added into it, this make sure several people have looked at the code and tested it before it can become a part of the kernel, but drivers does not necessarily undergo the same rigorous process.

2.2 Example

```
1    call    __fentry__          ; PIC mode
2    push    rbp
3    mov     rdx, offset aBcm203x ; "bcm203x"
4    mov     rsi, offset __this_module
5    mov     rdi, offset bcm203x_driver
6    mov     rbp, rsp
7    call    usb_register_driver ; PIC mode
8    pop     rbp
9    retn
```

Listing 2.2 show the registration function of the bcm203x driver. The function start by calling fentry, this is used by the ftrace tool, this happens if the binary was compiled with the -pg -mfentry flags, this function does not have an impact on how the function runs and will be ignored. First the base pointer is pushed to the stack to prepare to call the usb_register_driver function, the 3 addresses is moved into the appropriate registers, that being rdx, rsi and rdi these are the third second and first argument as can be seen in Table 2.1. mov rbp, rsp set up the base pointer for the function call that happen as the next instruction. After the function call the base pointer is restored and the function return.

2.3 Syntax

2.3.1 Language

All values in this language is encoded as a 64 bit value. Values cover both addresses and intermediate values used by instructions. It is not possible for all

Regiset	Common use
RAX	Caller saved
RBX	Callee saved
RCX	Fourth argument
RDX	Third argument
RSI	Second argument
RDI	First argument
RSP	Register stack pointer
RBP	Register base pointer
R8	Fifth argument
R9	Sixth argument
R10	Caller saved
R11	Caller saved
R12	Callee saved
R13	Callee saved
R14	Callee saved
R15	Callee saved

Table 2.1: Register common use

instruction to know if a value is an address or an intermediate value without context from other instructions.

$$val = \mathbb{B}^{64} \text{ where } \mathbb{B} = \{0,1\}$$

x86-64 defines 17 registers, 16 of witch is general purpose registers, and one for the program counter. Some of the general purpose registers is used for special tasks like RSP is used to store the stack pointer, these common uses can be seen in Table 2.1. The program pointer RPC can be read from like all other registers but can not be written to directly by instructions like mov.

$$GPR = \{RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, R10, R11, R12, R13, R14, R15\}$$
$$ControlRegister = \{RPC\}$$
$$Register = GPR \cup ControlRegister$$

Assembler does not define how registers should be used, so for programs to be able to interface with librarys call conventions are used to define how registers should be handled before and after a function call. Currently the two most used call conventions are Microsoft x64 calling convention and System V AMD64 ABI, Windows systems use the Microsoft calling convention, while unix and mac use the System V. Call convention define how arguments are saved but also it is the caller or the callee responsible to make sure the value of a register is saved. This report will be looking at unix drivers and therefor is only looking at the System V AMD64 ABI, a list of the rules for every register can be seen at Table 2.1

Registers is a complete function from a register to a value.

$$GPRS = GPR \to val$$
$$ControlRegisters = ControlRegister \to val$$
$$Registers = Register \to val$$

Bit	Lable	Description
0	CF	Carry Flag
1	1	Reserved
2	PF	Parity Flag
3	0	Reserved
4	AF	Auxillary Carry Flag
5	0	Reserved
6	ZF	Zero Flag
7	SF	Sign Flag
8	TF	Trap Flag
9	IF	Interrupt Enable Flag
10	DF	Direction Flag
11	OF	Overflow Flag
12-13	IOPL	I/O Privilege Level
14	NT	Nested Task
15	0	Reserved
16	RF	Resume Flag
17	VM	Virtual-8086 Mode
18	AC	Alignment Check
19	VIF	Virtual Interupt Flag
20	VIP	Virtual Interupt Pending
21	ID	ID Flag
22-63	0	Reserved

Table 2.2: x86-64 Flags

2.3.2 flags

x86-64 use one register for flags, as registers are 64 bit it can have up to 64 flags. most of the bits are not used for flags but are assigned a value of 0 or 1 that it have to have. Table 2.2 shows the the bits in the flag register and what they are used for.

Not all flags are used for control flow of basic programs and have been cut from the final semantic, as to keep it cleaner. These will have to be reintroduced if thing like debugger, nested tasks and virtual 8086.

Flag = {CF, PF, AF, ZF, SF, TF, IF, DF, OF} $Flags = Flag \to \mathbb{B}$

2.3.3 Formalisation

Here the syntax for the language is shown.

$$Inst ::= mov\, gpr\, r \mid mov\, gpr\, i \mid push\, r \mid push\, i \mid pop\, gpr \mid call\, r \mid call\, i \mid retn \mid cmp\, r\, r \mid cmp\, r\, i \mid je\, r \mid je\, i$$
$$gpr \in GPRS$$
$$r \in Registers$$
$$i \in val$$

A program is a partial function from a value to an instruction.

$$Program = val \rightarrow inst$$

Memory is complete function from a value to a value.

$$Memory = val \rightarrow val$$

The formal configuration of the structural operational semantics then is as follows:

$$Conf = Program \times Registers \times Memory \times Flags$$

The semantic is complete with the addition of a reduction relation from C to C' that is as forllows:

$$C \Rightarrow C' \quad where\, C, C' \in Conf$$

2.4 Semantic

The memory segment have to be initialised with the proper values, these can be found in different way depending on the format of the program. For elf files the sections that is most important is the .text .data and .rodata, as they are copied into memory, the .bss data is not defined by x86-64 but is left up to the operation system of how to handle, this can be zeroing on initialisation, zero on use or nothing. For this paper we will look at zeroing on initialisation as used by unix like systems and windows. All flags are initialised as 0, same for registers. The program counter is update by adding 8 as the size of an instruction is 8 bytes. The compare instruction Equation 2.8 also change CF, OF, SF, AF and PF flags but as they are not used any other place in the semantic this was cut to make the compare instruction more readable.

$$[mov - r - r]\ \frac{P(R(rpc)) = mov\ gpr\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[gpr \rightarrow R(r), rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.1}$$

$$[mov - r - i]\ \frac{P(R(rpc)) = mov\ gpr\ i}{\langle P, R, M, F\rangle \rightarrow \langle P, R[gpr \rightarrow i, rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.2}$$

$$[push - r]\ \frac{P(R(rpc)) = push\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rsp \rightarrow R(rsp) - 8, rpc \rightarrow R(rpc) + 8], M[R(rsp) \rightarrow R(r)], F\rangle} \tag{2.3}$$

$$[push - i]\ \frac{P(R(rpc)) = push\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rsp \rightarrow R(rsp) - 8, rpc \rightarrow R(rpc) + 8], M[R(rsp) \rightarrow i], F\rangle} \tag{2.4}$$

$$[pop - r]\ \frac{P(R(rpc)) = pop\ gpr}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rsp \rightarrow R(rsp) + 8, R(rpc) \rightarrow R(rpc) - 8, R(gpr) \rightarrow M(R(rsp))], M, F\rangle} \tag{2.5}$$

$$[call - r]\ \frac{P(R(rpc)) = call\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow R(r), rbp \rightarrow R(rsp) - 8], M[R(rsp) \rightarrow R(rpc) - 8, R(rsp) - 8 \rightarrow R(rbp)], F\rangle} \tag{2.6}$$

$$[call - i]\ \frac{P(R(rpc)) = call\ i}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow i, rbp \rightarrow R(rsp) - 8], M[R(rsp) \rightarrow R(rpc) - 8, R(rsp) - 8 \rightarrow R(rbp)], F\rangle} \tag{2.7}$$

$$[CMP - r - r - true]\ \frac{P(R(rpc)) = cmp\ r_1, r_2}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow R(rpc) + 8], M, F[zf \rightarrow 1]\rangle} if\ r_1 == r_2 \tag{2.8}$$

$$[CMP - r - r - false]\ \frac{P(R(rpc)) = cmp\ r_1, r_2}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow R(rpc) + 8], M, F[zf \rightarrow 0]\rangle} if\ r_1 != r_2 \tag{2.9}$$

$$[je - r - true]\ \frac{P(R(rpc)) = je\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow R(r)], M, F\rangle} if\ f(zf) = 1 \tag{2.10}$$

$$[je - r - false]\ \frac{P(R(rpc)) = je\ r}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow R(rpc) + 8], M, F\rangle} if\ f(zf) = 0 \tag{2.11}$$

$$[retn]\ \frac{P(R(rpc)) = retn}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow M(R(rbp) - 8), rsp \rightarrow R(rbp) - 16, rbp \rightarrow M(R(rbp))], M, F\rangle} \tag{2.12}$$

2.4.1 Registers

The initial value of registers is defined in the ABI for the machine and therefor there is no single way it is done. The UNIX elf ABI handle this by zeroing all registers this way will be used for this language. The special purpose registers RIP is initialised to the entry point of the program.

RIP

RIP is a special purpose register that contains the next instruction to be executed, every instruction that is executed got a side effect of updating the instruction pointer. RIP can not be change by the program other then by side effect. RIP can still be used for position-independent code addressing.

RSP

RSP is used to store the stack pointer, it contanis the address of the last value on the stack. It is used by the push and pop instruction to access the stack

RBP

RBP is the base pointer, holding a pointer to the start of the current frame. RBP is commenly used to access the local variables of the current frame, it is also used when the return instruction is called, as the old base pointer is stored on the stack at the location that the current base pointer is pointing. It can be used as a general purpose register but have to be restored before return is called.

2.4.2 Memory

In theroy x86-64 would make it possible to address 64bit of memory but as that is far larger then what is needed for applications as of 2019, therefor chips does not implement the entire 64bit address space but only a limited address space, the current AMD64 architecture only use the lower 48bit to address memory. As this is a limit set by proccessor developers and not by the x86-64 language it is not reflected in the semantics, as this limit will in theory be removed in time.[1]

2.4.3 Stack

The stack is a memory region that is used to store values, the stack is when calling functions depending on the calling convension. As this is not part of the language the only part that is in the syntax is the push and pop instruction that is made for interaction with the stack

2.5 Typesystem

The base of the type system is introduced in section 1.1 and will be slightly modified and used.

2.5.1 Types

The typesystem that was proposed in the old paper and shown in the introduction Figure 1.1 show a complete typesystem for x86-64. For the subset that will be made type rules for only the 64bit types are intresting and therefor all

the other types have been removed. The arrows show the relations between type so that $a \sqsupseteq b$, mean that a is a sub type of b. So every place where you can use a you can also use b. This relation is transitive meaning that $int64_t$ can be substituted with $num64_t$.

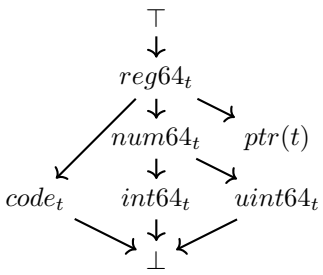

Figure 2.1: Subtypes

2.5.2 Formalisation

$$T_r(pc) = Register \rightarrow Type$$
$$T_m(pc) = Val \rightarrow Type$$

Both T_r and T_m are complete functions, and all registers and memory addresses start with the type $\top$ as there are no type information on them. As only 64 bit types are left

$$T_r(pc) \sqsupseteq T_r(pc+1) iff \forall_r \in Reg : T_r(pc)(r) \sqsupseteq T_r(pc+1)(r)$$
$$T_r(pc) \sqsupseteq_{\setminus r'} T_r(pc+1) iff \forall_r \in Reg \setminus \{r'\} : T_r(pc)(r) \sqsupseteq T_r(pc+1)(r)$$
$$T_m(pc) \sqsupseteq T_m(pc+1) iff \forall_a ddr \in Val : T_m(pc)(addr) \sqsupseteq T_m(pc+1)(addr)$$
$$T_m(pc) \sqsupseteq_{\setminus addr'} T_m(pc+1) iff \forall_a ddr \in Val \setminus addr' : T_m(pc)(addr) \sqsupseteq T_m(pc+1)(addr)$$
$$T_{64} = reg64_t \vee ptr(t) \vee num64_t \vee code_t \vee int64_t \vee uint64_t$$

$$[MOV - R - R] \frac{T_r(pc)(r_2) = T \quad T \sqsupseteq T_r(pc+1)(r_1) \quad T_r(pc) \sqsupseteq_{\setminus r_1} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ r_1\ r_2 : Inst} \tag{2.13}$$

$$[MOV - R - im] \frac{T_r(pc)(r) = T \quad T \sqsupseteq T_{64} \quad T_r(pc) \sqsupseteq_{\setminus r} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ r\ im : Inst} \tag{2.14}$$

$$[CMP - R - R] \frac{T_r(pc)(r_1) \sqsupseteq reg64_t \quad T_r(pc)(r_2) \sqsupseteq reg64_t \quad T_r(pc) \sqsupseteq T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : cmp\ r_1\ r_2 : Inst} \tag{2.15}$$

$$[CMP - R - R] \frac{T_r(pc)(r_1) \sqsupseteq reg64_t \quad T_{64} \sqsupseteq reg64_t \quad T_r(pc) \sqsupseteq T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : cmp\ r_1\ im : Inst} \tag{2.16}$$

$$[JE - r - false] \frac{T_r(pc)(r)ptr(code_t) \quad T_r(pc) \sqsupseteq T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : je\ r : Inst} f(zf) = 0 \tag{2.17}$$

$$[JE - r - true] \frac{T_r(pc)(r) \sqsupseteq ptr(code_t) \quad T_r(pc) \sqsupseteq T_r(R(r)) \quad T_m(pc) \sqsupseteq T_m(R(r))}{T_r, T_m \vdash pc : je\ r : Inst} f(zf) = 1 \tag{2.18}$$

2.5.3 Push and pop

The instruction show in the prior section is the basic control flow instructions, push and pop is used to access memory more specifically the stack.

$$[push - R] \frac{T_m(pc)(rsp) = T \quad T \sqsupseteq T_r(pc+1)(r) \quad T_r(pc) \sqsupseteq T_r(pc+1) \quad T_m(pc) \sqsupseteq_{\setminus rsp} T_m(pc+1)}{T_r, T_m \vdash pc : push\ r : Inst} \tag{2.19}$$

$$[pop - R] \frac{T_r(pc)(r) = T \quad T \sqsupseteq T_m(pc+1)(rsp) \quad T_r(pc) \sqsupseteq_{\setminus r} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : pop\ r : Inst} \tag{2.20}$$

2.5.4 Example

The code the syntax is used on is the same from section 2.2 but it stop after the call instruction as the stack is modified inside the function call. The stack is presumed to be empty when the program function is called this not the case when calling the function, but as this is a driver it does not have the initialization code as part of the program.

$$R = \{rbp = 0x1808, rsp = 0x1808\}$$

$$M = \{\}$$

$$F = \{\} \tag{2.21}$$

$$[push - R]\ \frac{P(R(rpc)) = push\ rbp}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rsp \rightarrow R(rsp) - 8, rpc \rightarrow R(rpc) + 8], M[R(rsp) \rightarrow R(rbp)], F\rangle} \tag{2.22}$$

$$R = \{rbp = 0x1808, rsp = 0x1800\}$$

$$M = \{0x1808 = 0x1808\}$$

$$F = \{\}$$

$$[mov - R - I]\ \frac{P(R(rpc)) = mov\ rdx\ 0x0786}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rdx \rightarrow 0x0786, rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.23}$$

$$R = \{rbp = 0x1808, rsp = 0x1800, rdx = 0x0786\}$$

$$M = \{0x1808 = 0x1808\}$$

$$F = \{\}$$

$$[mov - R - I]\ \frac{P(R(rpc)) = mov\ rsi\ 0x0D40}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rsi \rightarrow 0x0D40, rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.24}$$

$$R = \{rbp = 0x1808, rsp = 0x1800, rdx = 0x0786, rsi = 0x0D40\}$$

$$M = \{0x1808 = 0x1808\}$$

$$F = \{\}$$

$$[mov - R - I]\ \frac{P(R(rpc)) = mov\ rdi\ 0x0EB8}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rdi \rightarrow 0x0EB8, rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.25}$$

$$R = \{rbp = 0x1808, rsp = 0x1800, rdx = 0x0786, rsi = 0x0D40, rdi = 0x0EB8\}$$

$$M = \{0x1808 = 0x1808\}$$

$$F = \{\}$$

$$[mov - R - R]\ \frac{P(R(rpc)) = mov\ rbp\ rsp}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rbp \rightarrow R(rsp), rpc \rightarrow R(rpc) + 8], M, F\rangle} \tag{2.26}$$

$$R = \{rbp = 0x1800, rsp = 0x1800, rdx = 0x0786, rsi = 0x0D40, rdi = 0x0EB8, rbp = rsp\}$$

$$M = \{0x1808 = 0x1808\}$$

$$F = \{\}$$

$$[call - R]\ \frac{P(R(rpc)) = call\ 0x10B8}{\langle P, R, M, F\rangle \rightarrow \langle P, R[rpc \rightarrow 0x10B8, rbp \rightarrow R(rsp) - 8],}$$

$$M[R(rsp) \rightarrow R(rpc) - 8, R(rsp) - 8 \rightarrow R(rbp)], F\rangle \tag{2.27}$$

$$R = \{rbp = 0x1800, rsp = 0x1800, rdx = 0x0786, rsi = 0x0D40, rdi = 0x0EB8, rbp = rsp\}$$

$$M = \{0x1808 = 0x1808\ 0x1800 = R(rpc) - 8\}$$

$$F = \{\}$$

(2.28)

Type Example

$$[push - R]\ \frac{T_m(pc)(rsp) = T \quad T \sqsupseteq T_r(pc+1)(rbp) \quad T_r(pc) \sqsupseteq T_r(pc+1) \quad T_m(pc) \sqsupseteq_{\backslash rsp} T_m(pc+1)}{T_r, T_m \vdash pc : push\ rbp : Inst} \tag{2.29}$$

$$T_m = \{rsp \sqsupseteq t_r(rbp)\}$$

$$T_r = \{\}$$

$$[MOV - R - im]\ \frac{T_r(pc)(rbx) = T \quad T \sqsupseteq T_{64} \quad T_r(pc) \sqsupseteq_{\backslash r} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ rdx\ 0x0786 : Inst} \tag{2.30}$$

$$T_m = \{rsp \sqsupseteq t_r(rbp)\}$$

$$T_r = \{rbx \sqsupseteq T_{64}\}$$

$$[MOV - R - im]\ \frac{T_r(pc)(rsi) = T \quad T \sqsupseteq T_{64} \quad T_r(pc) \sqsupseteq_{\backslash r} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ rpc\ 0xd40 : Inst} \tag{2.31}$$

$$T_m = \{rsp \sqsupseteq t_r(rbp)\}$$

$$T_r = \{rbx \sqsupseteq T_{64}, rsi \sqsupseteq T_{64}\}$$

$$[MOV - R - im]\ \frac{T_r(pc)(rdi) = T \quad T \sqsupseteq T_{64} \quad T_r(pc) \sqsupseteq_{\backslash r} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ rdx\ 0x0eb8 : Inst} \tag{2.32}$$

$$T_m = \{rsp \sqsupseteq t_r(rbp)\}$$

$$T_r = \{rbx \sqsupseteq T_{64}, rsi \sqsupseteq T_{64}, rdi \sqsupseteq T_{64}\}$$

$$[MOV - R - R]\ \frac{T_r(pc)(rsp) = T \quad T \sqsupseteq T_r(pc+1)(rbp) \quad T_r(pc) \sqsupseteq_{\backslash r_1} T_r(pc+1) \quad T_m(pc) \sqsupseteq T_m(pc+1)}{T_r, T_m \vdash pc : mov\ rbp\ rsp : Inst} \tag{2.33}$$

$$T_m = \{rsp \sqsupseteq t_r(rbp)\}$$

$$T_r = \{rbx \sqsupseteq T_{64}, rsi \sqsupseteq T_{64}, rdi \sqsupseteq T_{64}\}$$

(2.34)

The example does not have any real type information, with the current type system and the call function will add a bit of information as the location that is called is know to be code. But as the syntax is mainly control flow there is not much type information that can be gained.

Part II

Closing

3. Conclution

A syntax and typesystem for a subset of x86-64 have been proposed, [2] was used as inspiration for the syntax. Both the syntax and typesystem is mainly of control flow instruction because of this there are not much type information that can be gained from this, adding a few instruction like addition, subtraction, multiplication and division would be able to narrow the types more. Type information is found for every value of the program counter, this way the type information of the volatile registers is saved, the same is true for the stack that is mean to reused. As memory protection is not implemented all memory is actually volatile there is nothing that stop the program from changing it. Hard coded data like the name of the driver from section 2.2 is normally stored in memory and is considered none volatile as compilers wont used the memory space for other things even it is done using it for the hard coded value.

Bibliography

[1] amd. (2012). Bios and kernel developer's guide(bkdg)for amd family 10hprocessors. English, amd, [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/31116.pdf (visited on Oct. 7, 2019).

[2] R. Hansen, K. Larsen, M. Olesen, and E. Wognsen, "Formal modelling and analysis of bitflips in arm assembly code", English, 5, vol. 18, Springer, Jun. 2016, pp. 909–925. DOI: 10.1007/s10796-016-9665-7.