Addendum to "AVR Processors as a Platform for Language-Based Security"

Florian Dewald, Heiko Mantel, and Alexandra Weber

Computer Science Department, TU Darmstadt, Germany
{dewald,mantel,weber}@mais.informatik.tu-darmstadt.de

This addendum document is structured as follows:

- In Section A, we provide our complete operational semantics. It consists of definitions to model data storages, instructions and programs. The section concludes with presenting and arguing for the faithfulness of our small-step semantics.
- In Section B, we start of by repeating domain assignments and our security property. We continue by introducing all special concepts used in our security type system, before we define typability and present the complete set of typing rules.
- In Section C, we prove our security type system sound with respect to our operational semantics and security property.

A Complete Definition of Semantics

A.1 Hardware Modeling

Definition 1 (Status registers). *The set of status registers is* $\{C, Z\}$.

Definition 2 (Status register assignment). *Values of status registers are given by a function out of the set*

$$\texttt{STAT-VAL} := \{C, Z\} \to \{0, 1\}$$

The identifier Z represents the zero status flag and the C represents the carry status flag.

Definition 3 (8-bit values). *The set* $\texttt{VAL}_8$ *contains all possible values, that can be stored on an 8-bit machine. That is, the set* $\texttt{VAL}_8$ *is defined as:*

$$\texttt{VAL}_8 := \{z \in \mathbb{Z} \mid z \geq -2^7 \wedge z \leq 2^7 - 1\}$$

AVR operates with 8-bit values, so the set $\texttt{VAL}_8$ represents all possible representable values in 8-bit two's complement.

Definition 4 (Registers). *The set* REG *denotes the set of available registers and is given by*

$$\texttt{REG} := \{r_n \mid n \in \{0, \ldots, 31\}\} \cup \{\texttt{sp}_l, \texttt{sp}_u\}$$

Definition 5 (Register assignment). *Values of registers are given by a function out of the set*

$$\texttt{REG-VAL} := \texttt{REG} \to \texttt{VAL}_8$$

The AVR microcontrollers contain 32 general purpose registers, in our case identified by $r_0, \ldots, r_{31}$. The stack pointer is 16-bit wide, requiring it to be split into two separate registers $\texttt{sp}_l$ for the lower part and $\texttt{sp}_u$ for the higher part of a 16-bit value.

Definition 6 (16-bit values). *The set* $\texttt{VAL}_{16}$ *contains all possible 16-bit values and is given by*

$$\texttt{VAL}_{16} := \{z \in \mathbb{Z} \mid z \geq -2^{15} \wedge z \leq 2^{15} - 1\}$$

Definition 7 (Register pairs). *The set* REGPAIR *contains all possible register pairs and is given by*

$$\texttt{REGPAIR} := \{r_0, \ldots, r_{30}\} \cup \{X, Y, Z, \texttt{sp}_l\}$$

Definition 8 (Low register pair conversion). *The function* regconvert-l : $\texttt{REGPAIR} \to \texttt{REG}$ *maps a given register pair to the lower concatenated register and is given by*

$$\text{regconvert-l}(a) := \begin{cases} a & \text{if } a \in \{r_0, \ldots, r_{30}\} \\ r_{26} & \text{if } a = X \\ r_{28} & \text{if } a = Y \\ r_{30} & \text{if } a = Z \\ \texttt{sp}_l & \text{if } a = \texttt{sp}_l \end{cases}$$

Definition 9 (Upper register pair conversion). *The function* regconvert-u : $\texttt{REGPAIR} \to \texttt{REG}$ *maps a given register pair to the upper concatenated register and is given by*

$$\text{regconvert-u}(a) := \begin{cases} r_{n+1} & \text{if } \exists r_n \in \{r_0, \ldots, r_{30}\} : a = r_n \\ r_{27} & \text{if } a = X \\ r_{29} & \text{if } a = Y \\ r_{31} & \text{if } a = Z \\ \texttt{sp}_u & \text{if } a = \texttt{sp}_l \end{cases}$$

Definition 10 (Register pair assignment). *Let* $r \in \texttt{REG-VAL}$ *be a register assignment. The function*

$$\text{regpair-val}_r : \texttt{REGPAIR} \to \texttt{VAL}_{16}$$

is given by

$$\text{regpair-val}_r(a) := r(\text{regconvert-u}(a)) \odot r(\text{regconvert-l}(a))$$

The stack pointer is a first example of register pairs. Register pairs are used for 16-bit wide values by concatenating two adjacent registers. Given a register, the corresponding lower and upper registers that belong to the pair can be obtained using functions regconvert-l and regconvert-u, respectively. For convenience, the 16-bit value stored in a register pair with respect to a register assignment can be obtained using $\mathsf{regpair\text{-}val}_r$.

Definition 11 (Memory addresses). *The memory addresses are given by the set*

$$\texttt{MEM} := \{0, \ldots, \texttt{MAXADDR}\}$$

Definition 12 (Memory assignment). *Values of memories are given by a function out of the set*

$$\texttt{MEM-VAL} := \texttt{MEM} \to \texttt{VAL}_8$$

A main memory cell is identified by its address, given by MEM. It contains all available memory addresses, where the largest one is denoted by MAXADDR.

Definition 13 (Stack assignment). *The set of possible* stack assignments *is given by* $\texttt{STACK-VAL} := \texttt{VAL}_8^*$.

The stack is modeled by a list of values, where the head of the list represents the topmost element on the stack.

Definition 14 (Function updating). *Let $f : X \to Y$ be a function, $x \in X$ and $y \in Y$, then the function $f[x \mapsto y] : X \to Y$ is given by*

$$f[x \mapsto y](a) := \begin{cases} f(a) & \text{if } a \neq x \\ y & \text{if } a = x \end{cases}$$

When computations on values are performed and the result shall be stored inside a data storage, it might happen that the result is too large for the value range of the desired data storage. Unfortunately, we were unable to find an exact documentation what happens in such a case. A reasonable and in this work used way is that the stored value is determined bitwise to match the bit length of the desired storage container and if necessary the most significant bits are cut off.

Example 1. Assume the result of adding $a = 1111\ 1111_2$ and $b = 0000\ 0001_2$ shall be stored inside a register r that can hold 8 bits. The result would be $r = a + b = 1\ 0000\ 0000_2$, where only the last 8 bit can be stored inside the register. Thus, the final content of r is $0000\ 0000_2$.

Definition 15 (Register pair updating). *Let $r \in \texttt{REG-VAL}$ be a register assignment, $r_n \in \texttt{REGPAIR}$ be a possible register pair and $z \in \texttt{VAL}_{16}$ a new value. Then the function $r[r_n \mapsto_p z] : \texttt{REG} \to \texttt{VAL}_8$ is given by*

$$r[r_n \mapsto_p z](a) := \begin{cases} \mathsf{lower}(z) & \text{if } a = \mathsf{regconvert\text{-}l}(r_n) \\ \mathsf{upper}(z) & \text{if } a = \mathsf{regconvert\text{-}u}(r_n) \\ r(a) & \text{otherwise} \end{cases}$$

Definition 16 (Status Register updating). *Let* $s \in \texttt{STAT-VAL}$ *be a status register assignment,* $x \in \{C, Z\}$ *a status register and* $t \in \mathbb{B}$ *a truth value, then the function* $s[x \mapsto_s t]$ *is given by*

$$s[x \mapsto_s t] := \begin{cases} s[x \mapsto 1] & \text{if } t = \mathit{True} \\ s[x \mapsto 0] & \text{if } t = \mathit{False} \end{cases}$$

Special notations are introduced to update register pairs and status registers. When a register pair is updated, the 16-bit value is split in its lower and upper part and put in the corresponding registers. The notation for updating a status register allows to pass a truth value which is converted into 0 and 1.

A.2 Instructions and Programs

Definition 17 (Execution point). *The set of* base execution points *is given by*

$$\texttt{EPS}_0 := \{(f, a) \mid f \in \texttt{FUNC} \wedge a \in \mathbb{N}\}$$

The set of execution points *is given by*

$$\texttt{EPS} := \texttt{EPS}_0 \times \texttt{EPS}_0^*$$

Definition 18 (Execution Point addition). *The function* $+_{\texttt{ep}} : \texttt{EPS} \times \mathbb{N} \to \texttt{EPS}$ *is given by*

$$+_{\texttt{ep}}((f, a, c), n) := (f, a + n, c)$$

An execution point models a pointer to a location inside a program. We distinguish between base execution points and plain execution points. A base execution point is a pair consisting of a function name and an address to determine the instruction inside the function. It corresponds to the layout found inside an object dump that acts as input for our prototypical implementation. An execution point contains additionally a list of base execution points that act as call stack to determine return locations upon a function call.

Whenever necessary, we rely on implicit conversions from execution points to base execution points, where the call stack is dropped.

The special $+_{\texttt{ep}}$ symbol is used to shift forward the address of an execution point without affecting its remaining components.

Definition 19 (Instructions without operands). *The set*

$$\texttt{INSTR-NO-OP} := \{(in, ()) \mid in \in \{\texttt{clc}, \texttt{cli}, \texttt{ret}\}\}$$

contains instructions which take no operand.

Definition 20 (Instructions influencing control flow). *The set*

$$\begin{aligned} \texttt{INSTR-CF} := &\{(in, (epa)) \mid epa \in \texttt{EPS} \wedge in \in \\ &\{\texttt{brcc}, \texttt{brcs}, \texttt{breq}, \texttt{brne}, \texttt{call}, \texttt{jmp}, \texttt{rcall}, \texttt{rjmp}\}\} \end{aligned}$$

contains instructions which take an execution point as operand.

Definition 21 (Instructions with immediate and register operands). *The set*

$$\begin{aligned}\texttt{INSTR-IMM-REG} :=\{(in,(Rd,im)) \mid im \in \mathbb{Z} \wedge Rd \in \texttt{REG} \wedge in \in \\ \{\texttt{adiw},\texttt{andi},\texttt{cpi},\texttt{ldi}, \\ \texttt{sbci},\texttt{sbiw},\texttt{subi}\}\} \cup \\ \{(\texttt{in},(Rd,a)),(\texttt{out},(a,Rr)) \mid im \in \mathbb{Z} \wedge \\ Rr \in \texttt{REG} \wedge Rd \in \texttt{REG} \wedge a \in \{\texttt{0x3d},\texttt{0x3e},\texttt{0x3f}\}\}\end{aligned}$$

contains instructions which take an immediate and a register(-pair) as operand.

Definition 22 (Instructions with one register as operand). *The set*

$$\begin{aligned}\texttt{INSTR-REG} :=\{(in,(Rd)) \mid Rd \in \texttt{REG} \wedge in \in \\ \{\texttt{dec},\texttt{inc},\texttt{lsr},\texttt{neg},\texttt{pop},\texttt{push},\texttt{ror}\}\}\end{aligned}$$

contains instructions which take one register as operand.

Definition 23 (Instructions with two register operands). *The set*

$$\begin{aligned}\texttt{INSTR-TWO-REG} :=\{(in,(Rd,Rr)) \mid Rd,Rr \in \texttt{REG} \wedge in \in \\ \{\texttt{adc},\texttt{add},\texttt{and},\texttt{cp},\texttt{cpc},\texttt{cpse},\texttt{eor},\texttt{mov}, \\ \texttt{movw},\texttt{mul},\texttt{or},\texttt{sbc},\texttt{sub}\}\}\end{aligned}$$

contains instructions which take two registers as operand.

Definition 24 (Instructions with special registers as operand). *The set*

$$\begin{aligned}\texttt{INSTR-REG-SPE} :=\{(\texttt{ld},(Rd,Rs,m)),(\texttt{st},(Rs,Rr,m)) \mid Rs \in \{X,Y,Z\} \wedge \\ Rd,Rr \in \texttt{REG} \wedge m \in \{-,+,\#\}\}\}\end{aligned}$$

contains instructions which take a special register and a modifier as operand.

Definition 25 (Instructions with special registers and displacement). *The set*

$$\begin{aligned}\texttt{INSTR-REG-SPE-DIS} :=\{(\texttt{ldd},(Rd,Rs,k)),(\texttt{std},(Rs,Rr,k)) \mid Rs \in \{X,Y,Z\} \wedge \\ Rd,Rr \in \texttt{REG} \wedge k \in \mathbb{Z}\}\}\end{aligned}$$

contains instructions which take a special register and a modifier as operand.

Definition 26 (Instructions). *The set*

$$\begin{aligned}\texttt{INSTR} :=&\texttt{INSTR-NO-OP} \cup \texttt{INSTR-CF} \cup \texttt{INSTR-IMM-REG} \cup \\ &\texttt{INSTR-REG} \cup \texttt{INSTR-TWO-REG} \cup \texttt{INSTR-REG-SPE} \cup \\ &\texttt{INSTR-REG-SPE-DIS}\end{aligned}$$

contains all possible instructions.[1]

[1] The notation for instructions in "AVR Processors as a Platform for Language-Based Security" omits the commas and parantheses in instructions for presentation purposes.

Instructions are pairs consisting of the instruction name and a possibly empty list of arguments. The allowed value range of some arguments vary between instructions. We assume that no invalid arguments are given. This assumption is safe as a compiler should respect all argument restrictions.

Definition 27 (Instruction execution time). *The function* $\mathtt{t} : \mathtt{INSTR} \to \mathbb{N}$ *assigns every instruction its corresponding execution time in clock cycles.*

It is defined as follows:

$$\mathtt{t}(i) = \begin{cases} 1 & i \in \{(\mathtt{adc}, (Rd, Rr)), (\mathtt{add}, (Rd, Rr)), (\mathtt{and}, (Rd, Rr)), (\mathtt{andi}, (Rd, k)), \\ & \quad (\mathtt{brcc}, (epa)), (\mathtt{brcs}, (epa)), (\mathtt{breq}, (epa)), (\mathtt{brne}, (epa)), (\mathtt{clc}, ()), \\ & \quad (\mathtt{cli}, ()), (\mathtt{cp}, (Rd, Rr)), (\mathtt{cpc}, (Rd, Rr)), (\mathtt{cpse}, (Rd, Rr)), \\ & \quad (\mathtt{cpi}, (Rd, k)), (\mathtt{dec}, (Rd)), (\mathtt{eor}, (Rd, Rr)), (\mathtt{in}, (Rd, k)), \\ & \quad (\mathtt{inc}, (Rd)), (\mathtt{ld}, (Rd, Rs, \#)), (\mathtt{ldi}, (Rd, k)), (\mathtt{lsr}, (Rd)), \\ & \quad (\mathtt{mov}, (Rd, Rr)), (\mathtt{movw}, (Rd, Rr)), (\mathtt{neg}, (Rd)), (\mathtt{or}, (Rd, Rr)), \\ & \quad (\mathtt{out}, (k, Rr)), (\mathtt{ror}, (Rd)), (\mathtt{sbc}, (Rd, Rr)), (\mathtt{sbci}, (Rd, k)), \\ & \quad (\mathtt{sub}, (Rd, Rr)), (\mathtt{subi}, (Rd\ k))\} \\ 2 & i \in \{(\mathtt{adiw}, (Rd, k)), (\mathtt{ld}, (Rd, Rs, +)), (\mathtt{ldd}, (Rd, Rs, k)), (\mathtt{mul}, (Rd, Rr)), \\ & \quad (\mathtt{pop}, (Rd)), (\mathtt{push}, (Rr)), (\mathtt{rjmp}, (epa)), (\mathtt{sbiw}, (Rd, k)), \\ & \quad (\mathtt{st}, (Rs, k, -)), (\mathtt{st}, (Rs, k, +)), (\mathtt{st}, (Rs, k, \#)), (\mathtt{std}, (Rs, Rr, k))\} \\ 3 & i \in \{(\mathtt{jmp}, (epa)), (\mathtt{ld}, (Rd, Rs, -)), (\mathtt{rcall}, (epa))\} \\ 4 & i \in \{(\mathtt{call}, (epa)), (\mathtt{ret}, ())\} \end{cases}.$$

Given an instruction, the function t returns its execution time. A special constant $\mathtt{br} := 1$ is used for branching instructions. When a branching condition is satisfied such that a branching is performed, the instruction takes additional br cycles to be executed. This additional timing is reflected in our small-step semantics.

Definition 28 (Two word instructions). *The predicate* $\mathtt{tw}_P : \mathtt{EPS} \to \mathbb{B}$ *is defined as:*

$$\mathtt{tw}_P(ep) := \exists (i, a) \in \mathtt{INSTR} : P(ep) = (i, a) \wedge i \in \{\mathtt{call}, \mathtt{jmp}\}$$

Some instructions are larger than normal, affecting the execution time of cpse. Those instruction can be identified by the predicate tw.

Definition 29 (Program). *A* program *is a function out of the set*

$$\mathtt{PROG} := \mathtt{EPS}_0 \rightharpoonup \mathtt{INSTR}$$

A program is a mapping from base execution points to instructions. We only consider programs that satisfy a well-formedness criterion, namely that each function contains a unique return instruction ret and the arguments to all instructions lie within the ranges permitted according to [2]. The execution point

arguments to all instructions must lie within the program. Immediate arguments must lie in the range $[0, 63]$ for all `adiw`, `sbiw`, `ldd` and `std` instructions and in the range $[0, 255]$ for all `andi`, `cpi`, `ldi`, `sbci`, and `subi` instructions. The special registers given as arguments to all `ldd` and `std` instructions must be from the set $\{Y, Z\}$. Register arguments must be from the set $\{r_n \mid n \in \{24, 26, 28, 30\}\}$ for all `adiw` and `sbiw` instructions and from the set $\{r_n \mid n \in [16, 31]\}$ for all `andi`, `cpi`, `ldi`, `sbci` and `subi` instructions.

A unique return instruction in each function can be achieved by rewriting return instructions in branches to jumps to a common return instruction. The permitted ranges for arguments should be enforced by compilation.

In addition to the well-formedness criterion we require that the immediate arguments to all `in` and `out` instructions are from the set $\{0x3f, 0x3e, 0x3d\}$. The values in this set correspond to the addresses of the status register, $\texttt{sp}_u$ and $\texttt{sp}_l$ on an ATmega microcontroller [1]. The requirement on the arguments of `in` and `out` instructions restricts the support of these instructions. However, the supported arguments suffice already to cover relevant cryptographic implementations like the μNaCl implementations of Salsa20, XSalsa20 and Poly1305.

A.3 Big-Step Semantics

Definition 30 (States). *The set of possible microcontroller states is given by*

$$\texttt{STATE} := \texttt{STAT-VAL} \times \texttt{MEM-VAL} \times \texttt{REG-VAL} \times \texttt{STACK-VAL} \times (\texttt{EPS} \cup \{\epsilon\})$$

Definition 31 (Execution point selector). *The function* $\mathsf{epselect} : \texttt{STATE} \to (\texttt{EPS} \cup \{\epsilon\})$ *selects the current execution point from a state. It is defined as follows*

$$\mathsf{epselect}\,((sr, m, r, st, ep)) := ep$$

A state combines all hardware data storages from Section A.1 with the current execution point. The ϵ symbol models a terminating state.

Definition 32 (Transition relation). *Let* $P \in \texttt{PROG}$ *be a program . The* transition relation $\Downarrow_P^n \subseteq \texttt{STATE} \times \texttt{STATE} \times \mathbb{N}$ *is then given by the rules*

$$\frac{(sr, m, r, st, ep) \xrightarrow{c}_P (sr', m', r', st', ep') \quad (sr', m', r', st', ep') \Downarrow_P^{c'} (sr'', m'', r'', st'', \epsilon)}{(sr, m, r, st, ep) \Downarrow_P^{c+c'} (sr'', m'', r'', st'', \epsilon)}\ (\mathsf{Seq})$$

$$\frac{(sr, m, r, st, ep) \xrightarrow{c}_P (sr', m', r', st', \epsilon)}{(sr, m, r, st, ep) \Downarrow_P^{c} (sr', m', r', st', \epsilon)}\ (\mathsf{Ter})$$

where $(sr, m, r, st, ep), (sr', m', r', st', ep'), (sr'', m'', r'', st'', \epsilon) \in \texttt{STATE}$ *and* $c, c' \in \mathbb{N}$.

Our transition relation captures a whole execution. If $s \Downarrow_P^c s'$, then an execution starting in state s terminates in state s' after running for c clock cycles. Rule (Seq) models a sequential execution step while rule (Ter) allows termination.

A.4 Small-Step Semantics

Our small-step semantics consists of rules of the form $s \xrightarrow{c}_P s'$, representing that executing a single instruction in state s leads to state s' and requires c clock cycles. In the following, we present our full small-step semantics as well as arguments for the faithfulness of every small-step rule we have modeled with respect to the official instruction manual [2].

Arithmetic Instructions Arithmetic instructions have in common that they modify the Z status flag and overwrite the contents of the first register, mostly named Rd, according to the operation performed.

The instruction manual gives a common boolean formula for the Z status register flag, with the only exceptions being instructions operating on register pairs and instructions `cp`, `cpi`, `cpc`, `sbc` and `sbci`. The formula given is $\neg r'(Rd)_{[7]} \wedge \ldots \wedge \neg r'(Rd)_{[0]}$. This formula is captured by the update to the status register assignment $[Z \mapsto_s r'(Rd) = 0]$. In the condition $r'(Rd) = 0$, r' is the register assignment after execution of the instruction and Rd is the destination register, where the result is written to. Thus, it adequately checks if the result is zero and sets the Z bit accordingly.

For the carry flag C the situation looks a little different. Each instruction sets the C flag according to a different boolean formula. We thus give the formula found in [2]. This flag is translated directly into the boolean formula used to update the carry flag's value $sr(C)$.

Apart from providing faithfulness arguments for common aspects of arithmetic instructions, we provide detailed arguments for every single small step semantic rule in the following.

Adc

$$\frac{\begin{array}{c} P(ep) = (\texttt{adc}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) + r(Rr) + sr(C)] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0] \\ cf = (r(Rd)_{[7]} \wedge r(Rr)_{[7]}) \vee (r(Rr)_{[7]} \wedge \neg r'(Rd)_{[7]}) \vee (\neg r'(Rd)_{[7]} \wedge r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{adc})$$

Description	The `adc` instruction takes two registers and sums up their contents, including the carry flag. The sum is stored in the first passed register Rd.
Status Flags	C: $\big(r(Rd)_{[7]} \wedge r(Rr)_{[7]}\big) \vee \big(r(Rr)_{[7]} \wedge \neg r'(Rd)_{[7]}\big) \vee \big(\neg r'(Rd)_{[7]} \wedge r(Rd)_{[7]}\big)$
Faithfulness Effect	The desired effect of addition is captured by the premise $r' = r[Rd \mapsto r(Rd) + r(Rr) + sr(C)]$. It performs an addition of the two passed registers and the carry flag. The result is stored in register Rd.

Add

$$\frac{\begin{array}{c}P(ep) = (\texttt{add}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) + r(Rr)] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0] \\ cf = (r(Rd)_{[7]} \wedge r(Rr)_{[7]}) \vee (r(Rr)_{[7]} \wedge \neg r'(Rd)_{[7]}) \vee (\neg r'(Rd)_{[7]} \wedge r(Rd)_{[7]})\end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{add})$$

Description	The `add` instruction takes two registers and sums up their contents. The sum is stored in the first passed register Rd.
Status Flags	C: $\big(r(Rd)_{[7]} \wedge r(Rr)_{[7]}\big) \vee \big(r(Rr)_{[7]} \wedge \neg r'(Rd)_{[7]}\big) \vee \big(\neg r'(Rd)_{[7]} \wedge r(Rd)_{[7]}\big)$
Faithfulness Effect	The desired effect of addition is captured by the premise $r' = r[Rd \mapsto r(Rd)+r(Rr)]$. It performs an addition of the two passed registers and stores the result in register Rd.

Adiw

$$\frac{\begin{array}{c}P(ep) = (\texttt{adiw}, (Rd, k)) \qquad r' = r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) + k] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s \mathsf{regpair\text{-}val}_{r'}(Rd) = 0] \\ cf = (\neg \mathsf{regpair\text{-}val}_{r'}(Rd)_{[15]} \wedge r(\mathsf{regconvert\text{-}u}(Rd))_{[7]})\end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{adiw})$$

Description	The `adiw` instruction takes a register pair and sums up its content with a provided immediate value k. The sum is stored in the first passed register.
Status Flags	C: $\big(\neg \mathsf{regpair\text{-}val}_{r'}(Rd)_{[15]} \wedge r(\mathsf{regconvert\text{-}u}(Rd))_{[7]}\big)$
Faithfulness Effect	The desired effect of addition is captured by the update to the registers $[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd)+k]$. It performs an addition of the passed register pairs content and the given immediate value k. The result is stored in the register pair with lower register Rd. The Z flag is captured by $[Z \mapsto_s \mathsf{regpair\text{-}val}_{r'}(Rd) = 0]$. It is the standard Z flag update, only extended to work with register pairs.

And

$$\frac{\begin{array}{c}P(ep) = (\texttt{and}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) \wedge r(Rr)] \\ sr' = sr[Z \mapsto_s r'(Rd) = 0]\end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{and})$$

Description	The `and` instruction performs a logical and operation between the given register contents. The result is stored in the first register.
Status Flags	None, except Z.
Faithfulness Effect	The logical and effect is captured by $[Rd \mapsto r(Rd) \wedge r(Rr)]$. The first passed register Rd is updated to the $\wedge$ operation with the contents of Rd and Rr as arguments.

Andi

$$\frac{P(ep) = (\texttt{andi}, (Rd, k)) \qquad r' = r[Rd \mapsto r(Rd) \wedge k] \qquad sr' = sr[Z \mapsto_s r'(Rd) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{andi})$$

Description	The `andi` instruction performs a logical and operation between the given register content and a given immediate value. The result is stored in the Rd register.
Status Flags	None, except Z.
Faithfulness Effect	The logical and effect is captured by $[Rd \mapsto r(Rd)\ \&\ k]$. The first passed register Rd is updated to the & operation with the contents of Rd and the immediate k as arguments.

Cp

$$\frac{P(ep) = (\texttt{cp}, (Rd, Rr)) \quad sr' = sr[Z \mapsto_s r(Rd) - r(Rr) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r, st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{cp})$$

Description	The `cp` instruction takes two registers and compares their contents. It sets the Z flag for further usage in e.g. branching operations. Register contents are not modified.
Status Flags	According to [2], a subtraction is performed and if the result is 0, the flag is set.
Faithfulness Effect	The register content is left untouched, as required by the description. To determine the value of Z, a subtraction is performed as instructed by the manual. If the result is 0, the flag is set, according to $[Z \mapsto_s r(Rd) - r(Rr) = 0]$.

Cpi

$$\frac{P(ep) = (\texttt{cpi}, (Rd, k)) \quad sr' = sr[Z \mapsto_s r(Rd) = k][C \mapsto_s |k| > |r(Rd)|]}{(sr, m, r, st, ep) \xrightarrow{\mathrm{t}(P(ep))}_P (sr', m, r, st, ep +_{\texttt{ep}} 1)} \ (\mathsf{cpi})$$

Description	The `cpi` instruction takes a registes and compares its content with a given immediate value. It sets the Z flag for further usage in e.g. branching operations depending on the outcome of this comparison. Additionally, the C flag is set depending on the sizes of compared values. Register contents are not modified.
Status Flags	According to [2], a subtraction is performed and if the result is 0, the Z flag is set. C: "Set if the absolute value of K is larger than the absolute value of Rd; cleared otherwise." [2]
Faithfulness Effect	The register content is left untouched, as required by the description. To determine the value of Z, the content of Rd is compared with the given immediate value k. A subtraction of those values if 0 if and only if they are equal. Thus, the use update $[Z \mapsto_s r(Rd) = k]$ correctly handles the Z flag. According to the supplied explanation of the C flag, it is sufficient to have a look at the absolute values. The explanation is directly captured by the update $[C \mapsto_s \|k\| > \|r(Rd)\|]$.

Cpc

$$\frac{P(ep) = (\texttt{cpc}, (Rd, Rr)) \quad sr' = sr[Z \mapsto_s r(Rd) - r(Rr) - sr(C) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathrm{t}(P(ep))}_P (sr', m, r, st, ep +_{\texttt{ep}} 1)} \ (\mathsf{cpc})$$

Description	The `cpc` instruction takes two registers and compares their contents, also taking the carry flag into account. It sets the Z flag for further usage in e.g. branching operations.
Status Flags	According to [2], a subtraction is performed and if the result is 0, the flag is set.
Faithfulness Effect	The register content is left untouched, as required by the description. To determine the value of Z, a subtraction, including the value of the C flag, is performed as instructed by the manual. If the result is 0, the flag is set, according to $[Z \mapsto_s r(Rd) - r(Rr) - sr(C) = 0]$.

Dec

$$\frac{P(ep) = (\texttt{dec}, (Rd)) \quad r' = r[Rd \mapsto r(Rd) - 1] \quad sr' = sr[Z \mapsto_s r'(Rd) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathrm{t}(P(ep))}_P (sr', m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{dec})$$

Description	The `dec` instruction takes a registers and decreases its content by one. The result is stored inside the Rd register.
Status Flags	None, except Z.
Faithfulness Effect	The decreasing operation is captured by performing the update $[Rd \mapsto r(Rd) - 1]$. It directly captures the informal description.

Eor

$$\frac{\begin{array}{c} P(ep) = (\texttt{eor}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) \oplus r(Rr)] \\ sr' = sr[Z \mapsto_s r'(Rd) = 0] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathrm{t}(P(ep))}_P (sr', m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{eor})$$

Description	The `eor` instruction takes two registers and performs an exclusive-or operation on their contents. The result is stored inside the Rd register.
Status Flags	None, except Z.
Faithfulness Effect	The exclusive-or operation is captured by updating the register's contents $[Rd \mapsto r(Rd) \oplus r(Rr)]$. Here, $\oplus$ is the exclusive-or operator.

Inc

$$\frac{P(ep) = (\texttt{inc}, (Rd)) \quad r' = r[Rd \mapsto r(Rd) + 1] \quad sr' = sr[Z \mapsto_s r'(Rd) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathrm{t}(P(ep))}_P (sr', m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{inc})$$

Description	The `inc` instruction takes a registers and increases its content by one. The result is stored inside the Rd register.
Status Flags	None, except Z.
Faithfulness Effect	The increasing operation is captured by performing the update $[Rd \mapsto r(Rd) + 1]$. It directly captures the informal description.

Lsr

$$\frac{P(ep) = (\texttt{lsr}, (Rd)) \qquad r' = r[Rd \mapsto r(Rd) \ggg 1] \\ sr' = sr[Z \mapsto_s r'(Rd) = 0][C \mapsto_s r(Rd) \bmod 2 = 1]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{lsr})$$

Description	The `lsr` instruction takes a register and performs a logical shift right. The result is stored in Rd.
Status Flags	C: $r(Rd)_{[0]}$
Faithfulness Effect	The logical shift right operation is captured by updating the register's contents $[Rd \mapsto r(Rd) \ggg 1]$. The $\ggg$ operator stands for the logical shift right. According to the provided formula, the C flag is set if the least significant bit is set. This is the case, if $r(Rd)$ is odd. Thus, the updated performed by $[C \mapsto_s r(Rd) \bmod 2 = 1]$ is correct.

Mul

$$\frac{P(ep) = (\texttt{mul}, (Rd, Rr)) \qquad r' = r[r_0 \mapsto_p r(Rd) \cdot r(Rr)] \\ sr' = sr[C \mapsto_s \mathsf{regpair\text{-}val}_{r'}(r_0) \ggg 15 \geq 1][Z \mapsto_s \mathsf{regpair\text{-}val}_{r'}(r_0) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{mul})$$

Description	The `mul` instruction takes two registers and performs a multiplication on their contents. The result is stored in r_0 and r_1.
Status Flags	$C : r'(r_1)_{[15]} = 1$ Z : The result is 0.
Faithfulness Effect	The exclusive-or operation is captured by updating the register's contents $[Rd \mapsto r(Rd) \oplus r(Rr)]$. Here, $\oplus$ is the exclusive-or operator. The carry flag, according to the given formula, is set when the 15-th bit of the result is set. The used update $[C \mapsto_s \mathsf{regpair\text{-}val}_{r'}(r_0) \ggg 15 \geq 1]$ does this by right shifting the result by 15. Note, that the value is 16 bit long, such that a right shift of 15 leaves only the rightmost bit. If this right shifted value is greater or equal 1, then the 15-th bit must have been set. Setting the Z flag is done according to the value found in the result register pair.

Neg

$$\frac{P(ep) = (\texttt{neg}, (Rd)) \qquad r' = r[Rd \mapsto -r(Rd)] \qquad sr' = sr[Z \mapsto_s r'(Rd) = 0][C \mapsto_s \neg(r'(Rd) = 0)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{neg})$$

Description	The `neg` instruction takes a registers and negates its value. The result is stored inside the Rd register.
Status Flags	C: "The C Flag will be set in all cases except when the contents of register Rd after operation is $00." [2].
Faithfulness Effect	The negation operation is captured by $[Rd \mapsto -r(Rd)]$. This is adequate, as performing a two's complement negation as intended by `neg`, leads to a simple negation in decimal notation. C and Z are set according to the result.

Or

$$\frac{P(ep) = (\texttt{or}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) \vee r(Rr)] \qquad sr' = sr[Z \mapsto_s r'(Rd) = 0]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{or})$$

Description	The `or` instruction takes two registers and performs an logical or operation on their contents. The result is stored in Rd.
Status Flags	None, except Z.
Faithfulness Effect	The or operation is captured by updating the register's contents $[Rd \mapsto r(Rd) \vee r(Rr)]$.

Ror

$$\frac{P(ep) = (\texttt{ror}, (Rd)) \qquad r' = r[Rd \mapsto (r(Rd) \ggg 1) + (sr(C) \lll 7)] \qquad sr' = sr[Z \mapsto_s r'(Rd) = 0][C \mapsto_s r(Rd) \bmod 2 = 1]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{ror})$$

Description	The `ror` instruction takes a register and performs a shift right. The most significant bit of the result is set to the C flag. The result is stored in Rd.
Status Flags	C: $r(Rd)_{[0]}$

Faithfulness The update is performed according to $[Rd \mapsto (r(Rd) \ggg 1) + (sr(C) \lll 7)]$. Here, $(r(Rd) \ggg 1)$ performs the right shift of the previous contents of register Rd. The carry flag is left shifted to the right by $(sr(C) \lll 7)$. By adding it, it can be placed directly to the most significant bit of the result.
Effect According to the provided formula, the C flag is set if the least significant bit is set. This is the case, if $r(Rd)$ is odd. Thus, the updated performed by $[C \mapsto_s r(Rd) \bmod 2 = 1]$ is correct.

Sbc

$$\frac{\begin{array}{c} P(ep) = (\texttt{sbc}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) - r(Rr) - sr(C)] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0 \wedge sr(Z) = 1] \\ cf = (\neg r(Rd)_{[7]} \wedge r(Rr)_{[7]}) \vee (r(Rr)_{[7]} \wedge r'(Rd)_{[7]}) \vee (r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{sbc})$$

Description The `sbc` instruction takes two registers and performs a a subtraction on their contents, including the carry flag. The result is stored in Rd.
Status Flags C: $\big(\neg r(Rd)_{[7]} \wedge r(Rr)_{[7]}\big) \vee \big(r(Rr)_{[7]} \wedge r'(Rd)_{[7]}\big) \vee \big(r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}\big)$
Faithfulness The subtraction operation is captured by $[Rd \mapsto r(Rd) - r(Rr) - sr(C)]$. The register Rd is updated according to the difference between the values of Rd, Rr and the C flag.
Effect

Sbci

$$\frac{\begin{array}{c} P(ep) = (\texttt{sbci}, (Rd, k)) \qquad r' = r[Rd \mapsto r(Rd) - k - sr(C)] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0 \wedge sr(Z) = 1] \\ cf = (\neg r(Rd)_{[7]} \wedge k_{[7]}) \vee (k_{[7]} \wedge r'(Rd)_{[7]}) \vee (r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{sbci})$$

Description The `sbci` instruction takes a register Rd and performs a subtraction with a given immediate value k and the C flag. The result is stored in register Rd.
Status Flags Z: $\neg r'(Rd)_{[7]} \wedge \ldots \wedge \neg r'(Rd)_{[0]} \wedge sr(Z)$
C: $\big(\neg r(Rd)_{[7]} \wedge k_{[7]}\big) \vee \big(k_{[7]} \wedge r'(Rd)_{[7]}\big) \vee \big(r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}\big)$

Faithfulness Effect	The subtraction operation is captured by $[Rd \mapsto r(Rd) - k - sr(C)]$. The register Rd is updated according to the difference between the values of the register Rd, the given immediate value k and the carry flag C.

Sub

$$\frac{\begin{array}{c} P(ep) = (\texttt{sub}, (Rd, Rr)) \qquad r' = r[Rd \mapsto r(Rd) - r(Rr)] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0] \\ cf = (\neg r(Rd)_{[7]} \wedge r(Rr)_{[7]}) \vee (r(Rr)_{[7]} \wedge r'(Rd)_{[7]}) \vee (r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{sub})$$

Description	The `sub` instruction takes two registers and performs a subtraction on their contents. The result is stored in register Rd.
Status Flags	C: $\big(\neg r(Rd)_{[7]} \wedge r(Rr)_{[7]}\big) \vee \big(r(Rr)_{[7]} \wedge r'(Rd)_{[7]}\big) \vee \big(r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}\big)$
Faithfulness Effect	The subtraction operation is captured by $[Rd \mapsto r(Rd) - r(Rr)]$. The register Rd is updated according to the difference between the values of registers Rd and Rr.

Subi

$$\frac{\begin{array}{c} P(ep) = (\texttt{subi}, (Rd, k)) \qquad r' = r[Rd \mapsto r(Rd) - k] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s r'(Rd) = 0] \\ cf = (\neg r(Rd)_{[7]} \wedge k_{[7]}) \vee (k_{[7]} \wedge r'(Rd)_{[7]}) \vee (r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{subi})$$

Description	The `subi` instruction takes a register Rd and performs a subtraction with a given immediate value k. The result is stored in register Rd.
Status Flags	C: $\big(\neg r(Rd)_{[7]} \wedge k_{[7]}\big) \vee \big(k_{[7]} \wedge r'(Rd)_{[7]}\big) \vee \big(r'(Rd)_{[7]} \wedge \neg r(Rd)_{[7]}\big)$
Faithfulness Effect	The subtraction operation is captured by $[Rd \mapsto r(Rd) - k]$. The register Rd is updated according to the difference between the values of the register Rd and the given immediate value k.

Sbiw

$$\frac{P(ep) = (\texttt{sbiw}, (Rd, k)) \qquad r' = r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) - k] \\ sr' = sr[C \mapsto_s cf][Z \mapsto_s \mathsf{regpair\text{-}val}_{r'}(Rd) = 0] \\ cf = (\mathsf{regpair\text{-}val}_{r'}(Rd)_{[15]} \wedge \neg r(\mathsf{regconvert\text{-}u}(Rd))_{[7]})}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{sbiw})$$

Description	The `sbiw` instruction takes a register pair Rd and performs a subtraction with a given immediate value k. The result is stored in register pair Rd.
Status Flags	C: $\left(\mathsf{regpair\text{-}val}_{r'}(Rd)_{[15]} \wedge \neg r(\mathsf{regconvert\text{-}u}(Rd))_{[7]}\right)$
Faithfulness Effect	The subtraction operation is captured by $[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) - k]$. The register pair Rd is updated according to the difference between the values of the register pair Rd and the given immediate value k. The Z flag is captured by $[Z \mapsto_s \mathsf{regpair\text{-}val}_{r'}(Rd) = 0]$. It is the standard Z flag update, only extended to work with register pairs.

Branching Instructions There are two rules for every branching instruction. One rule handles a branching where the condition for performing a branching operation is satisfied. Such rules are suffixed with -t. A second rule is in place to handle a situation where the branching condition is not satisfied and the program shall continue with the normal control flow. Such rules are suffixed with -f.

In general, the syntax of a branching instruction is $(\texttt{instr}, (epa))$. If the intended branching condition is satisfied, the next execution point is modified by rules suffixed with -t. The rule also takes care of extra time required by performing a branching operation. This extra time is represented by the `br` constant. It has been defined to equal one clock cycle. It is added to the usual execution time modeled by $t(P(ep))$. If the intended branching condition is not satisfied, the execution point is shifted forward as usual by applying $+_{\mathsf{ep}}1$ by the rules suffixed with -f.

A branching operation does not modify registers, memory or status registers. Thus, the functions used to store values of those data storages are not modified. This is captured by using the same assignment functions for those data storages in conclusions of semantic rules.

According to the instruction manual, branching as well as control flow instructions set the next execution point to be their argument, incremented by one (e.g. [2, p. 25]). We observe a difference between this description in the instruction manual and actually disassembled code. It is for example not possible, to jump to address 0 when the description as in the instruction manual is used. Our small-step semantics use our observed version of control flow behavior.

We provide detailed faithfulness arguments for every single small step semantic rule in the following.

brcc

$$\frac{P(ep) = (\texttt{brcc}, (epa)) \quad sr(C) = 0 \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))+\texttt{br}}_P (sr, m, r, st, ep')} \text{ (brcc-t)}$$

$$\frac{P(ep) = (\texttt{brcc}, (epa)) \quad sr(C) \neq 0}{(sr, m, r, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, ep +_{\texttt{ep}} 1)} \text{ (brcc-f)}$$

Description	The `brcc` instruction performs a branching, if the carry flag C is *cleared*.
Status Flags	No flags are modified.
Faithfulness Effect	A comparison between the carry flag C and 0 is performed by $sr(C) = 0$. If the carry flag has 0 as value, it is cleared and thus a branching is performed. The rule (brcc-t) is applicable in this case and sets the next execution point to the given one, leaving the call stack unmodified.. If the carry flag is not set, the rule (brcc-f) is applicable, as it tests for inequality with 0. It continues with the normal control flow.

brcs

$$\frac{P(ep) = (\texttt{brcs}, (epa)) \quad sr(C) \neq 0 \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))+\texttt{br}}_P (sr, m, r, st, ep')} \text{ (brcs-t)}$$

$$\frac{P(ep) = (\texttt{brcs}, (epa)) \quad sr(C) = 0}{(sr, m, r, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, ep +_{\texttt{ep}} 1)} \text{ (brcs-f)}$$

Description	The `brcs` instruction performs a branching, if the carry flag C is *set*.
Status Flags	No flags are modified.
Faithfulness Effect	A comparison between the carry flag C and 0 is performed by $sr(C) \neq 0$. If the carry flag is unequal to 0, it is set and thus a branching is performed. The rule (brcs-t) is applicable in this case and sets the next execution point to the given one, leaving the call stack unmodified. If the carry flag is set, the rule (brcs-f) is applicable, as it tests for equality with 0. It continues with the normal control flow.

breq

$$\frac{P(ep) = (\texttt{breq}, (epa)) \quad sr(Z) = 1 \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))+\mathtt{br}}_P (sr, m, r, st, ep')} \text{ (breq-t)}$$

$$\frac{P(ep) = (\texttt{breq}, (epa)) \quad sr(Z) \neq 1}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr, m, r, st, ep +_{\mathtt{ep}} 1)} \text{ (breq-f)}$$

Description	The `breq` instruction performs a branching, if the zero flag Z is *set*.
Status Flags	No flags are modified.
Faithfulness Effect	A comparison between the zero flag Z and 1 is performed by $sr(Z) = 1$. Upon equality, (breq-t) is applicable and sets the next execution point to the given one, leaving the call stack unmodified. (breq-f) is applicable otherwise and continues with the normal control flow.

brne

$$\frac{P(ep) = (\texttt{brne}, (epa)) \quad sr(Z) \neq 1 \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))+\mathtt{br}}_P (sr, m, r, st, ep')} \text{ (brne-t)}$$

$$\frac{P(ep) = (\texttt{brne}, (epa)) \quad sr(Z) = 1}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr, m, r, st, ep +_{\mathtt{ep}} 1)} \text{ (brne-f)}$$

Description	The `brne` instruction performs a branching, if the zero flag Z is *cleared*.
Status Flags	No flags are modified.
Faithfulness Effect	A comparison between the zero flag Z and 1 is performed by $sr(Z) = 1$. Upon inequality, (breq-t) is applicable and sets the next execution point to the given one, leaving the call stack unmodified. (breq-f) is applicable otherwise and continues with the normal control flow.

cpse

$$\frac{P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad r(Rd) = r(Rr) \quad \neg \mathsf{tw}_P(ep +_{\mathtt{ep}} 1)}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))+\mathtt{br}}_P (sr, m, r, st, ep +_{\mathtt{ep}} 2)} \text{ (cpse-t)}$$

$$\frac{P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad r(Rd) = r(Rr) \quad \mathsf{tw}_P(ep +_{\mathsf{ep}} 1)}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))+\texttt{2br}}_P (sr, m, r, st, ep +_{\mathsf{ep}} 2)} \ (\mathsf{cpse\text{-}t\text{-}2w})$$

$$\frac{P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad r(Rd) \neq r(Rr)}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r, st, ep +_{\mathsf{ep}} 1)} \ (\mathsf{cpse\text{-}f})$$

Description	In comparison to the other branching operations, `cpse` does not take a branching target as argument. It rather takes two registers and compares their contents. If the contents are equal, the next instruction is skipped. Depending on the skipped instruction's length, execution takes additional 1 or 2 cycles.
Status Flags	No flags are modified.
Faithfulness Effect	A comparison is performed by $r(Rd) = r(Rr)$. In rule (cpse-t) where this comparison has to be satisfied in order to be applicable, it is skipped to the next instruction by adding 2 to the current execution point *ep*. This skips the next instruction. In the case of inequality in rule (cpse-f), the control flow continues as normal. Timing is correctly captured, as (cpse-t) checks for the next instruction's length. If it is a two word instruction, (cpse-t-2w) has to be applied which takes the double `br` additionally to execute.

Control Flow Instructions All control flow instructions have in common that they manipulate the next execution point and set it to their argument value.

There are two types of control flow instructions that have been considered. The first type are function calls. The currently executed function is left and a new function execution is started with the given current data storage values. Once the function call is terminated, the execution point is set back to the execution point represented by the call, shifted forward by one. The second type are jumps that do only manipulate the next executed instruction. It is not returned to the jump source once the execution has terminated.

Control flow instructions do not modify register or status register contents.

We provide detailed faithfulness arguments for every single small step semantic rule in the following.

call & rcall

$$\frac{P(ep) = (\texttt{call}, (f', a')) \quad ep = (f, a, l) \quad ep' = (f', a', (f, a) +_{\mathsf{ep}} 1 :: l)}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r, st, ep')} \ (\mathsf{call})$$

$$\frac{P(ep) = (\texttt{rcall}, (f', a')) \quad ep = (f, a, l) \quad ep' = (f', a', (f, a) +_{\mathsf{ep}} 1 :: l)}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r, st, ep')} \ (\mathsf{rcall})$$

Description	A `call` instruction halts the execution of the currently executed function, starts a new function execution and after termination of the called function, the execution of the function before the call is resumed.
Status Flags	No flags are modified.
Faithfulness Effect	Rule (call) modifies the next execution pint. Its sets its function and address component according to its given argument. The usually next execution point is added on the call stack, such that a following `ret` instruction can find where to jump back to. The `rcall` is different to `call` only in terms of a limited argument space. Not the whole program memory is reachable by a `rcall` instruction. Not taking this into account in the rule modeling is faithful according to our assumptions. Thus, the same arguments as in the `call` instruction apply to `rcall` instruction.

jmp & rjmp

$$\frac{P(ep) = (\texttt{jmp}, (epa)) \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, st, ep')} \text{ (jmp)}$$

$$\frac{P(ep) = (\texttt{rjmp}, (epa)) \quad ep = (ep_0, fs) \quad ep' = (epa, fs)}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, st, ep')} \text{ (rjmp)}$$

Description	A `jmp` instruction modifies the control flow. The next instruction that is executed is set to the given argument *epa*.
Status Flags	No flags are modified.
Faithfulness Effect	The instruction takes an argument *epa*. This argument is set as the next execution point, performing a modification in the control flow as required. The call stack is left intact. The `rjmp` is different to `jmp` only in terms of a limited argument space. Not the whole program memory is reachable by a `rjmp` instruction. Not taking this into account in the rule modeling is faithful according to our assumptions. Thus, the same arguments as in the `jmp` instruction apply to `rjmp` instruction.

Stack Instructions The stack is direct part of the memory. The stack pointer register $\texttt{sp}_l$ and $\texttt{sp}_u$ mark the beginning of the stack inside the memory.

We discuss the faithfulness of `push` and `pop` instructions which handle reading and writing to the stack in detail in the following.

push

$$\frac{P(ep) = (\texttt{push}, (Rd)) \qquad st' = r(Rd) :: st \qquad r' = r[\mathsf{sp}_l \mapsto_p \mathsf{regpair\text{-}val}_r(\mathsf{sp}_l) - 1]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st', ep +_{\mathsf{ep}} 1)} \ (\mathsf{push})$$

Description	The `push` instruction takes a register as argument. Its content is stored at the memory address the stack pointer currently points at. Afterwards, the stack pointer is decremented by 1.
Status Flags	No flags are modified.
Faithfulness	There are two steps that have to be performed.
Effect	At first, the value of register Rd has to be stored on the stack. This is done by modifying the memory according to $[\mathsf{regpair\text{-}val}_r(\mathsf{sp}_l) \mapsto r(Rd)]$. The memory address the stack pointer is pointing at, represented by $\mathsf{regpair\text{-}val}_r(\mathsf{sp}_l)$, is updated to the contents of register Rd. In a second step, the stack pointer is decremented by 1. This is done by updating the register's contents according to $[\mathsf{sp}_l \mapsto_p \mathsf{regpair\text{-}val}_r(\mathsf{sp}_l) - 1]$. The register pair sp_l is decremented by 1. All other components of the state, except the execution point, are left unmodified.

pop

$$\frac{P(ep) = (\texttt{pop}, (Rd)) \qquad st = x :: xs \qquad st' = xs \qquad r'' = r[Rd \mapsto x] \qquad r' = r''[\mathsf{sp}_l \mapsto_p \mathsf{regpair\text{-}val}_r(\mathsf{sp}_l) + 1]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st', ep +_{\mathsf{ep}} 1)} \ (\mathsf{pop})$$

Description	The `pop` instruction increments the stack pointer by 1. Afterwards, it reads the topmost entry using the updated stack pointer and stores its content in the given Rd register.
Status Flags	No flags are modified.
Faithfulness	The topmost entry is read and its content stored in register Rd. This is done by the assignment $r'' = r[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(\mathsf{sp}_l))]$. Reading from the memory is performed by $m(\mathsf{regpair\text{-}val}_r(\mathsf{sp}_l))$ which concatenates the stack pointer register pair and reads its value from the register assignment r. The result of this read is stored in register Rd.
Effect	The stack pointer is incremented by the update $[\mathsf{sp}_l \mapsto_p \mathsf{regpair\text{-}val}_r(\mathsf{sp}_l)+1]$. The stack pointer register pair, represented by sp_l, is increased by one. All other components of the state, except the execution point, are left unmodified.

Memory Instructions Instructions for reading from and writing into memory are fairly similar when it comes to their inner workings. The `ld` and `st` instructions for reading and writing, respectively, take two registers and a modifier. One of the registers holds the memory address that is read from or written to, the other holds the data that has to be written or serves as destination for the read data. The modifier allows the modification of the memory address that has been used. Modifier # leaves the memory address unchanged, + post-increments it, and − pre-decrements it. Instructions `ldd` and `std` allow the passing of an immediate value to be added when determining the memory address, instead of a modifier.

We provide detailed faithfulness arguments for every single small step semantic rule in the following.

Ld

$$\frac{P(ep) = (\texttt{ld}, (Rd, Rr, \#)) \quad r' = r[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(Rr))]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{ld\text{-}nm})$$

$$\frac{\begin{array}{c} P(ep) = (\texttt{ld}, (Rd, Rr, +)) \\ r'' = r[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(Rr))] \\ r' = r''[Rr \mapsto_p \mathsf{regpair\text{-}val}_{r''}(Rr) + 1] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{ld\text{-}post\text{-}inc})$$

$$\frac{\begin{array}{c} P(ep) = (\texttt{ld}, (Rd, Rr, -)) \\ r'' = r[Rr \mapsto_p \mathsf{regpair\text{-}val}_r(Rr) - 1] \\ r' = r''[Rd \mapsto m(\mathsf{regpair\text{-}val}_{r''}(Rr))] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{ld\text{-}pre\text{-}dec})$$

Description The `ld` instruction takes two registers and a modifier. The first register Rd is the destination register for the read value. The second register Rr is treated as pair and represents the memory address that is being read from. It is also the register that is being influenced by modifiers.

Status Flags No flags are modified.

Faithfulness In all instructions, reading from memory and writing to register Rd is handled by the update $[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(Rr))]$. If a pre-decrement is performed, as in (ld-pre-dec), the modified register assignment r'' is used to read the memory address.

Effect Rule (ld-nm) handles the # modifier that does not modify the Rr register.

Rule (ld-post-inc) handles the + modifier that performs a post-increment of Rr. This is performed by the update $r''[Rr \mapsto_p \mathsf{regpair\text{-}val}_{r''}(Rr) + 1]$, where r'' is the register assignment after the memory has been read.

Rule (ld-pre-dec) handles the − modifier that performs a pre-decrement of Rr. It is performed by the assignment $r'' = r[Rr \mapsto_p \mathsf{regpair\text{-}val}_r(Rr) - 1]$. When reading from the memory, the register assignment r'' is used to get the memory address that is being read from, correctly capturing that it is a pre-decrement.

St

$$\frac{P(ep) = (\mathtt{st}, (Rd, Rr, \#)) \quad m' = m[\mathsf{regpair\text{-}val}_r(Rd) \mapsto r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m', r, st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{st\text{-}nm})$$

$$\frac{\begin{array}{c} P(ep) = (\mathtt{st}, (Rd, Rr, +)) \\ m' = m[\mathsf{regpair\text{-}val}_r(Rd) \mapsto r(Rr)] \\ r' = r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) + 1] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m', r', st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{st\text{-}post\text{-}inc})$$

$$\frac{\begin{array}{c} P(ep) = (\mathtt{st}, (Rd, Rr, -)) \\ r' = r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) - 1] \\ m' = m[\mathsf{regpair\text{-}val}_{r'}(Rd) \mapsto r(Rr)] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m', r', st, epa +_{\mathsf{ep}} 1)} \ (\mathsf{st\text{-}pre\text{-}dec})$$

Description The `st` instruction takes two registers and a modifier. The first register Rd is treated as pair and used as memory address that is being written to. It is also the register that is being influenced by modifiers. The second register Rr contains the value that is being written into the memory.

Status Flags No flags are modified.

Faithfulness In all instructions, reading from memory and writing to register Rd is handled by the update $[\mathsf{regpair\text{-}val}_r(Rd) \mapsto r(Rr)]$ to m

Effect The current content of register Rr is read by $r(Rr)$ and is stored in the memory at the address represented by the register pair in Rd. If a pre-decrement is performed, as in (st-pre-dec), the modified register assignment r' is used to read the memory address which holds the memory address after the decrement operation. Rule (st-nm) handles the # modifier that does not modify the Rr register.

Rule (st-post-inc) handles the + modifier that performs a post-increment of Rd. This is performed by the update $r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) + 1]$.

Rule (st-pre-dec) handles the − modifier that performs a pre-decrement of Rd. It is performed by the update $[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rd) - 1]$. When writing to the memory, the register assignment r' is used to get the memory address that is being written to, correctly capturing that it is a pre-decrement.

Ldd

$$\frac{P(ep) = (\texttt{ldd}, (Rd, Rr, k)) \quad r' = r[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(Rr) + k)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, epa +_{\mathtt{ep}} 1)} \ (\mathsf{ldd})$$

Description The `ldd` instruction takes two registers and an immediate value. The first register Rd is the destination register for the read value. The value is being read from the memory. The memory address that is being read is determined by the register pair represented by Rr and the immediate value k.

Status Flags No flags are modified.

Faithfulness Reading from memory and storing in Rd is performed by the update $[Rd \mapsto m(\mathsf{regpair\text{-}val}_r(Rr) + k)]$ to the registers. The memory address that is being read is determined by $\mathsf{regpair\text{-}val}_r(Rr) + k$. This correctly captures the base address read from register pair Rr, summed up with the given immediate value.

Effect Other components of the state are not modified.

Std

$$\frac{P(ep) = (\texttt{std}, (Rd, Rr, k)) \quad m' = m[\mathsf{regpair\text{-}val}_r(Rd) + k \mapsto r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m', r, st, epa +_{\mathtt{ep}} 1)} \ (\mathsf{std})$$

Description The `std` instruction takes two registers and an immediate value. The first register Rd represents the register pair that serves as base address. To determine the memory address to which the value in Rr is stored, the base address is summed up with the given immediate value k.

Status Flags No flags are modified.

Faithfulness Effect Writing to memory is performed by the update $[\mathsf{regpair\text{-}val}_r(Rd) + k \mapsto r(Rr)]$ to m. The address that is being written to is calculated by $\mathsf{regpair\text{-}val}_r(Rd) + k$. The value of the register pair represented by Rd is added with k, correctly capturing the memory address calculation. The contents of this memory address is replaced with the contents of register Rr. Other components of the state are not modified.

Input/Output Instructions The `in` and `out` instructions are to read and write data from some special locations such as ports on a microcontroller or specific configuration registers. In our setting, they are used for direct access to the stack pointer and the status register flags.

Both instructions take a register that is either destination or source. In addition, an identifier for the location is taken. Exact values for those identifiers are not to be found in [2]. We rely on the register summary in [1, p. 402].

Considered locations are:

- `0x3d`: Contents of $\mathtt{sp}_l$.
- `0x3e`: Contents of $\mathtt{sp}_u$.
- `0x3f`: A combination of all status registers, where bit 0 is the C flag and bit 1 is the Z flag.

We provide detailed faithfulness arguments for every single small step semantic rule in the following.

In

$$\frac{P(ep) = (\mathtt{in}, (Rd, 0\text{x}3\text{d})) \quad r' = r[Rd \mapsto r(\mathtt{sp}_l)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{in\text{-}sp\text{-}l})$$

$$\frac{P(ep) = (\mathtt{in}, (Rd, 0\text{x}3\text{e})) \quad r' = r[Rd \mapsto r(\mathtt{sp}_u)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{in\text{-}sp\text{-}u})$$

$$\frac{P(ep) = (\mathtt{in}, (Rd, 0\text{x}3\text{f})) \quad r' = r[Rd \mapsto sr(C) + (sr(Z) \lll 1)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{in\text{-}stat})$$

Description The `in` instruction takes a register Rd and an immediate value representing a special location. Register Rd shall be set according to the value found in the special location.

Status Flags No flags are modified.

Faithfulness All rules have in common, that Rd is updated according to $[Rd \mapsto x]$ where x is set according to the given identifier.

Effect Values for x are:

- `0x3d`: It is set to $r(\mathtt{sp}_l)$, as required by the identifier.
- `0x3e`: It is set to $r(\mathtt{sp}_u)$, as required by the identifier.
- `0x3f`: The leftmost bit is set to the carry flag C. The following bit has to be set to the zero flag Z. This is done by left-shifting $sr(Z)$ by one and adding it to the carry flag.

Out

$$\frac{P(ep) = (\mathtt{out}, (\text{0x3d}, Rr)) \quad r' = r[\mathtt{sp}_l \mapsto r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr, m, r', st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{out\text{-}sp\text{-}l})$$

$$\frac{P(ep) = (\mathtt{out}, (\text{0x3e}, Rr)) \quad r' = r[\mathtt{sp}_u \mapsto r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr, m, r', st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{out\text{-}sp\text{-}u})$$

$$\frac{\begin{array}{c} P(ep) = (\mathtt{out}, (\text{0x3f}, Rr)) \\ sr' = sr[C \mapsto r(Rr)_{[0]}][Z \mapsto r(Rr)_{[1]}] \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr', m, r, st, ep +_{\mathtt{ep}} 1)} \ (\mathsf{out\text{-}stat})$$

Description The `out` instruction takes a register Rr and an immediate value representing a special location. The special location's value is set to the content of register Rr.

Status Flags No flags are modified.

Faithfulness Rules (out-sp-l) and (out-sp-u) straightforwardly set the contents

Effect of the stack pointer registers to the value of the given register by updating the register assignment according to $[\mathtt{sp}_l \mapsto r(Rr)]$ and $[\mathtt{sp}_u \mapsto r(Rr)]$, respectively.

Rule (out-stat) shall update flags Z and C to the contents of the bits 1 and 0 of Rr. The idea behind the rules is to force all other bits to 0 and shift it according to the position.

Miscellaneous Instructions There are a few instructions missing for whose semantic rules we have not yet introduced and given faithfulness arguments for. They are handled in the following.

Mov & Movw

$$\frac{P(ep) = (\texttt{mov}, (Rd, Rr)) \quad r' = r[Rd \mapsto r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{mov})$$

$$\frac{P(ep) = (\texttt{movw}, (Rd, Rr)) \quad r' = r[Rd \mapsto_p \mathsf{regpair\text{-}val}_r(Rr)]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{movw})$$

Description	The `mov` and `movw` instructions move a value from the first register (pair) Rd to the second register (pair) Rr. It does not modify and status register flags.
Status Flags	No flags are modified.
Faithfulness	Moving contents of registers is done straightforward by the update $[Rd \mapsto r(Rr)]$ to r. The register Rd is overwritten by the contents of Rr, as required by the description.
Effect	For words, pair updating and reading is used.
	Other components of the state are not modified.

Ldi

$$\frac{P(ep) = (\texttt{ldi}, (Rd, k)) \quad r' = r[Rd \mapsto k]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr, m, r', st, ep +_{\texttt{ep}} 1)} \ (\mathsf{ldi})$$

Description	The `ldi` instruction takes an immediate value k and a register Rd as arguments. The register Rd is set to the value of k.
Status Flags	No flags are modified.
Faithfulness	Setting the register Rd to contain k is done straightforward by
Effect	the assignment $r' = r[Rd \mapsto k]$.
	Other components of the state are not modified.

Clc

$$\frac{P(ep) = (\texttt{clc}, ()) \quad sr' = sr[C \mapsto 0]}{(sr, m, r, st, ep) \xrightarrow{\mathsf{t}(P(ep))}_P (sr', m, r, st, ep +_{\texttt{ep}} 1)} \ (\mathsf{clc})$$

Description	The `clc` instruction takes no arguments. It resets the carry flag C to 0.
Status Flags	C: 0
Faithfulness Effect	The C flag is statically set to 0 by performing the assignment $sr' = sr[C \mapsto 0]$.

Cli

$$\frac{P(ep) = (\texttt{cli}, ())}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, st, ep +_{\texttt{ep}} 1)} \ (\mathsf{cli})$$

Description	The `cli` instruction takes no arguments. It resets the interrupt flag I to 0.
Status Flags	No flags are modified (the interrupt flag is out of scope in this work).
Faithfulness Effect	No effect (the interrupt flag is out of scope in this work).

B Complete Definition of Type System

B.1 Security Property

We use security domains $\mathcal{H}$ and $\mathcal{L}$ to denote confidential and public data, respectively. The security lattice allows $\mathcal{L} \sqsubseteq \mathcal{H}$, where $\sqcup$ represents the least upper bound operation.

Definition 33 (Register domain assignment). *A register domain assignment is a function out of the set*

$$\texttt{REG-DA} := \texttt{REG} \to \{\mathcal{L}, \mathcal{H}\}$$

Definition 34 (Status register domain assignment). *A status register domain assignment is a function out of the set*

$$\texttt{STAT-DA} := \{C, Z\} \to \{\mathcal{L}, \mathcal{H}\}$$

Definition 35 (Flows to relation for domain assignments). *Let A be a set. The relation $\sqsubseteq \subseteq (A \to \{\mathcal{L}, \mathcal{H}\})^2$, is given by*

$$(a \sqsubseteq b) \Leftrightarrow \forall x \in \mathsf{dom}(a) : a(x) \sqsubseteq b(x)$$

Confidentiality of registers and status registers are expressed by domain assignments mapping identifiers for (status) registers to security domains.

Definition 36 (Stack domain assignment). *The set of possible stack domain assignments is given by*

$$\texttt{STACK-DA} := \{\mathcal{L}, \mathcal{H}\}^*$$

The confidentiality of values on the stack are modeled by a list of security domains that correspond to elements on the stack.

Definition 37 (Indistinguishability of stack assignments). *The indistinguishability relation of stack assignments $\simeq_{\texttt{sda}} \subseteq \texttt{STACK-VAL} \times \texttt{STACK-VAL}$ for $\texttt{sda} \in \texttt{STACK-DA}$ is the smallest relation, such that the following properties are satisfied:*

1. $xs \simeq_{[]} zs$
2. $[] \simeq_{ys} []$
3. $xs \simeq_{ys} zs \Rightarrow x :: xs \simeq_{\mathcal{L}::ys} x :: zs$
4. $xs \simeq_{ys} zs \Rightarrow x :: xs \simeq_{\mathcal{H}::ys} z :: zs$
5. $(xs \simeq_{ys} zs \wedge ys \in \{\mathcal{H}\}^*) \Rightarrow xs \simeq_{\mathcal{H}::ys} z :: zs$
6. $(xs \simeq_{ys} zs \wedge ys \in \{\mathcal{H}\}^*) \Rightarrow x :: xs \simeq_{\mathcal{H}::ys} zs$

for all $xs, zs \in \texttt{STACK-VAL}$, $x, z \in \texttt{VAL}_8$ and $ys \in \texttt{STACK-DA}$.

Definition 38 (Indistinguishability of functions). *Let A and B be sets, $f, f' : A \to B$ be two functions and $ass : A \to \{\mathcal{L}, \mathcal{H}\}$ be a domain assignment. Functions f and f' are indistinguishable with respect to ass, written $f \approx_{ass} f'$, if and only if*

$$\forall x \in \mathsf{dom}(ass) : ass(x) = \mathcal{L} \Rightarrow f(x) = f'(x)$$

Definition 39 (Indistinguishability of states). *Let* (sr, m, r, st, ep), $(sr', m', r', st', ep') \in \texttt{STATE}$ *be two states. Let* $\texttt{srda} \in \texttt{STAT-DA}$, $\texttt{md} \in \{\mathcal{L}, \mathcal{H}\}$, $\texttt{rda} \in \texttt{REG-DA}$ *and* $\texttt{sda} \in \texttt{STACK-DA}$ *be domain assignments. The states are indistinguishable with respect to* srda, md, rda *and* sda, *written*

$$(sr, m, r, st, ep) \approx_{\texttt{sda,md,rda,srda}} (sr', m', r', st', ep')$$

if and only if $sr \approx_{\texttt{srda}} sr'$, $\texttt{md} = \mathcal{L} \Rightarrow m = m'$ $r \approx_{\texttt{rda}} r'$, *and* $st \simeq_{\texttt{sda}} st'$.

We express capabilities of the attacker through an indistinguishability relation on states. He is allowed to observe all $\mathcal{L}$ annotated data. Indistinguishability of stack assignments also allows him to probe for the height of the stack.

Definition 40 (TSNI). *Let* $P \in \texttt{PROG}$ *be a program. Program* P *satisfies TSNI starting from* $\texttt{ep}_s \in \texttt{EPS}$ *with initial domain assignments* $\texttt{srda} \in \texttt{STAT-DA}$, $\texttt{md} \in \{\mathcal{L}, \mathcal{H}\}$, $\texttt{rda} \in \texttt{REG-DA}$ *and* $\texttt{sda} \in \texttt{STACK-DA}$ *and finishing domain assignments* $\texttt{srda}' \in \texttt{STAT-DA}$, $\texttt{md}' \in \{\mathcal{L}, \mathcal{H}\}$, $\texttt{rda}' \in \texttt{REG-DA}$ *and* $\texttt{sda}' \in \texttt{STACK-DA}$ *if and only if*

$$\begin{aligned}
&\forall s_0, s_0', s_1, s_1' \in \texttt{STATE}\colon \forall n, n' \in \mathbb{N}\colon \\
(1)\quad &\mathsf{epselect}(s_0) = \texttt{ep}_s \wedge \mathsf{epselect}(s_0') = \texttt{ep}_s \wedge \\
(2)\quad &s_0 \approx_{\texttt{sda,md,rda,srda}} s_0' \wedge \\
(3)\quad &(s_0, s_1) \in \Downarrow_P^n \wedge (s_0', s_1') \in \Downarrow_P^{n'} \\
(4)\quad &\quad \Rightarrow s_1 \approx_{\texttt{sda}',\texttt{md}',\texttt{rda}',\texttt{srda}'} s_1' \wedge n = n'
\end{aligned}$$

Our security property requires two terminating executions starting in indistinguishable states to terminate in still indistinguishable states after the same amount of clock cycles. Thus, the attacker is additionally able to observe the exact required execution time.

B.2 Used Concepts in Type System

Definition 41 (Lift). *The function* $\mathsf{lift} : (\texttt{STACK-DA} \times \{\mathcal{L}, \mathcal{H}\}) \to \texttt{STACK-DA}$ *is recursively defined as follows:*

$$\begin{aligned}
\mathsf{lift}([], d) &:= [] \\
\mathsf{lift}(x :: xs, d) &:= (x \sqcup d) :: \mathsf{lift}(xs, d)
\end{aligned}$$

The lift function raises security domains in a given list to a least upper bound.

Definition 42 (Successor relation). *Let* P *be a program. The successor relation* $\rightsquigarrow_P \subseteq \texttt{EPS} \times \texttt{EPS}$ *is defined such that for all* $\texttt{ep}_1$, $\texttt{ep}_2 \in \texttt{EPS}$ *it holds that* $\texttt{ep}_1 \rightsquigarrow_P \texttt{ep}_2$ *if and only if*

$$\exists s_1, s_2 \in \texttt{STATE} : \exists c \in \mathbb{N} : s_1 \xrightarrow{c}_P s_2 \wedge \mathsf{epselect}(s_1) = \texttt{ep}_1 \wedge \mathsf{epselect}(s_2) = \texttt{ep}_2$$

Definition 43 (Control dependence regions). *Let P be a program. The functions* $\mathsf{region}^1_P, \mathsf{region}^2_P : \mathtt{EPS} \to \mathcal{P}(\mathtt{EPS})$ *and* $\mathsf{jun}_P : \mathtt{EPS} \rightharpoonup \mathtt{EPS}$ *are a safe over approximation of the program's control dependence regions if they satisfy the safe over approximation properties (SOAPs):*

SOAP1 *For all execution points* $\mathtt{ep}_1, \mathtt{ep}_2, \mathtt{ep}_3 \in \mathtt{EPS}$ *such that* $\mathtt{ep}_1 \rightsquigarrow_P \mathtt{ep}_2$, $\mathtt{ep}_1 \rightsquigarrow_P \mathtt{ep}_3$ *and* $\mathtt{ep}_2 \neq \mathtt{ep}_3$ *exactly one of the following holds*

- $\mathtt{ep}_2 \in \mathsf{region}^1_P(\mathtt{ep}_1)$ *and* $\mathtt{ep}_3 \in \mathsf{region}^2_P(\mathtt{ep}_1)$
- $\mathtt{ep}_2 \in \mathsf{region}^2_P(\mathtt{ep}_1)$ *and* $\mathtt{ep}_3 \in \mathsf{region}^1_P(\mathtt{ep}_1)$
- $\mathtt{ep}_2 \in \mathsf{region}^1_P(\mathtt{ep}_1)$ *and* $\mathsf{jun}_P(\mathtt{ep}_1) = \mathtt{ep}_3$
- $\mathtt{ep}_3 \in \mathsf{region}^1_P(\mathtt{ep}_1)$ *and* $\mathsf{jun}_P(\mathtt{ep}_1) = \mathtt{ep}_2$

SOAP2 *For all execution points* $\mathtt{ep} \in \mathtt{EPS}$ *it holds that* $\mathsf{region}^1_P(\mathtt{ep}) \cap \mathsf{region}^2_P(\mathtt{ep}) = \emptyset$.

SOAP3 *For all execution points* $\mathtt{ep}_1, \mathtt{ep}_2, \mathtt{ep}_3 \in \mathtt{EPS}$, *for all* $i \in \{1, 2\}$, *if* $\mathtt{ep}_2 \in \mathsf{region}^i_P(\mathtt{ep}_1)$ *and* $\mathtt{ep}_2 \rightsquigarrow_P \mathtt{ep}_3$, *then either* $\mathtt{ep}_3 \in \mathsf{region}^i_P(\mathtt{ep}_1)$ *or* $\mathsf{jun}_P(\mathtt{ep}_1) = \mathtt{ep}_3$.

SOAP4 *For all execution points* $\mathtt{ep}_1, \mathtt{ep}_2 \in \mathtt{EPS}$, *for all* $i \in \{1, 2\}$, *if* $\mathtt{ep}_2 \in \mathsf{region}^i_P(\mathtt{ep}_1)$ *and there exists no* $\mathtt{ep}_3 \in \mathtt{EPS}$ *such that* $\mathtt{ep}_2 \rightsquigarrow_P \mathtt{ep}_3$, *then* $\mathsf{jun}_P(\mathtt{ep}_1)$ *is undefined.*

The function $\mathsf{region}_P : \mathtt{EPS} \to \mathcal{P}(\mathtt{EPS})$ *is given by*

$$\mathsf{region}_P(\mathtt{ep}) := \mathsf{region}^1_P(\mathtt{ep}) \cup \mathsf{region}^2_P(\mathtt{ep})$$

Definition 44 (Then and else regions). *Let $P \in \mathtt{PROG}$ be a program. The functions* $\mathsf{region}^{then}_P, \mathsf{region}^{else}_P : \mathtt{EPS} \to \mathcal{P}(\mathtt{EPS})$ *are defined as follows*

$$\mathsf{region}^{then}_P(\mathtt{ep}) := \begin{cases} \mathsf{region}^1_P(\mathtt{ep}) & \text{if the branch target at } \mathtt{ep} \text{ is in } \mathsf{region}^1_P(\mathtt{ep}) \\ \mathsf{region}^2_P(\mathtt{ep}) & \text{else} \end{cases}$$

$$\mathsf{region}^{else}_P(\mathtt{ep}) := \begin{cases} \mathsf{region}^2_P(\mathtt{ep}) & \text{if the branch target at } \mathtt{ep} \text{ is in } \mathsf{region}^1_P(\mathtt{ep}) \\ \mathsf{region}^1_P(\mathtt{ep}) & \text{else} \end{cases}$$

Definition 45 (Security environment). *A security environment is a function* $se : \mathtt{EPS} \to \{\mathcal{L}, \mathcal{H}\}$. *A security environment se is* smaller *than another security environment se' if and only if $se \sqsubseteq se'$.*

Control dependence regions model dependence between execution points. If $\mathtt{ep}_1 \in \mathsf{region}_P(\mathtt{ep}_2)$, then the execution of $\mathtt{ep}_1$ depends on the outcome of a branching of $\mathtt{ep}_2$. As we care for the branches separately to determine their execution time in the following, they are separated into "then" and "else" branches. If then $\mathtt{ep}_2$ operates on high data, the security environment of $\mathtt{ep}_1$ has to be set to high. In the following we only consider functions region^{then}_P and region^{else}_P that satisfy the SOAPs.

Definition 46 (Branchtime). *Let $P \in \mathtt{PROG}$ be a program. The functions* $\mathsf{branchtime}^r_P : \mathtt{EPS} \to \mathbb{N}$ *for* $r \in \{then, else\}$ *are defined as follows*

$$\mathsf{branchtime}^r_P(\mathsf{ep}) := \sum_{\substack{\mathsf{ep}_i \in \mathsf{region}^r_P(\mathsf{ep}) \\ \mathsf{ep}_i \neq \mathsf{ep}}} \Big(\mathsf{t}(P(\mathsf{ep}_i)) - \mathsf{branchtime}^{then}_P(\mathsf{ep}_i) \Big)$$

The $\mathsf{branchtime}^r_P$ function calculates the execution time a branch requires. It is the key to verify the absence of timing variations in high branchings.

Definition 47 (Loop detection). *Let $P \in \mathtt{PROG}$ be a program. The predicate $\mathsf{loop}_P : \mathtt{EPS} \to \mathbb{B}$ is defined as*

$$\mathsf{loop}_P(\mathsf{ep}) := \exists \mathsf{ep}' \in \mathsf{region}_P(\mathsf{ep}) : \mathsf{ep} \rightsquigarrow^+_P \mathsf{ep}' \text{ is a back edge}$$

The predicate loop allows the detection of loops inside the control flow of a program. Later on, this predicate is used to forbid loops where the number of iteration depends on high data.

Definition 48 (Reachability). *Let $P \in \mathtt{PROG}$ be a program. Then the function $\mathsf{reachable}_P : \mathtt{EPS} \to \mathcal{P}(\mathtt{EPS})$ is defined by*

$$\mathsf{reachable}_P(\mathsf{ep}) := \{\mathsf{ep}' \in \mathtt{EPS} \mid \mathsf{ep} \rightsquigarrow^+_P \mathsf{ep}'\}$$

B.3 Typable Programs

We use typing judgments of the form

$$P, \mathsf{region}^{then}_P, \mathsf{region}^{else}_P, se, \mathsf{ep}_i : (\mathtt{sda}_{\mathsf{ep}_i}, \mathtt{md}_{\mathsf{ep}_i}, \mathtt{rda}_{\mathsf{ep}_i}, \mathtt{srda}_{\mathsf{ep}_i}) \vdash (\mathtt{sda}_{\mathsf{ep}_j}, \mathtt{md}_{\mathsf{ep}_j}, \mathtt{rda}_{\mathsf{ep}_j}, \mathtt{srda}_{\mathsf{ep}_j})$$

that relates domain assignments $(\mathtt{sda}_{\mathsf{ep}_i}, \mathtt{md}_{\mathsf{ep}_i}, \mathtt{rda}_{\mathsf{ep}_i}, \mathtt{srda}_{\mathsf{ep}_i})$ before the execution of instruction $P(\mathsf{ep}_i)$ to domain assignments $(\mathtt{sda}_{\mathsf{ep}_j}, \mathtt{md}_{\mathsf{ep}_j}, \mathtt{rda}_{\mathsf{ep}_j}, \mathtt{srda}_{\mathsf{ep}_j})$ after its execution.

In the following, we use the abbreviation

$$P, \cdots, \mathsf{ep}_i : (\mathtt{sda}_{\mathsf{ep}_i}, \mathtt{md}_{\mathsf{ep}_i}, \mathtt{rda}_{\mathsf{ep}_i}, \mathtt{srda}_{\mathsf{ep}_i}) \vdash (\mathtt{sda}_{\mathsf{ep}_j}, \mathtt{md}_{\mathsf{ep}_j}, \mathtt{rda}_{\mathsf{ep}_j}, \mathtt{srda}_{\mathsf{ep}_j})$$

instead of the full judgment.

Definition 49 (Typable programs). *Let $P \in \mathtt{PROG}$ be a program with control dependence regions $\mathsf{region}^{then}_P, \mathsf{region}^{else}_P$. The program is typable with starting execution point ep_s, initial domain assignments $\mathtt{sda}_{\mathsf{ep}_s}$, $\mathtt{md}_{\mathsf{ep}_s}$, $\mathtt{rda}_{\mathsf{ep}_s}$, $\mathtt{srda}_{\mathsf{ep}_s}$, final domain assignments $\mathtt{sda}_{\mathsf{ep}_{final}}$, $\mathtt{md}_{\mathsf{ep}_{final}}$, $\mathtt{rda}_{\mathsf{ep}_{final}}$, $\mathtt{srda}_{\mathsf{ep}_{final}}$, and security environment se, if and only if for every $\mathsf{ep}' \in \mathsf{reachable}_P(\mathsf{ep}_s)$ there exist domain assignments $\mathtt{sda}_{\mathsf{ep}'}$, $\mathtt{md}_{\mathsf{ep}'}$, $\mathtt{rda}_{\mathsf{ep}'}$, $\mathtt{srda}_{\mathsf{ep}'}$ such that*

1. *for all* $\mathtt{ep}_i, \mathtt{ep}_j \in \mathsf{reachable}_P(\mathtt{ep}_s) \cup \{\mathtt{ep}_s\}$, *if* $\mathtt{ep}_i \rightsquigarrow_P \mathtt{ep}_j$ *then there exists* $\mathtt{sda}'_{\mathtt{ep}_j}$, $\mathtt{md}'_{\mathtt{ep}_j}$, $\mathtt{rda}'_{\mathtt{ep}_j}$, $\mathtt{srda}'_{\mathtt{ep}_j}$ *such that* $\mathtt{sda}'_{\mathtt{ep}_j} \sqsubseteq_s \mathtt{sda}_{\mathtt{ep}_j}$, $\mathtt{md}'_{\mathtt{ep}_j} \sqsubseteq \mathtt{md}_{\mathtt{ep}_j}$, $\mathtt{rda}'_{\mathtt{ep}_j} \sqsubseteq \mathtt{rda}_{\mathtt{ep}_j}$, $\mathtt{srda}'_{\mathtt{ep}_j} \sqsubseteq \mathtt{srda}_{\mathtt{ep}_j}$ *and the judgment*

$$P, \cdots, \mathtt{ep}_i : (\mathtt{sda}_{\mathtt{ep}_i}, \mathtt{md}_{\mathtt{ep}_i}, \mathtt{rda}_{\mathtt{ep}_i}, \mathtt{srda}_{\mathtt{ep}_i}) \vdash (\mathtt{sda}'_{\mathtt{ep}_j}, \mathtt{md}'_{\mathtt{ep}_j}, \mathtt{rda}'_{\mathtt{ep}_j}, \mathtt{srda}'_{\mathtt{ep}_j})$$

is derivable, and

2. *for all* $\mathtt{ep}_i \in \mathsf{reachable}_P(\mathtt{ep}_s) \cup \{\mathtt{ep}_s\}$ *if there exists no* $\mathtt{ep}_j \in \mathsf{reachable}_P(\mathtt{ep}_s)$ *such that* $\mathtt{ep}_i \rightsquigarrow_P \mathtt{ep}_j$ *then the judgment*

$$P, \cdots, \mathtt{ep}_i : (\mathtt{sda}_{\mathtt{ep}_i}, \mathtt{md}_{\mathtt{ep}_i}, \mathtt{rda}_{\mathtt{ep}_i}, \mathtt{srda}_{\mathtt{ep}_i}) \vdash (\mathtt{sda}_{\mathtt{ep}_i}, \mathtt{md}_{\mathtt{ep}_i}, \mathtt{rda}_{\mathtt{ep}_i}, \mathtt{srda}_{\mathtt{ep}_i})$$

is derivable and $\mathtt{sda}_{\mathtt{ep}_i} \sqsubseteq_s \mathtt{sda}_{\mathtt{ep}_{final}}$, $\mathtt{md}_{\mathtt{ep}_i} \sqsubseteq \mathtt{md}_{\mathtt{ep}_{final}}$, $\mathtt{rda}_{\mathtt{ep}_i} \sqsubseteq \mathtt{rda}_{\mathtt{ep}_{final}}$, $\mathtt{srda}_{\mathtt{ep}_i} \sqsubseteq \mathtt{srda}_{\mathtt{ep}_{final}}$.

A program is typable if intermediate domain assignments can be assigned to all intermediate states in the program execution. The intermediate domain assignments must allow the derivation of a typing judgment that relates the intermediate domain assignments of an execution point to domain assignments that are at most as restrictive as the intermediate domain assignments of all successors of the execution point.

B.4 Typing Rules

$$\frac{\exists instr \in \{\texttt{call, jmp, rcall, rjmp}\} : P(ep) = (instr, (epa))}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda})} \ \textsf{(t-cf)}$$

Figure 1: Typing rules for control flow instructions.

$$\frac{P(ep) = (\mathtt{ret}, ())}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda})} \ \textsf{(t-ret)}$$

$$\frac{P(ep) = (\mathtt{clc}, ()) \quad \mathtt{srda}' = \mathtt{srda}[C \mapsto se(ep)]}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}', \mathtt{srda})} \ \textsf{(t-clc)}$$

$$\frac{P(ep) = (\mathtt{cli}, ())}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda})} \ \textsf{(t-cli)}$$

Figure 2: Typing rules for instructions without arguments.

$$\frac{\exists instr \in \{\texttt{breq, brne}\} : P(ep) = (instr, (epa)) \quad se(ep) \sqcup \texttt{srda}(Z) = \mathcal{L}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-brZ-l)}$$

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{breq, brne}\} : P(ep) = (instr, (epa)) \\ \neg \mathsf{loop}_P(ep) \quad se(ep) \sqcup \texttt{srda}(Z) = \mathcal{H} \quad se(ep) = \mathcal{H} \\ \forall ep' \in \mathsf{region}_P(ep) : se(ep') = \mathcal{H} \quad \texttt{sda}' = \mathsf{lift}(\texttt{sda}, \mathcal{H}) \\ \mathsf{branchtime}_P^{then}(ep) + \texttt{br} = \mathsf{branchtime}_P^{else}(ep)\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}', \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-brZ-h)}$$

$$\frac{\exists instr \in \{\texttt{brcc, brcs}\} : P(ep) = (instr, (epa)) \quad se(ep) \sqcup \texttt{srda}(C) = \mathcal{L}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-brC-l)}$$

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{brcc, brcs}\} : P(ep) = (instr, (epa)) \\ \neg \mathsf{loop}_P(ep) \quad se(ep) \sqcup \texttt{srda}(C) = \mathcal{H} \quad se(ep) = \mathcal{H} \\ \forall ep' \in \mathsf{region}_P(ep) : se(ep') = \mathcal{H} \quad \texttt{sda}' = \mathsf{lift}(\texttt{sda}, \mathcal{H}) \\ \mathsf{branchtime}_P^{then}(ep) + \texttt{br} = \mathsf{branchtime}_P^{else}(ep)\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}', \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-brC-h)}$$

$$\frac{P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad se(ep) \sqcup \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) = \mathcal{L}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-cpse-l)}$$

$$\frac{\begin{array}{c}P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad se(ep) \sqcup \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) = \mathcal{H} \\ \neg \mathsf{tw}_P(ep +_{\mathsf{ep}} 1) \quad se(ep) = \mathcal{H} \quad \neg \mathsf{loop}_P(ep) \quad \forall ep' \in \mathsf{region}_P(ep) : se(ep') = \mathcal{H} \\ \texttt{sda}' = \mathsf{lift}(\texttt{sda}, \mathcal{H}) \quad \mathsf{branchtime}_P^{then}(ep) + \texttt{br} = \mathsf{branchtime}_P^{else}(ep)\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}', \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-cpse-h)}$$

$$\frac{\begin{array}{c}P(ep) = (\texttt{cpse}, (Rd, Rr)) \quad se(ep) \sqcup \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) = \mathcal{H} \\ \mathsf{tw}_P(ep +_{\mathsf{ep}} 1) \quad se(ep) = \mathcal{H} \quad \neg \mathsf{loop}_P(ep) \quad \forall ep' \in \mathsf{region}_P(ep) : se(ep') = \mathcal{H} \\ \texttt{sda}' = \mathsf{lift}(\texttt{sda}, \mathcal{H}) \quad \mathsf{branchtime}_P^{then}(ep) + 2\texttt{br} = \mathsf{branchtime}_P^{else}(ep)\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}', \texttt{md}, \texttt{rda}, \texttt{srda})} \text{ (t-cpse-h2w)}$$

Figure 3: Typing rules for branching instructions.

$$\frac{\exists instr \in \{\texttt{dec, inc}\} : P(ep) = (instr, (Rd, k)) \quad \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{rda}(Rd) \sqcup se(ep)] \quad \texttt{srda}' = \texttt{srda}[Z \mapsto \texttt{rda}(Rd) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-1a-z)}$$

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{lsr, neg, ror}\} : P(ep) = (instr, (Rd, k)) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{rda}(Rd) \sqcup se(ep)] \\ \texttt{srda}' = \texttt{srda}[C \mapsto \texttt{rda}(Rd) \sqcup se(ep)][Z \mapsto \texttt{rda}(Rd) \sqcup se(ep)]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-1a-cz)}$$

Figure 4: Typing rules for instructions with one register as argument.

$$\frac{\begin{array}{c} P(ep) = (\texttt{push}, (Rr)) \qquad erg = \texttt{rda}(Rr) \sqcup se(ep) \\ spa = se(ep) \sqcup \texttt{rda}(\texttt{sp}_l) \sqcup \texttt{rda}(\texttt{sp}_u) \quad \texttt{rda}' = \texttt{rda}[\texttt{sp}_l \mapsto spa][\texttt{sp}_u \mapsto spa] \end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (erg :: \texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-push)}$$

$$\frac{\begin{array}{c} P(ep) = (\texttt{pop}, (Rd)) \qquad \texttt{rda}'' = \texttt{rda}[Rd \mapsto erg \sqcup se(ep)] \\ spa = se(ep) \sqcup \texttt{rda}(\texttt{sp}_l) \sqcup \texttt{rda}(\texttt{sp}_u) \quad \texttt{rda}' = \texttt{rda}''[\texttt{sp}_l \mapsto spa][\texttt{sp}_u \mapsto spa] \end{array}}{P, \cdots, ep : (erg :: \texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-pop)}$$

Figure 5: Typing rules for `push` and `pop` instructions.

$$\frac{\begin{array}{c} \exists instr \in \{\texttt{ld, ldd}\} : \exists a \in (\{\#\} \cup \mathbb{Z}) : P(ep) = (instr, (Rd, Rr, a)) \\ sd = \texttt{md} \sqcup se(ep) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}l}(Rr)) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}u}(Rr)) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto sd] \end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-ld-nm)}$$

$$\frac{\begin{array}{c} \exists a \in \{-, +\} : P(ep) = (\texttt{ld}, (Rd, Rr, a)) \qquad sd_2 = sd_1 \sqcup \texttt{md} \\ sd_1 = \texttt{rda}(\mathsf{regconvert\text{-}l}(Rr)) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}u}(Rr)) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto sd_2][\mathsf{regconvert\text{-}l}(Rr) \mapsto sd_1][\mathsf{regconvert\text{-}u}(Rr) \mapsto sd_1] \end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-ld-m)}$$

$$\frac{\begin{array}{c} \exists instr \in \{\texttt{st, std}\} : \exists a \in (\{\#\} \cup \mathbb{Z}) : P(ep) = (instr, (Rd, Rr, a)) \\ \texttt{md}' = \texttt{md} \sqcup \texttt{rda}(Rr) \sqcup se(ep) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}l}(Rd)) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}u}(Rd)) \end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}', \texttt{rda}, \texttt{srda})} \text{ (t-st-nm)}$$

$$\frac{\begin{array}{c} \exists a \in \{-, +\} : P(ep) = (\texttt{st}, (Rd, Rr, a)) \\ sd_1 = \texttt{rda}(\mathsf{regconvert\text{-}l}(Rd)) \sqcup \texttt{rda}(\mathsf{regconvert\text{-}u}(Rd)) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[\mathsf{regconvert\text{-}l}(Rd) \mapsto sd_1][\mathsf{regconvert\text{-}u}(Rd) \mapsto sd_1] \\ \texttt{md}' = \texttt{md} \sqcup \texttt{rda}(Rr) \sqcup sd_1 \end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}', \texttt{rda}', \texttt{srda})} \text{ (t-st)}$$

Figure 6: Typing rules for instructions with displacement.

$$\frac{P(ep) = (\texttt{in}, (Rd, \text{0x3d}))\quad \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{rda}(\texttt{sp}_l) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})}\ \textsf{(t-in-sp-l)}$$

$$\frac{P(ep) = (\texttt{in}, (Rd, \text{0x3e}))\quad \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{rda}(\texttt{sp}_u) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})}\ \textsf{(t-in-sp-u)}$$

$$\frac{P(ep) = (\texttt{in}, (Rd, \text{0x3f}))\quad \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{srda}(C) \sqcup \texttt{srda}(Z) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})}\ \textsf{(t-in-stat)}$$

$$\frac{P(ep) = (\texttt{out}, (\text{0x3d}, Rr))\quad \texttt{rda}' = \texttt{rda}[\texttt{sp}_l \mapsto \texttt{rda}(Rr) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})}\ \textsf{(t-out-sp-l)}$$

$$\frac{P(ep) = (\texttt{out}, (\text{0x3e}, Rr))\quad \texttt{rda}' = \texttt{rda}[\texttt{sp}_u \mapsto \texttt{rda}(Rr) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})}\ \textsf{(t-out-sp-u)}$$

$$\frac{\begin{array}{c}P(ep) = (\texttt{out}, (\text{0x3f}, Rr))\qquad erg = \texttt{rda}(Rr) \sqcup se(ep)\\ \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}')}\ \textsf{(t-out-stat)}$$

Figure 7: Typing rules for I/O instructions.

$$\frac{P(ep) = (\texttt{subi}, (Rd, k)) \qquad erg = \texttt{rda}(Rd) \sqcup se(ep) \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg][C \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-subi)}$$

$$\frac{P(ep) = (\texttt{andi}, (Rd, k)) \qquad erg = \texttt{rda}(Rd) \sqcup se(ep) \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-andi)}$$

$$\frac{P(ep) = (\texttt{cpi}, (Rd, k)) \qquad erg = \texttt{rda}(Rd) \sqcup se(ep) \qquad \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}')} \text{ (t-cpi)}$$

$$\frac{P(ep) = (\texttt{sbci}, (Rd, k)) \qquad erg = \texttt{rda}(Rd) \sqcup \texttt{srda}(C) \sqcup se(ep) \qquad \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg \sqcup \texttt{srda}(Z)] \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-sbci)}$$

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{adiw, sbiw}\} : P(ep) = (instr, (Rd, k)) \\ erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(\text{regconvert-u}(Rd)) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto erg][\text{regconvert-u}(Rd) \mapsto erg] \\ \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-ai-word)}$$

$$\frac{P(ep) = (\texttt{ldi}, (Rd, k)) \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-ldi)}$$

Figure 8: Typing rules for instructions with register and immediate arguments

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{add, sub}\} : P(ep) = (instr, (Rd, Rr)) \\ erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg][C \mapsto erg]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-2a-full)}$$

$$\frac{\begin{array}{c}\exists instr \in \{\texttt{and, or}\} : P(ep) = (instr, (Rd, Rr)) \\ erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-2a-full-oz)}$$

$$\frac{\begin{array}{c}P(ep) = (\texttt{adc}, (Rd, Rr)) \\ erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \sqcup \texttt{srda}(C) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-adc)}$$

$$\frac{\begin{array}{c}P(ep) = (\texttt{sbc}, (Rd, Rr)) \\ erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \sqcup \texttt{srda}(C) \\ \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg \sqcup \texttt{srda}(Z)]\end{array}}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-sbc)}$$

Figure 9: Typing rules for two argument instructions.

$$\frac{P(ep) = (\texttt{mul}, (Rd, Rr)) \qquad erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[r_0 \mapsto erg][r_1 \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-mul)}$$

$$\frac{P(ep) = (\texttt{eor}, (Rd, Rr)) \qquad erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \\ Rd \neq Rr \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg]}{P, \cdots ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-eor-normal)}$$

$$\frac{P(ep) = (\texttt{eor}, (Rd, Rr)) \qquad erg = se(ep) \\ Rd = Rr \qquad \texttt{rda}' = \texttt{rda}[Rd \mapsto erg] \qquad \texttt{srda}' = \texttt{srda}[Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda}')} \text{ (t-eor-erasure)}$$

$$\frac{P(ep) = (\texttt{mov}, (Rd, Rr)) \quad \texttt{rda}' = \texttt{rda}[Rd \mapsto \texttt{rda}(Rr) \sqcup se(ep)]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-mov)}$$

$$\frac{P(ep) = (\texttt{movw}, (Rd, Rr)) \\ erg = \texttt{rda}(\textsf{regconvert-l}(Rr)) \sqcup \texttt{rda}(\textsf{regconvert-u}(Rr)) \sqcup se(ep) \\ \texttt{rda}' = \texttt{rda}[\textsf{regconvert-l}(Rd) \mapsto erg][\textsf{regconvert-u}(Rd) \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}', \texttt{srda})} \text{ (t-movw)}$$

$$\frac{P(ep) = (\texttt{cp}, (Rd, Rr)) \qquad erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup se(ep) \\ \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}')} \text{ (t-cp)}$$

$$\frac{P(ep) = (\texttt{cpc}, (Rd, Rr)) \quad erg = \texttt{rda}(Rd) \sqcup \texttt{rda}(Rr) \sqcup \texttt{srda}(C) \sqcup se(ep) \\ \texttt{srda}' = \texttt{srda}[C \mapsto erg][Z \mapsto erg]}{P, \cdots, ep : (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}) \vdash (\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}')} \text{ (t-cpc)}$$

Figure 10: Typing rules for two argument instructions differing from groups.

C Proof of Type System Soundness

Our soundness proof utilizes the Unwinding technique and an additional lemma that guarantees the absence of timing variations in high branchings. The Unwinding lemmas have the following intuitions:

The main result of this section will be our soundness theorem.

Theorem 1 (Soundness). *Let $P \in \texttt{PROG}$ be a program and $\texttt{ep}_s \in \texttt{EPS}$ be an execution point. If P is typable starting from $\texttt{ep}_s$ with initial domain assignments $\texttt{sda}_{\texttt{ep}_s}, \texttt{md}_{\texttt{ep}_s}, \texttt{rda}_{\texttt{ep}_s}, \texttt{srda}_{\texttt{ep}_s}$ and final domain assignments $\texttt{sda}_{\texttt{ep}_f}, \texttt{md}_{\texttt{ep}_f}, \texttt{rda}_{\texttt{ep}_f}, \texttt{srda}_{\texttt{ep}_f}$, then P satisfies TSNI starting from $\texttt{ep}_s$ with the same initial and final domain assignments.*

Step Consistent 4: Executing an instruction in a low security environment from indistinguishable states leads to indistinguishable states and has a constant timing. Intuitively, this lemma ensures that no high data can be copied directly to a low data container.

Locally Respect 5: Executing an instruction in a high security environment leads to no observable change in states. Note, that states do not include running time. Intuitively, this lemma ensures no indirect leakage of information by control flow instructions.

High Branching 6: Control dependence regions are complete in the sense that all execution points that depend on a high branching decision are in a high security environment.

Indistinguishability after High Branching 7: Executing a sequence of instructions in a high security environment leads to no observable information leakage.

Timing Indistinguishability after High Branching 9: Executing a whole sequence of instructions in a high security environment up to the junction point leads to no observable information leakage. This includes no leakage of information by state indistinguishability, given by locally respect, as well as a passed execution time at the junction point that is constant.

Security of Typable Sequences 10: Starting an execution in indistinguishable states leads to indistinguishable states and does not leak any timing information.

The dependencies of the Unwinding lemmas are shown in Figure 11. During the proof, we further rely on a few properties of our indistinguishability relation $\approx$. We present the individual lemmas and their corresponding proofs in bottom up direction in the following, starting of with a collection of made assumptions and required properties of $\approx$. The section concludes with our proof of soundness, with Theorem 1.

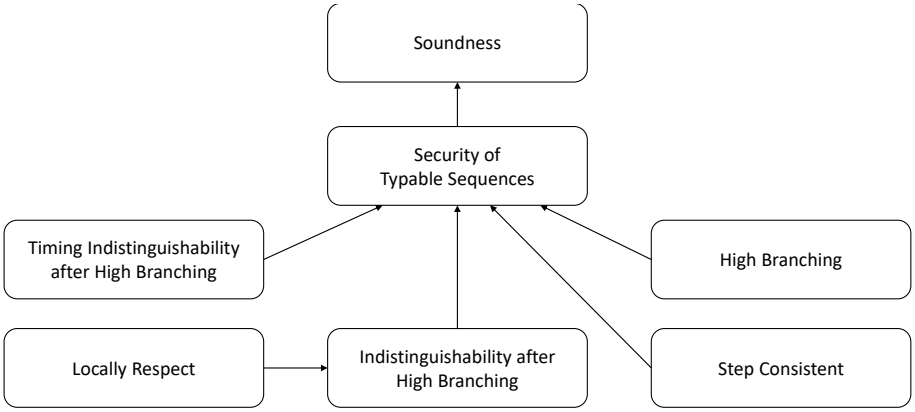

Figure 11: Dependencies of Unwinding lemmas.

C.1 Made Assumptions

Assumption 1 *The* `ret` *instruction inside a function is assumed to be unique.*

Inside our proofs, we rely heavily on the existence of junction points. A non-existent junction point can happen when there are separate `ret` statements inside each branch ("then" and "else") of a branching instruction. Thus, Assumption 1 guarantees that all branchings have a junction point. Note, that Assumption 1 has no negative impact of the expressiveness of our operational semantics, as programs can be rewritten to have such an unique `ret` instruction, e.g. jumping to a `ret` instruction at the end of branch.

Assumption 2 *It is assumed that control flow and branching instructions do not have an invalid execution point as argument.*

Control flow instruction and branching instructions have an execution point as operand. It might happen that this execution point is not in the domain of the considered program function. Such a case shall be avoided by Assumption 2.

Both, Assumption 1 and Assumption 2 hold for well-formed programs.

C.2 Properties of Indistinguishability

This section proves some required properties about the indistinguishability on states.

Lemma 1 (Reflexivity of $\approx$). *For all* $s \in \texttt{STATE}$ *and all domain assignments* $\texttt{sda}, \texttt{md}, \texttt{rda}, \texttt{srda}$ *it holds that* $s \approx_{\texttt{sda},\texttt{md},\texttt{rda},\texttt{srda}} s$.

Proof. Let $(sr, m, r, st, ep) \in \texttt{STATE}$ be an arbitrary state and let $\texttt{sda}, \texttt{md}, \texttt{rda}$, $\texttt{srda}$ be arbitrary domain assignments. It is now to show that

1. $sr \approx_{\texttt{srda}} sr$
2. $\texttt{md} = \mathcal{L} \Rightarrow m = m$
3. $r \approx_{\texttt{rda}} r$
4. $st \simeq_{\texttt{sda}} st$

Goal 2 is directly clear. For Goals 1 and 3 observe that indistinguishability of functions requires agreement on all $\mathcal{L}$ variables. As the functions are equal, they do agree on *all* values, so especially on all $\mathcal{L}$ values.

To show $st \simeq_{\texttt{sda}} st$, one can perform an induction over $|st|$:

Base Case $|st| = 0$*:* Applying Condition 2 in Definition 37 concludes the base case.

Induction Step: Let $st = x :: xs$. If $\texttt{sda} = []$, apply Condition 1 to conclude the induction step, else let $\texttt{sda} = y :: ys$. The induction hypothesis gives $xs \simeq_{ys} xs$, thus depending on y, apply Condition 3 or Condition 4 in Definition 37 to conclude the induction step. □

Lemma 2 (Monotonicity of $\approx$). *For all* $s_1, s_2, s_3 \in \texttt{STATE}$ *and for all domain assignments* $\texttt{sda}_1, \texttt{md}_1, \texttt{rda}_1, \texttt{srda}_1$, $\texttt{sda}_2, \texttt{md}_2, \texttt{rda}_2, \texttt{srda}_2$, *if* $s_1 \approx_{\texttt{sda}_1,\texttt{md}_1,\texttt{rda}_1,\texttt{srda}_1} s_2$, $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_3$, $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ *and* $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$, *then* $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_3$.

Proof. Let $(sr_1, m_1, r_1, st_1, ep_1), (sr_2, m_2, r_2, st_2, ep_2), (sr_3, m_3, r_3, st_3, ep_3) \in \texttt{STATE}$ be arbitrary states and let $\texttt{sda}_1, \texttt{md}_1, \texttt{rda}_1, \texttt{srda}_1$, $\texttt{sda}_2, \texttt{md}_2, \texttt{rda}_2, \texttt{srda}_2$ be arbitrary domain assignments, such that

1. $(sr_1, m_1, r_1, st_1, ep_1) \approx_{\texttt{sda}_1,\texttt{md}_1,\texttt{rda}_1,\texttt{srda}_1} (sr_2, m_2, r_2, st_2, ep_2)$
2. $(sr_2, m_2, r_2, st_2, ep_2) \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} (sr_3, m_3, r_3, st_3, ep_3)$
3. $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$

It is now to show that $(sr_1, m_1, r_1, st_1, ep_1) \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} (sr_3, m_3, r_3, st_3, ep_3)$, or expanded that

1. $sr_1 \approx_{\texttt{srda}_2} sr_3$
2. $\texttt{md}_2 = \mathcal{L} \Rightarrow m_1 = m_3$
3. $r_1 \approx_{\texttt{rda}_2} r_3$
4. $st_1 \simeq_{\texttt{sda}_2} st_3$

For Goal 1, let $r_s \in \{C, Z\}$ be an arbitrary identifier for a status register. It is now to show that $\texttt{srda}_2(r_s) = \mathcal{L} \Rightarrow sr_1(r_s) = sr_3(r_s)$. So assume $\texttt{srda}_2(r_s) = \mathcal{L}$. By Assumption 2 this directly gives $sr_2(r_s) = sr_3(r_s)$. By Assumption 3, one gets that $\texttt{srda}_1(r_s) \sqsubseteq \texttt{srda}_2(r_s)$, that is $\texttt{srda}_2(r_s) = \mathcal{L} \Rightarrow \texttt{srda}_1(r_s) = \mathcal{L}$. This gives $sr_1(r_s) = sr_2(r_s)$ and in combination $sr_1(r_s) = sr_3(r_s)$ what was to show. Goal 3 is shown analogously.

For Goal 2, assume $\texttt{md}_2 = \mathcal{L}$. By Assumption 2 this gives $m_2 = m_3$. By Assumption 3 one obtains that $\texttt{md}_1 = \mathcal{L}$ as well, so $m_1 = m_2$ by Assumption 1. Together, it is $m_1 = m_3$.

For Goal 4, let $h_1 \in \{\mathcal{H}\}^*$ and $h_2 \in \{\mathcal{H}\}^*$ be the largest lists such that $\mathtt{sda}_1 = \mathtt{sda}_1' :: h_1$, $\mathtt{sda}_2 = \mathtt{sda}_2' :: h_2$ and $|\mathtt{sda}_1'| = |\mathtt{sda}_2'|$. As we have by Assumption 3 that $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$ one obtains by the definition of $\sqsubseteq_s$ that $\mathtt{sda}_1' \sqsubseteq_s \mathtt{sda}_2'$ holds. In the lower part represented by h_1 and h_2, indistinguishability is directly given, according to the definition of $\simeq$. So it is left to show indistinguishability for the upper part. For this, we perform an induction over the height of the upper part, represented by $\mathtt{sda}_1'$ and $\mathtt{sda}_2'$.

Base Case: Assume $|\mathtt{sda}_2'| = 0$, then we get indistinguishability directly by condition 1 in Definition 37.

Induction Hypothesis: Indistinguishability is assumed to hold for all $\mathtt{sda}_2''$ with $|\mathtt{sda}_2''| < |\mathtt{sda}_2'|$.

Induction Step: Assume $\mathtt{sda}_2' = \mathcal{L} :: \mathtt{sda}_2''$. Let $st_1 = x_1 :: st_1'$, $st_2 = x_2 :: st_2'$ and $st_3 = x_3 :: st_3'$. According to Definition 37 it is now to show that $x_1 = x_3$. Let $\mathtt{sda}_1' = d :: \mathtt{sda}_1''$. As we have that $\mathtt{sda}_1' \sqsubseteq_s \mathtt{sda}_2'$ and $\mathtt{sda}_2' = \mathcal{L} :: \mathtt{sda}_2''$, one obtains that $d \sqsubseteq \mathcal{L}$ and as such, $d = \mathcal{L}$. By Assumption 1 and Assumption 2 one then obtains $x_1 = x_2$ and $x_2 = x_3$. This implies $x_1 = x_3$, what was to show. Applying the induction hypothesis on $\mathtt{sda}_2''$, which is now shorter than $\mathtt{sda}_2'$ concludes the induction step. □

Lemma 3 (Transitivity of $\approx$). *For all $s_1, s_2, s_3 \in \mathtt{STATE}$ and for all domain assignments $\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}$, whenever $s_1 \approx_{\mathtt{sda},\mathtt{md},\mathtt{rda},\mathtt{srda}} s_2$ and $s_2 \approx_{\mathtt{sda},\mathtt{md},\mathtt{rda},\mathtt{srda}} s_3$, then $s_1 \approx_{\mathtt{sda},\mathtt{md},\mathtt{rda},\mathtt{srda}} s_3$.*

Proof. Observe that $\mathtt{sda} \sqsubseteq_s \mathtt{sda}$, $\mathtt{md} \sqsubseteq \mathtt{md}$, $\mathtt{rda} \sqsubseteq \mathtt{rda}$ and $\mathtt{srda} \sqsubseteq \mathtt{srda}$. So transitivity is a special case of monotonicity and follows from Lemma 2. □

C.3 Step Consistent

Lemma 4 (Step Consistent). *Let $P \in \mathtt{PROG}$ be a program. For all states $s_1, s_1', s_2, s_2' \in \mathtt{STATE}$, clock cycles $c, c' \in \mathbb{N}$, domain assignments $\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1, \mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2$ and security environments se, if*

1. $\mathsf{epselect}(s_1) = \mathsf{epselect}(s_1')$
2. $se(\mathsf{epselect}(s_1)) = \mathcal{L}$
3. $s_1 \approx_{\mathtt{sda}_1,\mathtt{md}_1,\mathtt{rda}_1,\mathtt{srda}_1} s_1'$
4. $s_1 \xrightarrow{c}_P s_2$
5. $s_1' \xrightarrow{c'}_P s_2'$
6. $P, \cdots, \mathsf{epselect}(s_1) : (\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1) \vdash (\mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2)$

then $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$, $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$ and $c = c'$.

Proof (Lemma 4 – Step Consistent). Let $P \in \mathtt{PROG}$ be a program. Let s_1, s_1', $s_2, s_2' \in \mathtt{STATE}$ be states, $c, c' \in \mathbb{N}$ be clock cycles, $\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1, \mathtt{sda}_2$, $\mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2$ be domain assignments and *se* be a security environment, such that

1. $\mathsf{epselect}(s_1) = \mathsf{epselect}(s_1')$
2. $se(\mathsf{epselect}(s_1)) = \mathcal{L}$
3. $s_1 \approx_{\mathtt{sda}_1,\mathtt{md}_1,\mathtt{rda}_1,\mathtt{srda}_1} s_1'$
4. $s_1 \xrightarrow{c}_P s_2$
5. $s_1' \xrightarrow{c'}_P s_2'$
6. $P, \cdots, \mathsf{epselect}(s_1) : (\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1) \vdash (\mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2)$

It is now to show that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$, $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$ and $c = c'$.

This is proven by a case distinction over the possibly executed instruction $P(\mathsf{epselect}(s_1))$. For this, let $s_1 = (sr_1, m_1, r_1, st_1, ep_1)$, $s_1' = (sr_1', m_1', r_1', st_1', ep_1')$, $s_2 = (sr_2, m_2, r_2, st_2, ep_2)$ and $s_2' = (sr_2', m_2', r_2', st_2', ep_2')$.

As all instructions, except control flow and branching instructions, set their next execution point to $ep_1 +_{\mathtt{ep}} 1$ and by Assumption 1 $ep_1 = ep_1'$, one gets directly that $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$. Additionally, instructions (except branching instructions) are executed in constant time, giving $c = c'$ for all other instructions. We thus limit ourselves to show $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ in all cases except branching instructions and control flow instructions.

When listing changes to domain assignments, we leave out the security environment for brevity, as it is $\mathcal{L}$ by Assumption 2. Taking the least upper bound with a $\mathcal{L}$ security domain does not change it.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{push}, (Rr))$ The `push` instruction adds the contents of register Rr to the stack and decrements the stack pointer by one. Thus, the stack as well as the current register contents are modified.

Status registers and memory are left unmodified, giving $sr_1 = sr_2$ and $m_1 = m_2$ and the same for the primed ones. So it remains to show that registers and stack are still indistinguishable.

Assume $\mathtt{sda}_2 = \mathcal{L} :: \mathtt{sda}_1$, then it is to show that $r_1(Rr) = r_1'(Rr)$ to obtain stack indistinguishability according to its definition and the small-step semantic rule (push). From $\mathtt{sda}_2 = \mathcal{L} :: \mathtt{sda}_1$ one obtains that $\mathcal{L} = \mathtt{rda}_1(Rr) \sqcup se(\mathsf{epselect}(s_1))$ and from this it follows that $\mathtt{rda}_1(Rr) = \mathcal{L}$. As states s_1 and s_1' are indistinguishable by Assumption 3 and $\mathtt{rda}_1(Rr) = \mathcal{L}$, one obtains that $r_1(Rr) = r_1'(Rr)$, what was to show.

It remains to show indistinguishability for the stack pointer registers. For this, assume $\mathtt{rda}_2(\mathtt{sp}_l) = \mathcal{L}$ or $\mathtt{rda}_2(\mathtt{sp}_u) = \mathcal{L}$. According to rule (t-push), both register domain assignments are updated according to $\mathtt{rda}_1(\mathtt{sp}_l) \sqcup \mathtt{rda}_1(\mathtt{sp}_u)$. So this can only be the case if $\mathtt{rda}_1(\mathtt{sp}_l) = \mathcal{L}$ and $\mathtt{rda}_1(\mathtt{sp}_u) = \mathcal{L}$.

As states s_1 and s_1' are indistinguishable by Assumption 3 and $\mathtt{rda}_1(\mathtt{sp}_l) = \mathtt{rda}_1(\mathtt{sp}_u) = \mathcal{L}$, one obtains that $r_1(\mathtt{sp}_l) = r_1'(\mathtt{sp}_l)$ and $r_1(\mathtt{sp}_u) = r_1'(\mathtt{sp}_u)$. This directly gives $r_2(\mathtt{sp}_l) = r_2'(\mathtt{sp}_l)$ and $r_2(\mathtt{sp}_u) = r_2'(\mathtt{sp}_u)$, what was to show.

We have shown both required indistinguishabilities and can thus conclude $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ for this case.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{pop}, (Rd))$ The `pop` instruction removes the topmost element from the stack and stores it in register Rd. Additionally the stack pointer

is incremented by one. Thus, the stack as well as the current register contents are modified.

Status registers and memory are left unmodified, giving $sr_1 = sr_2$ and $m_1 = m_2$ and the same for the primed ones. So it remains to show that registers and stack are still indistinguishable.

According to rule (t-pop) one obtains for the stack domain assignment that $d :: \mathtt{sda}_2 = \mathtt{sda}_1$ for $d \in \{\mathcal{L}, \mathcal{H}\}$. We assume $d = \mathcal{L}$, as otherwise indistinguishabilty would follow from the minimality of $\simeq$. Let $st_1 = v :: vs$ and $st_1' = v' :: vs'$ for some $v, v' \in \mathtt{VAL}_8$ and $vs, vs' \in \mathtt{STACK\text{-}VAL}$. As we have $d = \mathcal{L}$ and by Assumption 3 that s_1 and s_1' are indistinguishable, we get by the definition of stack indistinguishability that $v = v'$ and $vs \simeq_{\mathtt{sda}_2} vs'$. As $r_2(Rd) = v$, $r_2'(Rd) = v'$ and $v = v'$, we get indistinguishability for register contents of Rd.

It remains to show indistinguishability for the stack pointer registers. For this, assume $\mathtt{rda}_2(\mathtt{sp}_l) = \mathcal{L}$ or $\mathtt{rda}_2(\mathtt{sp}_u) = \mathcal{L}$. According to rule (t-pop), both register domain assignments are updated according to $\mathtt{rda}_1(\mathtt{sp}_l) \sqcup \mathtt{rda}_1(\mathtt{sp}_u)$. So this can only be the case if $\mathtt{rda}_1(\mathtt{sp}_l) = \mathcal{L}$ and $\mathtt{rda}_1(\mathtt{sp}_u) = \mathcal{L}$.

As states s_1 and s_1' are indistinguishable by Assumption 3 and $\mathtt{rda}_1(\mathtt{sp}_l) = \mathtt{rda}_1(\mathtt{sp}_u) = \mathcal{L}$, one obtains that $r_1(\mathtt{sp}_l) = r_1'(\mathtt{sp}_l)$ and $r_1(\mathtt{sp}_u) = r_1'(\mathtt{sp}_u)$. This directly gives $r_2(\mathtt{sp}_l) = r_2'(\mathtt{sp}_l)$ and $r_2(\mathtt{sp}_u) = r_2'(\mathtt{sp}_u)$, what was to show.

We have shown both required indistinguishabilities and can thus conclude $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ for this case.

Case $P(\mathsf{epselect}(s_1)) = (instr, (epa))$ *for* $instr \in \{\mathtt{breq}, \mathtt{brne}\}$ All those instructions have in common that no data storage is modified. That is, their small-step rules give $sr_1 = sr_2, m_1 = m_2, r_1 = r_2$ and $st_1 = st_2$ and similar for the primed states. Since all applicable typing rules do not modify any domain assignments, it follows that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ by Assumption 3.

As we have by Assumption 2 that $se(\mathsf{epselect}(s_1)) = se(\mathsf{epselect}(s_1')) = \mathcal{L}$, only rule (t-brZ-l) can be applicable. Rule (t-brZ-h) can not be used as it requires a high security environment. The applicable rules require further that $\mathtt{srda}_1(Z) = \mathcal{L}$ which implies $sr_1(Z) = sr_1'(Z)$ by Assumption 3. As such the assumed transitions 4 and 5 have to be made with the same small-step semantic rule. As such, this gives $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$ and $c = c'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (epa))$ *for* $instr \in \{\mathtt{brcc}, \mathtt{brcs}\}$ All those instructions have in common that no data storage is modified. That is, their small-step rules give $sr_1 = sr_2, m_1 = m_2, r_1 = r_2$ and $st_1 = st_2$ and similar for the primed states. Since all applicable typing rules do not modify any domain assignments, it follows that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ by Assumption 3.

As we have by Assumption 2 that $se(\mathsf{epselect}(s_1)) = se(\mathsf{epselect}(s_1')) = \mathcal{L}$, only rule (t-brC-l) can be applicable. Rule (t-brC-h) can not be used as it requires a high security environment. The applicable rules require further that $\mathtt{srda}_1(C) = \mathcal{L}$ which implies $sr_1(C) = sr_1'(C)$. As such the assumed transitions 4 and 5 have to be made with the same small-step semantic rule. As such, this gives $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$ and $c = c'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{cpse}, (Rd, Rr))$ This instruction does not modify any data storage. That is, its small-step rules give $sr_1 = sr_2, m_1 = m_2, r_1 = r_2$ and $st_1 = st_2$ and similar for the primed states. Since all applicable typing rules do not modify any domain assignments, it follows that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ by Assumption 3.

As we have by Assumption 2 that $se(\mathsf{epselect}(s_1)) = se(\mathsf{epselect}(s_1')) = \mathcal{L}$, only rule (t-cpse-l) can be applicable. Rule (t-cpse-h) or (t-cpse-h2w) can not be used as it requires a high security environment. The applicable rules require further that $\mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr) = \mathcal{L}$ which implies $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(Rr) = \mathcal{L}$. This further gives $r_1(Rd) = r_1'(Rd)$ and $r_1(Rr) = r_1'(Rr)$. As such the assumed transitions 4 and 5 have to be made with the same small-step semantic rule. As such, this gives $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$ and $c = c'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr, a))$ *for* $instr \in \{\texttt{ld}, \texttt{ldd}\}$ Instructions `ld` and `ldd` load values from memory into register Rd and depending on a, they also increment or decrement the pointer register pair Rr. Memory, stack and status registers are left unmodified giving $sr_1 = sr_2$, $m_1 = m_2$, $st_1 = st_2$ and similar for the primed data containers. So it remains to show that registers are still indistinguishable. As parameter a has influence on which registers are modified, a case distinction over values of a is required:

$a = \#$ *or* $instr = \texttt{ldd}$*:* In this case, register Rr is left unmodified. Only Rd is modified.

So we look at the case $\mathtt{rda}_2(Rd) = \mathcal{L}$. As we have by (t-ld-nm) that $\mathtt{rda}_2(Rd) = \mathtt{md}_1 \sqcup \mathtt{rda}_1(\mathsf{regconvert\text{-}l}(Rr)) \sqcup \mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rr))$, it follows that $\mathtt{md}_1 = \mathcal{L}$, $\mathtt{rda}_1(\mathsf{regconvert\text{-}l}(Rr)) = \mathcal{L}$ and $\mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rr)) = \mathcal{L}$. Together with Assumption 3, one obtains that $m_1 = m_1'$ and $\mathsf{regpair\text{-}val}_{r_1}(Rr) = \mathsf{regpair\text{-}val}_{r_1'}(Rr)$.

According to the semantics, Rd is set to $m_1(\mathsf{regpair\text{-}val}_{r_1}(Rr))$. As we have equalities for both, the register pair Rr and m, we can conclude that $r_2(Rd) = r_2'(Rd)$ and thus $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

$a \in \{+, -\}$*:* In this case, Rr is modified. The arguments in the previous case can be applied the same way to obtain $m_1 = m_1'$ and $\mathsf{regpair\text{-}val}_{r_1}(Rr) = \mathsf{regpair\text{-}val}_{r_1'}(Rr)$.

From this, one similarly gets $r_2(Rd) = r_2'(Rd)$. As Rr is incremented or decremented by one depending on a, it follows that $\mathsf{regpair\text{-}val}_{r_2}(Rr) = \mathsf{regpair\text{-}val}_{r_2'}(Rr)$. Together, it follows that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr, a))$ *for* $instr \in \{\texttt{st}, \texttt{std}\}$ Instructions `st` and `std` load values from register Rr into the memory pointed to by register Rd. Stack and status registers are left unmodified giving $st_1 = st_2$ and $sr_1 = sr_2$ and similar for the primed data containers. Depending on a, they also increment or decrement the pointer register Rd. So another case distinction over values of a is required:

$a = \#$ *or* $instr = \texttt{std}$*:* In this case, only the memory is modified. Registers are left unchanged. So we have a look at the case $\mathtt{md}_2 = \mathcal{L}$. As we have by rule (t-st-nm) that $\mathtt{md}_2 = \mathtt{md}_1 \sqcup \mathtt{rda}_1(Rr) \sqcup \mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rd)) \sqcup$

$\mathtt{rda}_1(\mathsf{regconvert\text{-}l}(Rd))$, we obtain that $\mathtt{md}_1 = \mathcal{L}$, $\mathtt{rda}_1(Rr) = \mathcal{L}$, $\mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L}$ and $\mathtt{rda}_1(\mathsf{regconvert\text{-}l}(Rd)) = \mathcal{L}$. Together with Assumption 3, one obtains that $m_1 = m_1'$, $r_1(Rr) = r_1'(Rr)$ and $\mathsf{regpair\text{-}val}_{r_1}(Rd) = \mathsf{regpair\text{-}val}_{r_1'}(Rd)$.

According to the semantics, the memory at address $\mathsf{regpair\text{-}val}_{r_1}(Rd)$ is updated to $r_1(Rr)$. As we have all those equalities, one can conclude that $m_2 = m_2'$ and $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

$a \in \{+,-\}$*:* In this case, Rd is modified. The arguments in the previous case can be applied the same way to obtain $m_1 = m_1'$, $r_1(Rr) = r_1'(Rr)$ and $\mathsf{regpair\text{-}val}_{r_1}(Rd) = \mathsf{regpair\text{-}val}_{r_1'}(Rd)$.

From this, one similarly gets $m_2 = m_2'$. As Rd is incremented or decremented by one depending on a, it follows that $\mathsf{regpair\text{-}val}_{r_2}(Rd) = \mathsf{regpair\text{-}val}_{r_2'}(Rd)$. Together, it follows that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{adc}, (Rd, Rr))$ Instruction `adc` modifies register Rd and status registers C and Z, according to contents of register Rd and Rr and the carry flag C. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$, $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-adc) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr) \sqcup \mathtt{srda}_1(C)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$, $\mathtt{rda}_1(Rr) = \mathcal{L}$ and $\mathtt{srda}_1(C) = \mathcal{L}$.

By definition of indistinguishability and Assumption 3 this implies $r_1(Rd) = r_1'(Rd)$, $r_1(Rr) = r_1'(Rr)$ and $sr_1(C) = sr_1'(C)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{sbc}, (Rd, Rr))$ Instruction `sbc` modifies register Rd and status registers C and Z, according to contents of register Rd and Rr and the carry flag C. Status register Z also depends on the previous value of Z. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$, $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-sbc) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr) \sqcup \mathtt{srda}_1(C)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$, $\mathtt{rda}_1(Rr) = \mathcal{L}$ and $\mathtt{srda}_1(C) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$, $r_1(Rr) = r_1'(Rr)$ and $sr_1(C) = sr_1'(C)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(C) = sr_2'(C)$.

It remains to show that $sr_2(Z) = sr_2'(Z)$. For this, assume specifically that $\mathtt{srda}_2(Z) = \mathcal{L}$. According to (t-sbc) it follows that $\mathtt{srda}_1(Z) = \mathcal{L}$, which implies $sr_1(Z) = sr_1'(Z)$. As Z depends on the previous value of Z and the computation result, we can conclude $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{sbci}, (Rd, k))$ Instruction `sbci` modifies register Rd and status registers C and Z, according to contents of register Rd, the given immediate k and the carry flag C. Status register Z also depends on the previous value of Z. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$, $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-sbci) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{srda}_1(C)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{srda}_1(C) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $sr_1(C) = sr_1'(C)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(C) = sr_2'(C)$.

It remains to show that $sr_2(Z) = sr_2'(Z)$. For this, assume specifically that $\mathtt{srda}_2(Z) = \mathcal{L}$. According to (t-sbci) it follows that $\mathtt{srda}_1(Z) = \mathcal{L}$, which implies $sr_1(Z) = sr_1'(Z)$. As Z depends on the previous value of Z and the computation result, we can conclude $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr))$ *for* $instr \in \{\texttt{add}, \texttt{sub}\}$ Both possible instructions have in common, that they modify register Rd and status registers C and Z, according to contents of register Rd and Rr. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$, $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-2a-full) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(Rr) = r_1'(Rr)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr))$ *for* $instr \in \{\texttt{and}, \texttt{or}\}$ Both possible instructions have in common, that they modify register Rd and status register Z, according to contents of register Rd and Rr. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-2a-full-oz) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(Rr) = r_1'(Rr)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\texttt{dec}, \texttt{inc}\}$ Both possible instructions have in common, that they modify register Rd and status register Z, according to contents of register Rd. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. By rule (t-1a-z) one gets from this that $\texttt{rda}_1(Rd) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\texttt{lsr}, \texttt{neg}, \texttt{ror}\}$ All possible instructions have in common, that they modify register Rd and status registers C and Z, according to contents of register Rd. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\texttt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$, $\texttt{srda}_2(C) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. By rule (t-1a-cz) one gets from this that $\texttt{rda}_1(Rd) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{eor}, (Rd, Rr))$ Instruction `eor` performs an exclusive-or operation on the contents of registers Rd and Rr and stores the result in register Rd. It also modifies status register Z. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. Then there are two possible cases:
$Rd \neq Rr$: By rule (t-eor-normal) one gets from this that $erg = \mathcal{L}$ and as $erg = \texttt{rda}_1(Rd) \sqcup \texttt{rda}_1(Rr)$, it follows that $\texttt{rda}_1(Rd) = \mathcal{L}$ and $\texttt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(Rr) = r_1'(Rr)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(Z) = sr_2'(Z)$.

$Rd = Rr$: In this case typing rule (t-eor-erasure) is applicable. The semantics of `eor` imply that $r_2(Rd) = 0$ and $r_2'(Rd) = 0$, independent of the values in registers Rd and Rr. This also directly implies that $sr_2(Z) = 1$ and $sr_2'(Z) = 1$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{ldi}, (Rd, k))$ Instruction `ldi` loads the given immediate value k into register Rd. No status registers are modified. Thus, one gets directly that $sr_1 = sr_2$, $sr_1' = sr_2'$, $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r'_2(Rd)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$. Then according to the semantics, one gains that $r_2(Rd) = k$ and $r'_2(Rd) = k$, resulting in equality, what was to show.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s'_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cp}, (Rd, Rr))$ Instruction `cp` performs a comparison between the contents of registers Rd and Rr. It stores the results of the comparison in status registers C and Z. The registers are left unmodified. Thus, one gets directly that $r_1 = r_2$, $r'_1 = r'_2$, $m_1 = m_2$, $m'_1 = m'_2$, $st_1 = st_2$ and $st'_1 = st'_2$.

So it remains to show that $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr'_2(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr'_2(Z)$.

Assume $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-cp) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r'_1(Rd)$ and $r_1(Rr) = r'_1(Rr)$. As we have equality of all arguments and a function is executed, one gets that $sr_2(C) = sr'_2(C)$ and $sr_2(Z) = sr'_2(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s'_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cpc}, (Rd, Rr))$ Instruction `cpc` performs a comparison between the contents of registers Rd and Rr and status register C. It stores the results of the comparison in status registers C and Z. The registers are left unmodified. Thus, one gets directly that $r_1 = r_2$, $r'_1 = r'_2$, $m_1 = m_2$, $m'_1 = m'_2$, $st_1 = st_2$ and $st'_1 = st'_2$.

So it remains to show that $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr'_2(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr'_2(Z)$.

Assume $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-cpc) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr) \sqcup \mathtt{srda}_1(C)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$, $\mathtt{rda}_1(Rr) = \mathcal{L}$ and $\mathtt{srda}_1(C) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r'_1(Rd)$, $r_1(Rr) = r'_1(Rr)$ and $sr_1(C) = sr'_1(C)$. As we have equality of all arguments and a function is executed, one gets that $sr_2(C) = sr'_2(C)$ and $sr_2(Z) = sr'_2(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s'_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cpi}, (Rd, k))$ Instruction `cpi` performs a comparison between the contents of registers Rd and the given immediate value k. It stores the results of the comparison in status registers C and Z. The registers are left unmodified. Thus, one gets directly that $r_1 = r_2$, $r'_1 = r'_2$, $m_1 = m_2$, $m'_1 = m'_2$, $st_1 = st_2$ and $st'_1 = st'_2$.

So it remains to show that $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr'_2(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr'_2(Z)$.

Assume $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-cpi) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r'_1(Rd)$. As we have equality of all arguments and a function is executed, one gets that $sr_2(C) = sr'_2(C)$ and $sr_2(Z) = sr'_2(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{clc}, ())$ Instruction `clc` clears the C status register. No other data containers are modified. Thus, one gets directly that $r_1 = r_2$, $r_1' = r_2'$, $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$.

Assume $\mathtt{srda}_2(C) = \mathcal{L}$. Then according to the semantics, one gains that $sr_2(C) = 0$ and $sr_2'(C) = 0$, resulting in equality, what was to show.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cli}, ())$ Instruction `cli` clears the I status register. As we do not consider the I flag and the small-step rule has no effects, one gets directly that $r_1 = r_2$, $r_1' = r_2'$, $sr_1 = sr_2$, $sr_1' = sr_2'$, $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{mov}, (Rd, Rr))$ Instruction `mov` copies the contents of register Rr into register Rd. Thus, one gets directly that $sr_1 = sr_2$, $sr_1' = sr_2'$, $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$. By rule (t-mov) one gets from this that $\mathtt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rr) = r_1'(Rr)$ and. By the semantics of `mov` one obtains that $r_2(Rd) = r_1(Rr)$ and $r_2'(Rd) = r_1'(Rr)$, which are both equal what was to show.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{movw}, (Rd, Rr))$ Instruction `movw` copies the contents of register pair Rr into register pair Rd. Thus, one gets directly that $sr_1 = sr_2$, $sr_1' = sr_2'$, $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ and $\mathtt{rda}_2(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L} \Rightarrow r_2(\mathsf{regconvert\text{-}u}(Rd)) = r_2'(\mathsf{regconvert\text{-}u}(Rd))$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$ or $\mathtt{rda}_2(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L}$. By rule (t-movw) one gets from this that $erg = \mathcal{L}$ and $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rd))$. Thus, it must be that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(\mathsf{regconvert\text{-}u}(Rd)) = r_1'(\mathsf{regconvert\text{-}u}(Rd))$. This implies $\mathsf{regpair\text{-}val}_{r_1}(Rd) = \mathsf{regpair\text{-}val}_{r_1'}(Rd)$. As we have equality of all arguments, one gets that $r_2(Rd) = r_2'(Rd)$ and $r_2(\mathsf{regconvert\text{-}u}(Rd)) = r_2'(\mathsf{regconvert\text{-}u}(Rd))$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{subi}, (Rd, k))$ Instruction `subi` subtracts k from the register Rd. It stores results in register Rd and modifies status registers C and Z. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$, $\texttt{srda}_2(C) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. By rule (t-subi) one gets from this that $erg = \mathcal{L}$ and as $erg = \texttt{rda}_1(Rd)$, it follows that $\texttt{rda}_1(Rd) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$. As we have equality of all arguments (Rd and k) and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{andi}, (Rd, k))$ Instruction `andi` performs a logical and operation on k and the contents of register Rd. It stores its result in register Rd and modifies status register Z accordingly. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. By rule (t-andi) one gets from this that $erg = \mathcal{L}$ and as $erg = \texttt{rda}_1(Rd)$, it follows that $\texttt{rda}_1(Rd) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$. As we have equality of all arguments (Rd and k) and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\texttt{adiw}, \texttt{sbiw}\}$ All possible instructions have in common, that they modify register pair Rd and status registers C and Z, according to the previous contents of register pair Rd and the immediate value k. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$, $\texttt{rda}_2(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L} \Rightarrow r_2(\mathsf{regconvert\text{-}u}(Rd)) = r_2'(\mathsf{regconvert\text{-}u}(Rd))$, $\texttt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\texttt{rda}_2(Rd) = \mathcal{L}$, $\texttt{rda}_2(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L}$, $\texttt{srda}_2(C) = \mathcal{L}$ or $\texttt{srda}_2(Z) = \mathcal{L}$. By rule (t-ai-word) one gets from this that $erg = \mathcal{L}$ and $erg = \texttt{rda}_1(Rd) \sqcup \texttt{rda}_1(\mathsf{regconvert\text{-}u}(Rd))$. Thus, it must be that $\texttt{rda}_1(Rd) = \mathcal{L}$ and $\texttt{rda}_1(\mathsf{regconvert\text{-}u}(Rd)) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(\mathsf{regconvert\text{-}u}(Rd)) = r_1'(\mathsf{regconvert\text{-}u}(Rd))$. This implies $\mathsf{regpair\text{-}val}_{r_1}(Rd) = \mathsf{regpair\text{-}val}_{r_1'}(Rd)$. As we have equality of all arguments and a function is executed, one gets that $r_2(Rd) = r_2'(Rd)$, $r_2(\mathsf{regconvert\text{-}u}(Rd)) = r_2'(\mathsf{regconvert\text{-}u}(Rd))$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{mul}, (Rd, Rr))$ Instruction `mul` modifies register pair r_0 and status registers C and Z, according to the product of registers Rd and Rr. Thus, one gets directly that $m_1 = m_2$, $m_1' = m_2'$, $st_1 = st_2$ and $st_1' = st_2'$.

So it remains to show that $\texttt{rda}_2(r_0) = \mathcal{L} \Rightarrow r_2(r_0) = r_2'(r_0)$, $\texttt{rda}_2(r_1) = \mathcal{L} \Rightarrow r_2(r_1) = r_2'(r_1)$, $\texttt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\texttt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$.

Assume $\mathtt{rda}_2(r_0) = \mathcal{L}$, $\mathtt{rda}_2(r_1) = \mathcal{L}$, $\mathtt{srda}_2(C) = \mathcal{L}$ or $\mathtt{srda}_2(Z) = \mathcal{L}$. By rule (t-mul) one gets from this that $erg = \mathcal{L}$ and as $erg = \mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr)$, it follows that $\mathtt{rda}_1(Rd) = \mathcal{L}$ and $\mathtt{rda}_1(Rr) = \mathcal{L}$.

By definition of indistinguishability this implies $r_1(Rd) = r_1'(Rd)$ and $r_1(Rr) = r_1'(Rr)$. As we have equality of all arguments and a function is executed, one gets that $r_2(r_0) = r_2'(r_0)$, $r_2(r_1) = r_2'(r_1)$, $sr_2(C) = sr_2'(C)$ and $sr_2(Z) = sr_2'(Z)$.

Together, this gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{in}, (Rd, addr))$ *for* $addr \in \{\mathtt{0x3d}, \mathtt{0x3e}\}$*:* According to the semantics of `in` with the given $addr$, the register Rd is set to the contents of $\mathtt{sp}_l$ for $addr = \mathtt{0x3d}$ and to the contents of $\mathtt{sp}_u$ for $addr = \mathtt{0x3e}$. We show the case for $addr = \mathtt{0x3d}$. The other case is shown analogously, replacing $\mathtt{sp}_l$ with $\mathtt{sp}_u$. So it is to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$. According to rule (t-in-sp-l) we get that $\mathtt{rda}_2(Rd) = \mathtt{rda}_1(\mathtt{sp}_l) \sqcup se(\mathsf{epselect}(s_1))$. So $\mathtt{rda}_2(Rd) = \mathcal{L}$ can only be the case if $\mathtt{rda}_1(\mathtt{sp}_l) = \mathcal{L}$ as well. Thus, by Assumption 3, one gets that $r_1(\mathtt{sp}_l) = r_1'(\mathtt{sp}_l)$. As we have by the semantics of `in` that $r_2(Rd) = r_1(\mathtt{sp}_l)$ and $r_2'(Rd) = r_1'(\mathtt{sp}_l)$ we can conclude $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{in}, (Rd, \mathtt{0x3f}))$*:* The instruction sets the content of the register Rd according to the current status registers C and Z. So it is to show that $\mathtt{rda}_2(Rd) = \mathcal{L} \Rightarrow r_2(Rd) = r_2'(Rd)$ to obtain indistinguishability.

Assume $\mathtt{rda}_2(Rd) = \mathcal{L}$. According to rule (t-in-stat) we obtain that $\mathtt{rda}_2(Rd) = \mathtt{srda}_1(C) \sqcup \mathtt{srda}_1(Z) \sqcup se(\mathsf{epselect}(s_1))$. This implies $\mathtt{srda}_1(C) = \mathcal{L}$ and $\mathtt{srda}_1(Z) = \mathcal{L}$. Together with Assumption 3 one obtains $sr_1(C) = sr_1'(C)$ and $sr_1(Z) = sr_1'(Z)$. According to the semantics of `in` this gives $r_2(Rd) = r_2'(Rd)$ what shows $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{out}, (addr, Rr))$ *for* $addr \in \{\mathtt{0x3d}, \mathtt{0x3e}\}$*:* According to the semantics of `out` with the given $addr$, $\mathtt{sp}_l$ is set to the contents of register Rr in the case of $addr = \mathtt{0x3d}$ or $\mathtt{sp}_u$ is set to the contents of register Rr in the case of $addr = \mathtt{0x3e}$. We show the case for $addr = \mathtt{0x3d}$. The other case is shown analogously, replacing $\mathtt{sp}_l$ with $\mathtt{sp}_u$. So it is to show that $\mathtt{rda}_2(\mathtt{sp}_l) = \mathcal{L} \Rightarrow r_2(\mathtt{sp}_l) = r_2'(\mathtt{sp}_l)$.

Assume $\mathtt{rda}_2(\mathtt{sp}_l) = \mathcal{L}$. According to rule (t-out-sp-l) we get that $\mathtt{rda}_2(\mathtt{sp}_l) = \mathtt{rda}_1(Rr) \sqcup se(\mathsf{epselect}(s_1))$. Thus, it has to be that $\mathtt{rda}_1(Rr) = \mathcal{L}$. By Assumption 3 it follows that $r_1(Rr) = r_1'(Rr)$. By the semantics of instruction `out` we get that $r_1(\mathtt{sp}_l) = r_1(Rr)$ and $r_2'(\mathtt{sp}_l) = r_1'(Rr)$. Together, one obtains $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{out}, (\mathtt{0x3f}, Rr))$*:* The instruction sets the content of status registers C and Z according to the contents of register Rr. So it is to show that $\mathtt{srda}_2(C) = \mathcal{L} \Rightarrow sr_2(C) = sr_2'(C)$ and $\mathtt{srda}_2(Z) = \mathcal{L} \Rightarrow sr_2(Z) = sr_2'(Z)$ to obtain indistinguishability.

According to (t-out-stat) we get that $\mathtt{srda}_2(C) = \mathtt{srda}_2(Z)$, so we can safely only assume either one of those to be $\mathcal{L}$.

Assume $\mathtt{srda}_2(C) = \mathcal{L}$. According to (t-out-stat) we get that $\mathtt{srda}_2(C) = \mathtt{rda}_1(Rr) \sqcup se(\mathsf{epselect}(s_1))$. This implies $\mathtt{rda}_1(Rr) = \mathcal{L}$ and together with Assumption 3 we obtain $r_1(Rr) = r_1'(Rr)$. As sr_2 and sr_2' do only depend on register Rr and the contents are equal in both cases, we can conclude that $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (epa))$ *for* $instr \in \{\mathtt{call}, \mathtt{jmp}, \mathtt{rcall}, \mathtt{rjmp}\}$*:* All control flow instructions have in common, that no data storages are modified. This directly gives $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$. They do however modify the next execution point, so we show $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$.

Instructions `jmp` and `rjmp` have $ep_2 = epa$ and $ep_2' = epa$ statically by the semantics of the instructions.

As we have by Assumption 1 that $ep_1 = ep_1'$ we can conclude that also $ep_2 = ep_2'$ in the case of `call` and `rcall`, as both execution points are extended by the semantics of the instructions in the same way.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{ret}, ())$*:* The `ret` instruction does not modify any data storages, giving $s_2 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2'$ directly. It does however modify the next execution point, so we show $\mathsf{epselect}(s_2) = \mathsf{epselect}(s_2')$.

As we have by Assumption 1 that $ep_1 = ep_1'$ we can conclude that also $ep_2 = ep_2'$, as the same comparison and computation is performed on both of them.

□

C.4 Locally Respect

Lemma 5 (Locally Respect). *Let* $P \in \mathtt{PROG}$ *be a program. For all states* $s_1, s_2 \in \mathtt{STATE}$*, clock cycles* $c \in \mathbb{N}$*, domain assignments* $\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1, \mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2$ *and security environments se, if*

1. $se(\mathsf{epselect}(s_1)) = \mathcal{H}$
2. $s_1 \xrightarrow{c}_P s_2$
3. $P, \cdots, \mathsf{epselect}(s_1) : (\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1) \vdash (\mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2)$
4. $\mathtt{sda}_1 \in \{\mathcal{H}\}^*$

then $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$*,* $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$*,* $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$*,* $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ *and* $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$*.*

Proof (Lemma 5 – Locally Respect). Let $P \in \mathtt{PROG}$ be a program. Let $s_1, s_2 \in \mathtt{STATE}$ be states, $c \in \mathbb{N}$ be a clock cycle, $\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1, \mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2$ be domain assignments and se be a security environment, such that

1. $se(\mathsf{epselect}(s_1)) = \mathcal{H}$
2. $s_1 \xrightarrow{c}_P s_2$
3. $P, \cdots, \mathsf{epselect}(s_1) : (\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1) \vdash (\mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2)$
4. $\mathtt{sda}_1 \in \{\mathcal{H}\}^*$

It is now to show that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$, $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

This case is proven by a case distinction over the possibly executed instruction $P(\mathsf{epselect}(s_1))$. For this, let $s_1 = (sr_1, m_1, r_1, st_1, ep_1)$ and $s_2 = (sr_2, m_2, r_2, st_2, ep_2)$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{push}, (Rr))$ The `push` instruction adds the contents of register Rr to the stack and decrements the stack pointer by one. Thus, the stack as well as the current register domain assignment is modified. From the typing rules one obtains that $spa = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and by Assumption 1, $spa = \mathcal{H}$. Furthermore, the domain assignments are updated as follows: $\mathtt{sda}_2 = spa :: \mathtt{sda}_1$, $\mathtt{rda}_2 = \mathtt{rda}_1[\mathtt{sp}_l \mapsto spa][\mathtt{sp}_u \mapsto spa]$. As $spa = \mathcal{H}$, all modified register's security domains are $\mathcal{H}$ after typing, such that $r_1 \approx_{\mathtt{rda}_2} r_2$ holds. As $\mathtt{sda}_1 \in \{\mathcal{H}\}^*$ by Assumption 4 and another $\mathcal{H}$ element is added to the stack domain assignment, one obtains $st_1 \simeq_{\mathtt{sda}_2} st_2$ by the definition of stack indistinguishability.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{pop}, (Rd))$ The `pop` instruction removes the topmost element from the stack and stores it in register Rd. Additionally the stack pointer is incremented by one. Thus, the stack as well as the current register domain assignment is modified. From the typing rules one obtains that $spa = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and by Assumption 1, $spa = \mathcal{H}$. Furthermore, the domain assignments of the stack pointer is updated by $\mathtt{sda}'_2 = spa :: \mathtt{sda}_1$, $\mathtt{rda}_2 = \mathtt{rda}_1[\mathtt{sp}_l \mapsto spa][\mathtt{sp}_u \mapsto spa]$. The register where the content is stored is updated by $\mathtt{sda}_2 = \mathtt{sda}'_2[Rd \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$ and is by Assumption 1 set to $\mathcal{H}$. Thus, all modified register's security domains are $\mathcal{H}$ after typing, such that $r_1 \approx_{\mathtt{rda}_2} r_2$ holds. As $\mathtt{sda}_1 \in \{\mathcal{H}\}^*$ by Assumption 4 and a $\mathcal{H}$ element is removed from the stack domain assignment, one obtains $st_1 \simeq_{\mathtt{sda}_2} st_2$ by the definition of stack indistinguishability.

Case $P(\mathsf{epselect}(s_1)) = (instr, (epa))$ *for* $instr \in \{\mathtt{breq}, \mathtt{brne}, \mathtt{brcc}, \mathtt{brcs}\}$ *and* $P(\mathsf{epselect}(s_1)) = (\mathtt{cpse}, (Rd, Rr))$ All those instructions have in common that no data storage is modified. That is, their small-step rules give $sr_1 = sr_2, m_1 = m_2, r_1 = r_2$ and $st_1 = st_2$. This directly gives $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$. As a consequence, their typing rules have $\mathtt{sda}_1 = \mathtt{sda}_2$, $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{rda}_1 = \mathtt{rda}_2$ and $\mathtt{srda}_1 = \mathtt{srda}_2$. This directly gives $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr, a))$ *for* $instr \in \{\mathtt{ld}, \mathtt{ldd}\}$ Instructions `ld` and `ldd` load values from memory into register Rd. Depending on a, they also increment or decrement the pointer register pair Rr. So another case distinction over values of a is required:

$a = \#$ *or* $instr = \mathtt{ldd}$*:* In this case, register pair Rr is left unmodified. From the typing rules one gets that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto (se(\mathsf{epselect}(s_1)) \sqcup \ldots)]$ while all other domain assignments are left unmodified. As Rd is the only modified register by the small-step rules and one has that $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1,

one gets $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ and $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$ in this case.

$a \in \{+,-\}$: In this case, register Rd and register pair Rr are modified by the small-step rules. Likewise, one gets that in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto sd][\mathsf{regconvert\text{-}l}(Rr) \mapsto sd][\mathsf{regconvert\text{-}u}(Rr) \mapsto sd]$ where $sd = se(\mathsf{epselect}(s_1)) \sqcup \ldots$. All other domain assignments are left unmodified. By Assumption 1 one gets that $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ and as such, the security domain of Rd and both registers of the register pair Rr are set to $\mathcal{H}$. Thus, one can conclude that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ and $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr, a))$ for $instr \in \{\mathtt{st}, \mathtt{std}\}$ Instructions `st` and `std` load values from register Rr into the memory pointed to by register Rd. Depending on a, they also increment or decrement the pointer register Rd. So another case distinction over values of a is required:

$a = \#$ or $instr = \mathtt{std}$: In this case, only the memory is modified. From the typing rules one gets that $\mathtt{md}_2 = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ while all other domain assignments are left unmodified. As only the memory is modified by the small-step rules and one has that $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, one gets $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ and $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$ in this case.

$a \in \{+,-\}$: In this case, the pointer register pair Rd and the memory are modified by the small-step rules. Likewise, one gets that in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[\mathsf{regconvert\text{-}l}(Rd) \mapsto sd][\mathsf{regconvert\text{-}u}(Rd) \mapsto sd]$ and $\mathtt{md}_2 = sd$ where $sd = se(\mathsf{epselect}(s_1)) \sqcup \ldots$. All other domain assignments are left unmodified. By Assumption 1 one gets that $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ and as such memory security domain as well as both pointer registers are set to $\mathcal{H}$. Thus, one can conclude that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ and $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{adc}, (Rd, Rr))$ Instruction `adc` modifies register Rd and status registers C and Z, according to contents of registers Rd, Rr and the carry flag C. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{sbc}, (Rd, Rr))$ Instruction `sbc` modifies register Rd and status registers C and Z, according to contents of registers Rd, Rr and the carry flag C. Status register Z also depends on the previous value of Z. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in

$\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{sbci}, (Rd, k))$ Instruction `sbci` modifies register Rd and status registers C and Z, according to contents of register Rd, the given immediate k and the carry flag C. Status register Z also depends on the previous value of Z. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr))$ *for* $instr \in \{\mathtt{add}, \mathtt{sub}\}$ Both possible instructions have in common, that they modify register Rd and status registers C and Z, according to contents of registers Rd and Rr. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, Rr))$ *for* $instr \in \{\mathtt{and}, \mathtt{or}\}$ Both possible instructions have in common, that they modify register Rd and status register Z, according to contents of registers Rd and Rr. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\mathtt{dec}, \mathtt{inc}\}$ Both possible instructions have in common, that they modify register Rd and status register Z, according to contents of registers Rd. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\mathtt{lsr}, \mathtt{neg}, \mathtt{ror}\}$ All possible instructions have in common, that they modify register Rd and status registers C and Z, according to contents of registers Rd. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$, $\mathtt{srda}_2 = \mathtt{srda}_1[C \mapsto se(\mathsf{epselect}(s_1)) \sqcup$

$\ldots][Z \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{eor}, (Rd, Rr))$ Instruction `eor` modifies register Rd and status register Z, according to contents of registers Rd and Rr. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto erg]$, $\mathtt{srda}_2 = \mathtt{srda}_1[Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{ldi}, (Rd, k))$ Instruction `ldi` modifies register Rd according to value k. As we have in the typing rules that $\mathtt{rda}_2 = \mathtt{rda}_1[Rd \mapsto se(\mathsf{epselect}(s_1))]$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$, $\mathtt{srda}_1 = \mathtt{srda}_2$ and security domains in $\mathtt{rda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cp}, (Rd, Rr))$ Instruction `cp` modifies status registers C and Z, according to contents of registers Rd and Rr. As we have in the typing rules that $\mathtt{srda}_2 = \mathtt{srda}_1[C \mapsto erg][Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{rda}_1 = \mathtt{rda}_2$, $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cpc}, (Rd, Rr))$ Instruction `cpc` modifies status registers C and Z, according to contents of registers Rd and Rr and status register C. As we have in the typing rules that $\mathtt{srda}_2 = \mathtt{srda}_1[C \mapsto erg][Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{rda}_1 = \mathtt{rda}_2$, $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains in $\mathtt{rda}_2$ and $\mathtt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_2$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_2$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_2$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\mathtt{cpi}, (Rd, k))$ Instruction `cpi` modifies status registers C and Z, according to contents of registers Rd and the given immediate value k. As we have in the typing rules that $\mathtt{srda}_2 = \mathtt{srda}_1[C \mapsto erg][Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\mathtt{sda}_2,\mathtt{md}_2,\mathtt{rda}_2,\mathtt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\mathtt{rda}_1 = \mathtt{rda}_2$, $\mathtt{md}_1 = \mathtt{md}_2$, $\mathtt{sda}_1 = \mathtt{sda}_2$ and security domains

in $\texttt{rda}_2$ and $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{clc}, ())$ Instruction `clc` resets the status register C to 0. As we have in the typing rules that $\texttt{srda}_2 = \texttt{srda}_1[C \mapsto se(\mathsf{epselect}(s_1))]$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$, $\texttt{rda}_1 = \texttt{rda}_2$ and security domains in $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{cli}, ())$ Instruction `cli` resets the interrupt flag I to 0. As we do not consider this flag, neither small-step rule (cli) nor typing rule (t-cli) has an effect. Thus, one gets directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$, and as all domain assignments remain unchanged, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{mov}, (Rd, Rr))$ Instruction `mov` copies the content of register Rr into register Rd. As we have in the typing rule that $\texttt{rda}_2 = \texttt{rda}_1[Rd \mapsto se(\mathsf{epselect}(s_1)) \sqcup \ldots]$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$, $\texttt{srda}_1 = \texttt{srda}_2$ and security domains in $\texttt{rda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{movw}, (Rd, Rr))$ Instruction `movw` copies the contents of register pair Rr into register pair Rd. As we have in the typing rules that $\texttt{rda}_2 = \texttt{rda}_1[Rd \mapsto erg][\mathsf{regconvert\text{-}u}(Rd) \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{srda}_1 = \texttt{srda}_2$, $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$ and security domains in $\texttt{rda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{subi}, (Rd, k))$ Instruction `subi` modifies register Rd and status registers C and Z, according to contents of register Rd and the given immediate k. As we have in the typing rules that $\texttt{rda}_2 = \texttt{rda}_1[Rd \mapsto erg]$, $\texttt{srda}_2 = \texttt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$ and security domains in $\texttt{rda}_2$ and $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{andi}, (Rd, k))$ Instruction `andi` modifies register Rd and status register Z, according to contents of register Rd and the given immediate k. As we have in the typing rules that $\texttt{rda}_2 = \texttt{rda}_1[Rd \mapsto erg]$, $\texttt{srda}_2 = \texttt{srda}_1[Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data

containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$ and security domains in $\texttt{rda}_2$ and $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (Rd, k))$ *for* $instr \in \{\texttt{adiw}, \texttt{sbiw}\}$ Both instructions modify register pair Rd and status registers C and Z, according to contents of register pair Rd and the given immediate k. As we have in the typing rules that $\texttt{rda}_2 = \texttt{rda}_1[Rd \mapsto erg][\mathsf{regconvert\text{-}u}(Rd) \mapsto erg]$, $\texttt{srda}_2 = \texttt{srda}_1[Z \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$ and security domains in $\texttt{rda}_2$ and $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{mul}, (Rd, Rr))$ Instruction `mul` modifies r_0 and r_1 and status registers C and Z, according to contents of registers Rd and Rr. As we have in the typing rules that $\texttt{rda}_2 = \texttt{rda}_1[r_0 \mapsto erg][r_1 \mapsto erg]$, $\texttt{srda}_2 = \texttt{srda}_1[Z \mapsto erg][C \mapsto erg]$ and $erg = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it follows directly that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ as all modified data containers are $\mathcal{H}$ after modification. As $\texttt{md}_1 = \texttt{md}_2$, $\texttt{sda}_1 = \texttt{sda}_2$ and security domains in $\texttt{rda}_2$ and $\texttt{srda}_2$ have only been set to $\mathcal{H}$, one gets $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{in}, (Rd, addr))$ *for* $addr \in \{\texttt{0x3d}, \texttt{0x3e}, \texttt{0x3f}\}$*:* According to the semantics of `in` with the given $addr$, the register Rd is set to the contents of $\texttt{sp}_l$ for $addr = \texttt{0x3d}$, to the contents of $\texttt{sp}_u$ for $addr = \texttt{0x3e}$ and to the contents of status registers C and Z in the case of $addr = \texttt{0x3f}$. So in all cases, only register Rd is modified.

According to $(\mathsf{t\text{-}in\text{-}*})$ one obtains that $\texttt{rda}_2(Rd) = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ while all other domain assignments are left unmodified. As $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, one directly obtains that $\texttt{rda}_2(Rd) = \mathcal{H}$ and as such, $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ is given. As only one register is set to $\mathcal{H}$ and nothing else is modified, $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$ also holds.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{out}, (addr, Rr))$ *for* $addr \in \{\texttt{0x3d}, \texttt{0x3e}\}$*:* According to the semantics of `out` with the given $addr$, $\texttt{sp}_l$ is set to the contents of register Rr in the case of $addr = \texttt{0x3d}$ or $\texttt{sp}_u$ is set to the contents of register Rr in the case of $addr = \texttt{0x3e}$. We show the case for $addr = \texttt{0x3d}$. The other case is shown analogously, replacing $\texttt{sp}_l$ with $\texttt{sp}_u$.

According to rule $(\mathsf{t\text{-}out\text{-}sp\text{-}*})$ one obtains that $\texttt{rda}_2(\texttt{sp}_l) = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and as $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it must be that $\texttt{rda}_2(\texttt{sp}_l) = \mathcal{H}$. As $\texttt{sp}_l$ is the only modified register and all other domain assignments are left unmodified, we can conclude $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ and $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{out}, (\texttt{0x3f}, Rr))$*:* The instruction sets the content of status registers C and Z according to the contents of register Rr.

According to rule (t-out-stat) one obtains that $\texttt{srda}_2(C) = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and as $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it must be that $\texttt{srda}_2(C) = \mathcal{H}$. Again, according to rule (t-out-stat) one obtains that $\texttt{srda}_2(Z) = se(\mathsf{epselect}(s_1)) \sqcup \ldots$ and as $se(\mathsf{epselect}(s_1)) = \mathcal{H}$ by Assumption 1, it must be that $\texttt{srda}_2(Z) = \mathcal{H}$.

As all modified status registers are $\mathcal{H}$ and all other domain assignments are left unmodified, it follows that $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ and $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (instr, (epa))$ *for* $instr \in \{\texttt{call}, \texttt{jmp}, \texttt{rcall}, \texttt{rjmp}\}$*:* All control flow instructions have in common, that no data storages are modified. As such, also no domain assignments are modified. This directly gives $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ and $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

Case $P(\mathsf{epselect}(s_1)) = (\texttt{ret}, ())$*:* The `ret` instruction does not modify and data storages. As such, also no domain assignments are modified. This directly gives $s_1 \approx_{\texttt{sda}_2,\texttt{md}_2,\texttt{rda}_2,\texttt{srda}_2} s_2$ and $\texttt{sda}_1 \sqsubseteq_s \texttt{sda}_2$, $\texttt{md}_1 \sqsubseteq \texttt{md}_2$, $\texttt{rda}_1 \sqsubseteq \texttt{rda}_2$ and $\texttt{srda}_1 \sqsubseteq \texttt{srda}_2$.

□

C.5 High Branching

Lemma 6 (High Branching). *Let* $P \in \texttt{PROG}$ *be a program. For all states* $s_1, s_1', s_2, s_2' \in \texttt{STATE}$*, clock cycles* $c, c' \in \mathbb{N}$*, domain assignments* $\texttt{sda}_1, \texttt{md}_1, \texttt{rda}_1, \texttt{srda}_1, \texttt{sda}_2, \texttt{md}_2, \texttt{rda}_2, \texttt{srda}_2$ *and security environments se, if*

1. $\mathsf{epselect}(s_1) = \mathsf{epselect}(s_1')$
2. $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$
3. $s_1 \approx_{\texttt{sda}_1,\texttt{md}_1,\texttt{rda}_1,\texttt{srda}_1} s_1'$
4. $s_1 \xrightarrow{c}_P s_2$
5. $s_1' \xrightarrow{c'}_P s_2'$
6. $P, \cdots, \mathsf{epselect}(s_1) : (\texttt{sda}_1, \texttt{md}_1, \texttt{rda}_1, \texttt{srda}_1) \vdash (\texttt{sda}_2, \texttt{md}_2, \texttt{rda}_2, \texttt{srda}_2)$

then $se(ep) = \mathcal{H}$ *for all* $ep \in \mathsf{region}_P(\mathsf{epselect}(s_1))$ *and* $\texttt{sda}_2 \in \{\mathcal{H}\}^*$.

Proof (Lemma 6 – High Branching). Let $P \in \texttt{PROG}$ be a program. Let $s_1, s_1', s_2, s_2' \in \texttt{STATE}$ be states, $c, c' \in \mathbb{N}$ be clock cycles, $\texttt{sda}_1, \texttt{md}_1, \texttt{rda}_1, \texttt{srda}_1, \texttt{sda}_2, \texttt{md}_2, \texttt{rda}_2, \texttt{srda}_2$ be domain assignments and se be a security environment, such that

1. $\mathsf{epselect}(s_1) = \mathsf{epselect}(s_1')$
2. $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$
3. $s_1 \approx_{\texttt{sda}_1,\texttt{md}_1,\texttt{rda}_1,\texttt{srda}_1} s_1'$
4. $s_1 \xrightarrow{c}_P s_2$
5. $s_1' \xrightarrow{c'}_P s_2'$

6. $P, \cdots, \mathsf{epselect}(s_1) : (\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1) \vdash (\mathtt{sda}_2, \mathtt{md}_2, \mathtt{rda}_2, \mathtt{srda}_2)$

It is now to show that $se(ep) = \mathcal{H}$ for all $ep \in \mathsf{region}_P(\mathsf{epselect}(s_1))$ and $\mathtt{sda}_2 \in \{\mathcal{H}\}^*$. For this, let $s_1 = (sr_1, m_1, r_1, st_1, ep_1)$ and $s_1' = (sr_1', m_1', r_1', st_1', ep_1')$.

As we have that $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$, a branching instruction must reside at $\mathsf{epselect}(s_1)$. We perform a case distinction on the possible branching instructions:

Case $P(\mathsf{epselect}(s_1)) \in \{(\mathit{breq}, (epa)), (\mathit{brne}, (epa))\}$*:* By small-step rules, the branching decision is based on the the Z flag. As we have by Assumption 2 that $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$ it follows that $sr_1(Z) \neq sr_1'(Z)$ and together with Assumption 3 that $s_1 \approx_{\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1} s_1'$ it must be that $\mathtt{srda}_1(Z) = \mathcal{H}$.

Together, the typing rule used in Assumption 6 must have been (t-brZ-h) which does exactly require that $se(ep) = \mathcal{H}$, $\forall ep' \in \mathsf{region}_P(\mathsf{epselect}(s_1)) : se(ep') = \mathcal{H}$ and $\mathtt{sda}_2 = \mathsf{lift}(\mathtt{sda}_1, \mathcal{H})$, which forces $\mathtt{sda}_2 \in \{\mathcal{H}\}^*$, what was to show.

Case $P(\mathsf{epselect}(s_1)) \in \{(\mathit{brcc}, (epa)), (\mathit{brcs}, (epa))\}$*:* By small-step rules, the branching decision is based on the the C flag. As we have by Assumption 2 that $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$ it follows that $sr_1(C) \neq sr_1'(C)$ and together with Assumption 3 that $s_1 \approx_{\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1} s_1'$ it must be that $\mathtt{srda}_1(C) = \mathcal{H}$.

Together, the typing rule used in Assumption 6 must have been (t-brC-h) which does exactly require that $se(ep) = \mathcal{H}$, $\forall ep' \in \mathsf{region}_P(\mathsf{epselect}(s_1)) : se(ep') = \mathcal{H}$ and $\mathtt{sda}_2 = \mathsf{lift}(\mathtt{sda}_1, \mathcal{H})$, which forces $\mathtt{sda}_2 \in \{\mathcal{H}\}^*$, what was to show.

Case $P(\mathsf{epselect}(s_1)) = (\mathit{cpse}, (Rd, Rr))$*:* By small-step rules, the branching decision is based on the the contents of registers Rd and Rr. As we have by Assumption 2 that $\mathsf{epselect}(s_2) \neq \mathsf{epselect}(s_2')$ it follows that (without loss of generality) $r_1(Rd) = r_1(Rr)$ and $r_1'(Rd) \neq r_1'(Rr)$. Together with Assumption 3 that $s_1 \approx_{\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1} s_1'$ it must be that at least one of $\mathtt{rda}_1(Rd) = \mathcal{H}$ or $\mathtt{rda}_1(Rr) = \mathcal{H}$ holds, as otherwise states s_1 and s_1' would have to agree on the contents of both registers.

Together, the typing rule used in Assumption 6 must have been (t-cpse-h) or (t-cpse-h2w) as $\mathtt{rda}_1(Rd) \sqcup \mathtt{rda}_1(Rr) = \mathcal{H}$. Both rules do exactly require that $se(ep) = \mathcal{H}$, $\forall ep' \in \mathsf{region}_P(\mathsf{epselect}(s_1)) : se(ep') = \mathcal{H}$ and $\mathtt{sda}_2 = \mathsf{lift}(\mathtt{sda}_1, \mathcal{H})$, which forces $\mathtt{sda}_2 \in \{\mathcal{H}\}^*$, what was to show.

At this point, all possible cases have been considered. □

C.6 Indistinguishability after High Branching

Lemma 7 (Indistinguishability after High Branching). *Let $P \in$ PROG be a program. For all natural numbers $n \in \mathbb{N}$, states $s_i \in$ STATE, clock cycles $c_i \in \mathbb{N}$, domain assignments $\mathtt{sda}_i, \mathtt{md}_i, \mathtt{rda}_i, \mathtt{srda}_i$, security environments se for each $i \in \{0, \ldots, n\}$, if*

1. $se(\mathsf{epselect}(s_i)) = \mathcal{H}$ *for all* $i \in \{0, \ldots, n\}$
2. $s_0 \xrightarrow{c_0}_P \cdots \xrightarrow{c_{n-1}}_P s_n$
3. *for all* $j, k \in \{0, \ldots, n\}$ *if* $\mathsf{epselect}(s_j) \rightsquigarrow_P \mathsf{epselect}(s_k)$ *there are* $\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'}$ *such that the judgment* $P, \cdots, \mathsf{epselect}(s_j) : (\mathtt{sda}_j, \mathtt{md}_j, \mathtt{rda}_j, \mathtt{srda}_j) \vdash (\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'})$ *is derivable and* $\mathtt{sda}_{k'} \sqsubseteq \mathtt{sda}_k$, $\mathtt{md}_{k'} \sqsubseteq \mathtt{md}_k$, $\mathtt{rda}_{k'} \sqsubseteq \mathtt{rda}_k$, $\mathtt{srda}_{k'} \sqsubseteq \mathtt{srda}_k$.
4. $\mathtt{sda}_0 \in \{\mathcal{H}\}^*$

then $s_0 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_n$.

Proof (Lemma 7 – Indistinguishability after High Branching). Let $P \in \mathtt{PROG}$ be a program, $n \in \mathbb{N}$ be a natural number, $s_i \in \mathtt{STATE}$ be states, $c_i \in \mathbb{N}$ be clock cycles, $\mathtt{sda}_i, \mathtt{md}_i, \mathtt{rda}_i, \mathtt{srda}_i$ be domain assignments for $i \in \{0, \ldots, n\}$ and se be a security environment such that

1. $se(ep) = \mathcal{H}$ for all $ep \in \{\mathsf{epselect}(s) \mid s \in \{s_0, \ldots, s_n\}$
2. $s_0 \xrightarrow{c_0}_P \cdots \xrightarrow{c_{n-1}}_P s_n$
3. for all $j, k \in \{0, \ldots, n\}$ if $\mathsf{epselect}(s_j) \rightsquigarrow_P \mathsf{epselect}(s_k)$ there are $\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'}$ such that the judgment $P, \cdots, \mathsf{epselect}(s_j) : (\mathtt{sda}_j, \mathtt{md}_j, \mathtt{rda}_j, \mathtt{srda}_j) \vdash (\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'})$ is derivable and $\mathtt{sda}_{k'} \sqsubseteq \mathtt{sda}_k$, $\mathtt{md}_{k'} \sqsubseteq \mathtt{md}_k$, $\mathtt{rda}_{k'} \sqsubseteq \mathtt{rda}_k$, $\mathtt{srda}_{k'} \sqsubseteq \mathtt{srda}_k$.
4. $\mathtt{sda}_0 \in \{\mathcal{H}\}^*$

It is now to show that $s_0 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_n$. We prove this by induction on the number of execution steps n in sequence 2.

Base Case: Assume $n = 0$, then $s_0 = s_n$ and we obtain $s_0 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_0$ as $\approx$ is reflexive by Lemma 1.

Induction Hypothesis: The proof goal and $\mathtt{sda}_i \sqsubseteq_s \mathtt{sda}_n$, $\mathtt{md}_i \sqsubseteq \mathtt{md}_n$, $\mathtt{rda}_i \sqsubseteq \mathtt{rda}_n$ and $\mathtt{srda}_i \sqsubseteq \mathtt{srda}_n$ for all $i < n$ is assumed to hold for all execution sequences that are shorter than n steps, whenever the prerequisites are satisfied.

Induction Step: Let $n \in \mathbb{N}$ be arbitrary and $n > 0$. Then it is to show that $s_0 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_n$. As s_0 is in a $\mathcal{H}$ security environment by Assumption 1, a judgment is derivable by Assumption 3 and $\mathtt{sda}_0 \in \{\mathcal{H}\}^*$ by Assumption 4, Lemma 5 (Locally Respect) is applicable. Applying Lemma 5 to s_0 gives $s_0 \approx_{\mathtt{sda}_1, \mathtt{md}_1, \mathtt{rda}_1, \mathtt{srda}_1} s_1$ and $\mathtt{sda}_0 \sqsubseteq_s \mathtt{sda}_1$, $\mathtt{md}_0 \sqsubseteq \mathtt{md}_1$, $\mathtt{rda}_0 \sqsubseteq \mathtt{rda}_1$ and $\mathtt{srda}_0 \sqsubseteq \mathtt{srda}_1$. Applying the induction hypothesis to the remaining execution sequence starting in s_1, which has now length $n' = n - 1$ and is as such smaller than n, leads to $s_1 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_n$, $\mathtt{sda}_1 \sqsubseteq_s \mathtt{sda}_n$, $\mathtt{md}_1 \sqsubseteq \mathtt{md}_n$, $\mathtt{rda}_1 \sqsubseteq \mathtt{rda}_n$ and $\mathtt{srda}_1 \sqsubseteq \mathtt{srda}_n$.

As $\approx$ is monotone by Lemma 2, one can conclude $s_0 \approx_{\mathtt{sda}_n, \mathtt{md}_n, \mathtt{rda}_n, \mathtt{srda}_n} s_n$, what was to show. □

C.7 Timing Indistinguishability after High Branching

Definition 50 (Branching Depth). *Let $P \in \texttt{PROG}$ be a program and $r \in \{then, else\}$. The function $\mathsf{branchdepth}_P : \texttt{EPS} \to \mathbb{N}$ is defined by*

$$\mathsf{branchdepth}_P^r(ep) := \begin{cases} 0 & \text{if } \neg\exists ep' \in \mathsf{region}_P^r(ep) : \mathsf{region}_P(ep') \neq \emptyset \\ 1 + \mathsf{max}(b) & \text{else} \end{cases}$$

where $b = \{\mathsf{branchdepth}_P^{r'}(ep') \mid ep' \in \mathsf{region}_P^r(ep) \wedge r' \in \{then, else\}\}$.

The following proof performs an induction over the branching depth, given by the $\mathsf{branchdepth}_P$ function. Intuitively, the branching depth denotes how many branches are nested inside each other. A branching depth of 0 means that there are no other branchings inside the control dependence region of a branching instruction.

Lemma 8 (Correctness of $\mathsf{branchtime}_P$). *Let $P \in \texttt{PROG}$ be a program. For all natural numbers $n \in \mathbb{N}$, states $s_0, \ldots, s_n \in \texttt{STATE}$, clock cycles $c_0, \ldots, c_{n-1} \in \mathbb{N}$ security environments se and $r \in \{then, else\}$, if*

1. *$P(\mathsf{epselect}(s_0))$ is a branching instruction*
2. *$se(\mathsf{epselect}(s_0)) = \mathcal{H}$*
3. *$\mathsf{epselect}(s_n) = \mathsf{jun}_P(\mathsf{epselect}(s_0))$*
4. *$\mathsf{epselect}(s_1) \in \mathsf{region}_P^r(\mathsf{epselect}(s_0))$*
5. *$s_0 \xrightarrow{c_0}_P \cdots \xrightarrow{c_{n-1}}_P s_n$*
6. *for all $j, k \in \{0, \ldots, n\}$ if $\mathsf{epselect}(s_j) \rightsquigarrow_P \mathsf{epselect}(s_k)$ there are $\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'}$ such that the judgment $P, \cdots, \mathsf{epselect}(s_j) : (\mathtt{sda}_j, \mathtt{md}_j, \mathtt{rda}_j, \mathtt{srda}_j) \vdash (\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'})$ is derivable and $\mathtt{sda}_{k'} \sqsubseteq \mathtt{sda}_k$, $\mathtt{md}_{k'} \sqsubseteq \mathtt{md}_k$, $\mathtt{rda}_{k'} \sqsubseteq \mathtt{rda}_k$, $\mathtt{srda}_{k'} \sqsubseteq \mathtt{srda}_k$.*

then $\mathsf{branchtime}_P^r(\mathsf{epselect}(s_0)) = \sum_{i=1}^{n-1} c_i$.

Lemma Correctness of $\mathsf{branchtime}_P$ shall ensure that the $\mathsf{branchtime}_P$ does what it is supposed to do. Namely, for a branching instruction (1) in a high security environment (2), it shall give the execution time of a typable (6) execution (5) up to the junction point (4).

Proof (Lemma 8 – Correctness of $\mathsf{branchtime}_P$). Let $P \in \texttt{PROG}$ be a program. Let $n \in \mathbb{N}$, $c_0, \ldots, c_{n-1} \in \mathbb{N}$ be clock cycles and $s_0, \ldots, s_n \in \texttt{STATE}$ be states, se be a security environment and let $r \in \{then, else\}$ such that

1. $P(\mathsf{epselect}(s_0))$ is a branching instruction
2. $se(\mathsf{epselect}(s_0)) = \mathcal{H}$
3. $\mathsf{epselect}(s_n) = \mathsf{jun}_P(\mathsf{epselect}(s_0))$
4. $\mathsf{epselect}(s_1) \in \mathsf{region}_P^r(\mathsf{epselect}(s_0))$
5. $s_0 \xrightarrow{c_0}_P \cdots \xrightarrow{c_{n-1}}_P s_n$

6. for all $j, k \in \{0, \ldots, n\}$ if $\mathsf{epselect}(s_j) \rightsquigarrow_P \mathsf{epselect}(s_k)$ there are $\mathtt{sda}_{k'}, \mathtt{md}_{k'}$, $\mathtt{rda}_{k'}, \mathtt{srda}_{k'}$ such that the judgment $P, \cdots, \mathsf{epselect}(s_j) : (\mathtt{sda}_j, \mathtt{md}_j, \mathtt{rda}_j, \mathtt{srda}_j) \vdash (\mathtt{sda}_{k'}, \mathtt{md}_{k'}, \mathtt{rda}_{k'}, \mathtt{srda}_{k'})$ is derivable and $\mathtt{sda}_{k'} \sqsubseteq \mathtt{sda}_k$, $\mathtt{md}_{k'} \sqsubseteq \mathtt{md}_k$, $\mathtt{rda}_{k'} \sqsubseteq \mathtt{rda}_k$, $\mathtt{srda}_{k'} \sqsubseteq \mathtt{srda}_k$.

it is now to show that $\mathsf{branchtime}^r_P(\mathsf{epselect}(s_0)) = \sum_{i=1}^{n-1} c_i$.

As $\mathsf{epselect}(s_0)$ is typable and in a high security environment, all typing rules of branching instructions require that $\neg\mathsf{loop}_P(\mathsf{epselect}(s_0))$. As such, one gets that no execution point is reached twice inside the sequence of Assumption 5. We prove this goal by induction over the branching depth of $\mathsf{epselect}(s_0)$.

Base Case: Let $\mathsf{branchdepth}^r_P(\mathsf{epselect}(s_0)) = 0$, that is, there are no further branching instructions inside the region of $\mathsf{epselect}(s_0)$. This gives that for all $ep \in \mathsf{region}_P(\mathsf{epselect}(s_0))$ it holds that $\mathsf{region}_P(ep) = \emptyset$. As such, the calculations using the $\mathsf{branchtime}^r_P$ function simplifies to

$$\mathsf{branchtime}^r_P(\mathsf{epselect}(s_0)) = \sum_{\mathsf{ep}_i \in \mathsf{region}^r_P(\mathsf{epselect}(s_0))} \mathsf{t}(P(\mathsf{ep}_i)) \tag{1}$$

Additionally, branching instructions are the only ones where it can happen that $c_i \neq \mathsf{t}(P(\mathsf{epselect}(s_i)))$. So we get $c_i = \mathsf{t}(P(\mathsf{epselect}(s_i)))$ for all $i \in \{1, \ldots, n-1\}$. Furthermore, no more branching instructions and Assumption 3 give $\mathsf{region}^r_P(\mathsf{epselect}(s_0)) = \{\mathsf{epselect}(s_i) \mid i \in \{1, \ldots, n-1\}\}$. Together with Equation (1) this gives

$$\mathsf{branchtime}^r_P(\mathsf{epselect}(s_0)) = \sum_{i=1}^{n-1} c_i$$

what was to show.

Induction Hypothesis: The proof goal is assumed to hold for all ep with $\mathsf{branchdepth}^r_P(ep) < m$.

Induction Step: To simplify argumentation, we split up the sum of the $\mathsf{branchtime}_P$ to

$$\mathsf{branchtime}^r_P(\mathsf{ep}) := \sum_{\mathsf{ep}_i \in \mathsf{region}^r_P(\mathsf{ep})} \mathsf{t}(P(\mathsf{ep}_i)) - \sum_{\mathsf{ep}_i \in \mathsf{region}^r_P(\mathsf{ep})} \mathsf{branchtime}^{then}_P(\mathsf{ep}_i) \tag{2}$$

Let $\mathsf{branchdepth}^r_P(\mathsf{epselect}(s_0)) = m$ with $m > 0$. Then there is at least one additional branching instruction inside the r region and its junction point is reached. Let $\mathsf{epselect}(s_j)$ point to an arbitrary reached branching instruction and let $k \in \mathbb{N}$ be such that $\mathsf{jun}_P(\mathsf{epselect}(s_j)) = \mathsf{epselect}(s_k)$. As $\mathsf{epselect}(s_j)$ points to a branching instruction, either the "then" or the "else" branch are executed.

Observe the positive sum in (2). For the control dependence regions it holds that $\mathsf{region}_P(\mathsf{epselect}(s_j)) \subseteq \mathsf{region}^r_P(\mathsf{epselect}(s_0))$, so the positive sum is too large as both possible branches are inside the positive sum. One has to make

sure that only one of the taken branches of $\mathsf{epselect}(s_j)$ is part of the result. We argue why this is correctly handled by the negative sum in (2).

By definition of $\mathsf{branchdepth}^r_P$, one gets that $\mathsf{branchdepth}^{r'}_P(\mathsf{epselect}(s_j)) < m$ for either $r' = then$ or $r' = else$. This gives, by the induction hypothesis, correctness for $\mathsf{branchtime}^{r'}_P(\mathsf{epselect}(s_j))$ for either $r' = then$ or $r' = else$. More concretely, let

$$\mathsf{branchtime}^{r'}_P(\mathsf{epselect}(s_j)) = \sum_{i=j+1}^{k-1} c_i \tag{3}$$

for either $r' = then$ or $r' = else$.

As $\mathsf{epselect}(s_j)$ is typable according to Assumption 6, one gets that

$$\mathtt{br} + \mathsf{branchtime}^{then}_P(\mathsf{epselect}(s_j)) = \mathsf{branchtime}^{else}_P(\mathsf{epselect}(s_j)) \tag{4}$$

For $r' = then$, one gets that $c_j = \mathsf{t}(P(\mathsf{epselect}(s_j))) + \mathtt{br}$ and for $r' = else$ one gets that $c_j = \mathsf{t}(P(\mathsf{epselect}(s_j)))$. Together, this gives

$$\mathsf{t}(P(\mathsf{epselect}(s_j))) + \mathsf{branchtime}^{else}_P(\mathsf{epselect}(s_j)) = \sum_{i=j}^{k-1} c_i$$

Note, that only $\mathsf{t}(P(\mathsf{epselect}(s_j)))$ is part of the positive sum in calculation (2). Inside the subtraction, it is thus correct to subtract $\mathsf{branchtime}^{then}_P(\mathsf{epselect}(s_j))$, as this leaves an additional $\mathtt{br}$ inside the sum and equals exactly the other branches execution time, which is not executed.

The given argument can be repeated for every other branching instruction found inside $\mathsf{region}^r_P(\mathsf{epselect}(s_0))$ to conclude the induction step.

Lemma 9 (Timing Indistinguishability after High Branching). *Let $P \in \mathtt{PROG}$ be a program. For all natural numbers $n, m \in \mathbb{N}$, states $s^1_i, s^2_j \in \mathtt{STATE}$, clock cycles $c^1_i, c^2_j \in \mathbb{N}$, domain assignments $\mathtt{sda}_{ep}, \mathtt{md}_{ep}, \mathtt{rda}_{ep}, \mathtt{srda}_{ep}$, security environments se for each $i \in \{0, \ldots, n\}$, each $j \in \{0, \ldots, m\}$ and each $ep \in \{\mathsf{epselect}(s) \mid s \in \{s^1_0, \ldots, s^1_n, s^2_0, \ldots s^2_m\}\}$, if*

1. *$P(\mathsf{epselect}(s^1_0))$ is a branching instruction*
2. *$\mathsf{epselect}(s^1_0) = \mathsf{epselect}(s^2_0)$*
3. *$se(\mathsf{epselect}(s^1_0)) = \mathcal{H}$*
4. *$\mathsf{epselect}(s^1_n) = \mathsf{epselect}(s^2_m) = \mathsf{jun}_P(\mathsf{epselect}(s^1_0))$*
5. *$s^1_0 \xrightarrow{c^1_0}_P \cdots \xrightarrow{c^1_{n-1}}_P s^1_n$*
6. *$s^2_0 \xrightarrow{c^2_0}_P \cdots \xrightarrow{c^2_{m-1}}_P s^2_m$*
7. *for all $ep_1, ep_2 \in \{\mathsf{epselect}(s) \mid s \in \{s^1_0, \ldots, s^1_n, s^2_0, \ldots s^2_m\}\}$ if $ep_1 \rightsquigarrow_P ep_2$ there are $\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2}$ such that the judgment $P, \cdots, ep_1 : (\mathtt{sda}_{ep_1}, \mathtt{md}_{ep_1}, \mathtt{rda}_{ep_1}, \mathtt{srda}_{ep_1}) \vdash (\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2})$ is derivable and $\mathtt{sda}'_{ep_2} \sqsubseteq \mathtt{sda}_{ep_2}$, $\mathtt{md}'_{ep_2} \sqsubseteq \mathtt{md}_{ep_2}$, $\mathtt{rda}'_{ep_2} \sqsubseteq \mathtt{rda}_{ep_2}$, $\mathtt{srda}'_{ep_2} \sqsubseteq \mathtt{srda}_{ep_2}$.*

then $\sum_0^n c^1_n = \sum_0^m c^2_m$.

Proof (Lemma 9 – Timing Indistinguishability after High Branching). Let $P \in \texttt{PROG}$ be a program. Let $n, m \in \mathbb{N}$ be natural numbers, $s_i^1, s_j^2 \in \texttt{STATE}$ be states, $c_i^1, c_j^2 \in \mathbb{N}$ be clock cycles, $\texttt{sda}_{ep}, \texttt{md}_{ep}, \texttt{rda}_{ep}, \texttt{srda}_{ep}$ be domain assignments for each $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, m\}$ and $ep \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$ and let se be a security environment, such that

1. $P(\mathsf{epselect}(s_0^1))$ is a branching instruction
2. $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2)$
3. $se(\mathsf{epselect}(s_0^1)) = \mathcal{H}$
4. $\mathsf{epselect}(s_n^1) = \mathsf{epselect}(s_m^2) = \mathsf{jun}_P(\mathsf{epselect}(s_0))$
5. $s_0^1 \xrightarrow{c_0^1}_P s_1^1 \xrightarrow{c_1^1}_P \cdots \xrightarrow{c_{n-1}^1}_P s_n^1$
6. $s_0^2 \xrightarrow{c_0^2}_P s_1^2 \xrightarrow{c_1^2}_P \cdots \xrightarrow{c_{m-1}^2}_P s_m^2$
7. for all $ep_1, ep_2 \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$ if $ep_1 \rightsquigarrow_P ep_2$ there are $\texttt{sda}'_{ep_2}, \texttt{md}'_{ep_2}, \texttt{rda}'_{ep_2}, \texttt{srda}'_{ep_2}$ such that the judgment $P, \cdots, ep_1 : (\texttt{sda}_{ep_1}, \texttt{md}_{ep_1}, \texttt{rda}_{ep_1}, \texttt{srda}_{ep_1}) \vdash (\texttt{sda}'_{ep_2}, \texttt{md}'_{ep_2}, \texttt{rda}'_{ep_2}, \texttt{srda}'_{ep_2})$ is derivable and $\texttt{sda}'_{ep_2} \sqsubseteq \texttt{sda}_{ep_2}$, $\texttt{md}'_{ep_2} \sqsubseteq \texttt{md}_{ep_2}$, $\texttt{rda}'_{ep_2} \sqsubseteq \texttt{rda}_{ep_2}$, $\texttt{srda}'_{ep_2} \sqsubseteq \texttt{srda}_{ep_2}$.

It is now to show that $\sum_{i=0}^{n-1} c_i^1 = \sum_{i=0}^{m-1} c_i^2$.

As $\mathsf{epselect}(s_0^1)$ is typable and in a high security environment, all typing rules of branching instructions require that $\neg\mathsf{loop}_P(\mathsf{epselect}(s_0^1))$. As such, one gets that no execution point is reached twice inside the sequences 5 and 6.

After this first observation, the goal is shown by a case distinction on the second reached execution point.

Case $\mathsf{epselect}(s_1^1) = \mathsf{epselect}(s_1^2)$*:* In this case, the same small-step rule must have been executed. This leads to $c_0^1 = c_0^2$. By application of Lemma 8 one gets that $\mathsf{branchtime}_P^r(\mathsf{epselect}(s_0^1)) = \sum_{i=1}^{n-1} c_i^1$ and $\mathsf{branchtime}_P^{r'}(\mathsf{epselect}(s_0^2)) = \sum_{i=1}^{m-1} c_i^2$. As $\mathsf{epselect}(s_1^1) = \mathsf{epselect}(s_1^2)$ we get that $r = r'$. Together, we obtain what was to show.

Case $\mathsf{epselect}(s_1^1) \neq \mathsf{epselect}(s_1^2)$*:* In this case, one gets without loss of generality that $\mathsf{epselect}(s_1^1) \in \mathsf{region}_P^{then}(\mathsf{epselect}(s_0^1))$ and $\mathsf{epselect}(s_1^2) \in \mathsf{region}_P^{else}(\mathsf{epselect}(s_0^1))$. This holds true because of SOAP1 that separates both following execution points into separate regions. It can be assumed without loss of generality, because sequences 5 and 6 are interchangeable.

Typability of the branching instruction at $\mathsf{epselect}(s_0^1)$ now gives the equality

$$n \cdot \mathtt{br} + \mathsf{branchtime}_P^{then}(\mathsf{epselect}(s_0^1)) = \mathsf{branchtime}_P^{else}(\mathsf{epselect}(s_0^1)) \tag{5}$$

as requirement for the values of the $\mathsf{branchtime}$ function, where $n = 2$ for the case where $(\mathsf{t\text{-}cpse\text{-}h2w})$ is used and $n = 1$ in all other cases.

Application of Lemma 8 now gives

$$\mathsf{branchtime}_P^{then}(\mathsf{epselect}(s_0^1)) = \sum_{i=1}^{n-1} c_i^1 \tag{6}$$

$$\mathsf{branchtime}_P^{else}(\mathsf{epselect}(s_0^1)) = \sum_{i=1}^{m-1} c_i^2 \tag{7}$$

Inspecting the small-step semantic rules for branching instructions give required clock cycles $c_0^1 = t + n \cdot \mathtt{br}$ for the "then" case and $c_0^2 = t$ for the "else" case, for some $t \in \mathbb{N}$.

This gives us the following equation

$$\begin{aligned}
\sum_{i=0}^{n-1} c_n^1 &= c_0^1 + \sum_{i=1}^{n-1} c_i^1 \\
&= t + n \cdot \mathtt{br} + \sum_{i=1}^{n-1} c_i^1 \\
&\overset{(6)}{=} t + n \cdot \mathtt{br} + \mathsf{branchtime}_P^{then}(\mathsf{epselect}(s_0^1)) \\
&\overset{(5)}{=} t + \mathsf{branchtime}_P^{else}(\mathsf{epselect}(s_0^1)) \\
&\overset{(7)}{=} t + \sum_{i=1}^{m-1} c_i^2 \\
&= \sum_{i=0}^{m-1} c_i^2
\end{aligned}$$

what was to show. □

C.8 Security of Typable Sequences

Lemma 10 (Security of Typable Sequences). *Let $P \in \mathtt{PROG}$ be a program. For all natural numbers $n, m \in \mathbb{N}$, states $s_i^1, s_j^2 \in \mathtt{STATE}$, clock cycles $c_i^1, c_j^2 \in \mathbb{N}$, domain assignments $\mathtt{sda}_{ep}, \mathtt{md}_{ep}, \mathtt{rda}_{ep}, \mathtt{srda}_{ep}$, security environments se for each $i \in \{0, \ldots, n\}$, each $j \in \{0, \ldots, m\}$ and each $ep \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$, if*

1. $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2)$
2. $\mathsf{epselect}(s_n^1) = \epsilon$
3. $se(\mathsf{epselect}(s_m^2)) = \mathcal{L}$
4. $s_0^1 \approx_{\mathtt{sda}_{ep}, \mathtt{md}_{ep}, \mathtt{rda}_{ep}, \mathtt{srda}_{ep}} s_0^2$ *for* $ep = \mathsf{epselect}(s_0^1)$.
5. $s_0^1 \xrightarrow{c_0^1}_P \cdots \xrightarrow{c_{n-1}^1}_P s_n^1$
6. $s_0^2 \xrightarrow{c_0^2}_P \cdots \xrightarrow{c_{m-1}^2}_P s_m^2$
7. *for all* $ep_1, ep_2 \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$ *if* $ep_1 \rightsquigarrow_P ep_2$ *there are* $\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2}$ *such that the judgment* $P, \cdots, ep_1 : (\mathtt{sda}_{ep_1}, \mathtt{md}_{ep_1}, \mathtt{rda}_{ep_1}, \mathtt{srda}_{ep_1}) \vdash (\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2})$ *is derivable and* $\mathtt{sda}'_{ep_2} \sqsubseteq \mathtt{sda}_{ep_2}$, $\mathtt{md}'_{ep_2} \sqsubseteq \mathtt{md}_{ep_2}$, $\mathtt{rda}'_{ep_2} \sqsubseteq \mathtt{rda}_{ep_2}$, $\mathtt{srda}'_{ep_2} \sqsubseteq \mathtt{srda}_{ep_2}$ *and se is the smallest security environment that can be used in the derivation.*

then there exists $d \in \mathbb{N}$ such that

1. $d \leq n$
2. $\mathsf{epselect}(s_d^1) = \mathsf{epselect}(s_m^2)$
3. $s_d^1 \approx_{\mathtt{sda}_{ep'},\mathtt{md}_{ep'},\mathtt{rda}_{ep'},\mathtt{srda}_{ep'}} s_m^2$ *for* $ep' = \mathsf{epselect}(s_d^1)$
4. $\sum_{i=0}^{d-1} c_i^1 = \sum_{i=0}^{m-1} c_i^2$

Proof (Lemma 10 – Security of Typable Sequences). Let $P \in \mathtt{PROG}$ be a program. Let $n, m \in \mathbb{N}$ be natural numbers, $s_i^1, s_j^2 \in \mathtt{STATE}$ be states, $c_i^1, c_j^2 \in \mathbb{N}$ be clock cycles, $\mathtt{sda}_{ep}, \mathtt{md}_{ep}, \mathtt{rda}_{ep}, \mathtt{srda}_{ep}$ be domain assignments for $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, m\}$ and $ep \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$ and let se be a security environment, such that

1. $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2)$
2. $\mathsf{epselect}(s_n^1) = \epsilon$
3. $se(\mathsf{epselect}(s_m^2)) = \mathcal{L}$
4. $s_0^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_0^2$ for $ep = \mathsf{epselect}(s_0^1)$
5. $s_0^1 \xrightarrow{c_0^1}_P \cdots \xrightarrow{c_{n-1}^1}_P s_n^1$
6. $s_0^2 \xrightarrow{c_0^2}_P \cdots \xrightarrow{c_{m-1}^2}_P s_m^2$
7. for all $ep_1, ep_2 \in \{\mathsf{epselect}(s) \mid s \in \{s_0^1, \ldots, s_n^1, s_0^2, \ldots s_m^2\}\}$ if $ep_1 \rightsquigarrow_P ep_2$ there are $\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2}$ such that the judgment $P, \cdots, ep_1 : (\mathtt{sda}_{ep_1}, \mathtt{md}_{ep_1}, \mathtt{rda}_{ep_1}, \mathtt{srda}_{ep_1}) \vdash (\mathtt{sda}'_{ep_2}, \mathtt{md}'_{ep_2}, \mathtt{rda}'_{ep_2}, \mathtt{srda}'_{ep_2})$ is derivable and $\mathtt{sda}'_{ep_2} \sqsubseteq \mathtt{sda}_{ep_2}, \mathtt{md}'_{ep_2} \sqsubseteq \mathtt{md}_{ep_2}, \mathtt{rda}'_{ep_2} \sqsubseteq \mathtt{rda}_{ep_2}, \mathtt{srda}'_{ep_2} \sqsubseteq \mathtt{srda}_{ep_2}$ and se is the smallest security environment that can be used in the derivation.

Now it is to show that there is a $d \in \mathbb{N}$ such that

1. $d \leq n$
2. $\mathsf{epselect}(s_d^1) = \mathsf{epselect}(s_m^2)$
3. $s_d^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_m^2$ for $ep = \mathsf{epselect}(s_d^1)$
4. $\sum_{i=0}^{d-1} c_i^1 = \sum_{i=0}^{m-1} c_i^2$

We prove this objective by induction over m, thus over the length of the execution sequence denoted by requirement 6.

Base Case: In the base case, it is $m = 0$, thus $d = 0$ is a valid choice. In this case we get $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2)$ by Assumption 1, $s_0^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_0^2$ for $ep = \mathsf{epselect}(s_d^1)$ by Assumption 4 and $\sum_{i=0}^{-1} c_i^1 = \sum_{i=0}^{-1} c_i^2$ is trivially satisfied, as both sides equal 0.

Induction Hypothesis: The proof goal is assumed to hold whenever $m' < m$ and the prerequisites are satisfied.

Induction Step: Assume $m > 0$. In this case, there is a transition $s_0^2 \xrightarrow{c_0^2}_P s_1^2$. By assumption 1, one gets $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2)$ and by Assumption 2 $\mathsf{epselect}(s_n^1) = \epsilon$. As there is a transition into s_1^2, both s_0^1 and s_0^2 can not be terminating states. Thus, one gets that $n > 0$ and the existence of state s_1^1

inside the execution sequence 5. We further perform a case distinction on the security environment of the first execution point $\mathsf{epselect}(s_0^2)$ in sequence 6.

Case $se(\mathsf{epselect}(s_0^2)) = \mathcal{L}$*:* The assumption in this case together with assumptions 1, 4, 5, 6 and 7 satisfy the prerequisites for Lemma 4 (Step Consistent). Application of this lemma gives $s_1^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_1^2$, $\mathsf{epselect}(s_1^1) = \mathsf{epselect}(s_1^2)$ and $c_1^1 = c_1^2$ for $ep = \mathsf{epselect}(s_1^1)$.

The remaining sequence starting from s_1^2 is now strictly shorter than m steps and by the results gained from locally respect, the assumptions 1-7 do still hold, such that applying the induction hypothesis concludes this case.

Case $se(\mathsf{epselect}(s_0^2)) = \mathcal{H}$*:* We perform an additional case distinction on $P(\mathsf{epselect}(s_0^2))$, whether it is a branching instruction or not.

Case $P(\mathsf{epselect}(s_0^2))$ *is a branching instruction:* By Lemma 6 one gets that every execution point inside the region of $\mathsf{epselect}(s_0^2)$ has a high execution environment and $\mathtt{sda}_{\mathsf{epselect}(s_0^2)} \in \{\mathcal{H}\}^*$. As we have by the Assumption 3 that $se(\mathsf{epselect}(s_m^2)) = \mathcal{L}$, there has to be a junction point of the branching instruction, as otherwise by Lemma 6 $se(\mathsf{epselect}(s_m^2))$ would be $\mathcal{H}$. Let s_c^2 denote the state at which this junction point is reached. That is, $\mathsf{jun}_P(\mathsf{epselect}(s_0^2)) = \mathsf{epselect}(s_c^2)$ has to hold. By Assumption 1 and the case that execution sequence 5 leads to termination, the execution starting in s_0^1 has to pass this junction point as well. Thus, choose $d \in \mathbb{N}$ such that $\mathsf{epselect}(s_d^1) = \mathsf{epselect}(s_c^2)$.

One can now apply Lemma 7 twice to obtain that $s_0^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_d^1$ and $s_0^2 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_c^2$ for $ep = \mathsf{epselect}(s_d^1) = \mathsf{epselect}(s_c^2)$. As $\approx$ is transitive by Lemma 3 one obtains by Assumption 4 and the previous indistinguishability result that

$$s_d^1 \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s_c^2 \tag{8}$$

By application of Lemma 9 one obtains directly that

$$\sum_{i=0}^{d-1} c_i^1 = \sum_{i=0}^{c-1} c_i^2 \tag{9}$$

The remaining sequence starting from state s_c^2 is now strictly shorter and the indistinguishability requirement is still satisfied by (8), allowing the application of the induction hypothesis together with the sequence starting from states s_d^1 and s_c^2. This gives a $d' \in \mathbb{N}$ such that $\mathsf{epselect}(s_{d'}^1) = \mathsf{epselect}(s_m^2)$, $s_{d'}^1 \approx_{\mathtt{sda}_{ep'},\mathtt{md}_{ep'},\mathtt{rda}_{ep'},\mathtt{srda}_{ep'}} s_m^2$ for $ep' = \mathsf{epselect}(s_{d'}^1)$ and $\sum_{i=d}^{n-1} c_i^1 = \sum_{i=c}^{m-1} c_i^2$. Those directly conclude goals 1 to 3. Goal 4 can be obtained by combining $\sum_{i=d}^{n-1} c_i^1 = \sum_{i=c}^{m-1} c_i^2$ with the result (9).

Case $P(\mathsf{epselect}(s_0^2))$ *is not a branching instruction:* As we are in a high security environment in this case, there has to be some branching instruction on secrets prior to s_0^2. This follows from Assumption 7, that se is minimal, and Lemma 6 (High Branching), which is the only way an execution point can obtain a $\mathcal{H}$ security environment. Otherwise, $P(\mathsf{epselect}(s_0^2))$ could not be high. That is, there has to be some $ep \in \mathtt{EPS}$ such that $\mathsf{epselect}(s_0^2) \in \mathsf{region}_P(ep)$. The remainder of this case is now shown analogously as in the previous case using

Lemma 7, Lemma 9 and the induction hypothesis on the junction point of the branching of *ep*.

This concludes both open case distinctions as all cases have been considered and as such the proof of the induction step is complete. □

C.9 Soundness

Proof (Theorem 1 – Soundness). Let $p \in \texttt{PROG}$ be a program and $\texttt{ep}_s \in \texttt{EPS}$ be an execution point. Let p be typable starting from $\texttt{ep}_s$ with initial domain assignments $\texttt{sda}_{\texttt{ep}_s}, \texttt{md}_{\texttt{ep}_s}, \texttt{rda}_{\texttt{ep}_s}, \texttt{srda}_{\texttt{ep}_s}$ and final domain assignments $\texttt{sda}_{\texttt{ep}_f}$, $\texttt{md}_{\texttt{ep}_f}, \texttt{rda}_{\texttt{ep}_f}, \texttt{srda}_{\texttt{ep}_f}$.

It is then to show that p satisfies TSNI starting from $\texttt{ep}_s$ with the same domain assignments. That is, let $s_0^1, s_0^2, s_n^1, s_m^2 \in \texttt{STATE}$ be states and $n, m, c^1, c^2 \in \mathbb{N}$ such that

1. $s_0^1 \approx_{\texttt{sda}_{\texttt{ep}_s},\texttt{md}_{\texttt{ep}_s},\texttt{rda}_{\texttt{ep}_s},\texttt{srda}_{\texttt{ep}_s}} s_0^2$
2. $\mathsf{epselect}(s_0^1) = \mathsf{epselect}(s_0^2) = \texttt{ep}_s$
3. $s_0^1 \Downarrow_P^{c^1} s_n^1$
4. $s_0^2 \Downarrow_P^{c^2} s_m^2$
5. $\mathsf{epselect}(s_n^1) = \mathsf{epselect}(s_m^2) = \texttt{ep}_f$

Now it has to hold that

Goal I $s_n^1 \approx_{\texttt{sda}_{\texttt{ep}_f},\texttt{md}_{\texttt{ep}_f},\texttt{rda}_{\texttt{ep}_f},\texttt{srda}_{\texttt{ep}_f}} s_m^2$
Goal II $c^1 = c^2$

Performing an expansion on the big-step semantics leads to

$$s_0^1 \xrightarrow{c_0^1}_P \cdots \xrightarrow{c_{n-2}^1}_P s_{n-1}^1 \xrightarrow{c_{n-1}^1}_P s_n^1 \tag{10}$$

and

$$s_0^2 \xrightarrow{c_0^2}_P \cdots \xrightarrow{c_{m-2}^2}_P s_{m-1}^2 \xrightarrow{c_{m-1}^2}_P s_m^2 \tag{11}$$

for some intermediate states $s_1^1, \ldots, s_{n-1}^1$ and $s_1^2, \ldots, s_{m-1}^2$ and clock cycles $c_0^1, \ldots, c_{n-1}^1$ and $c_0^2, \ldots, c_{m-1}^2$. Note, as rule (Ter) is the only way to reach termination and it requires the execution of a `ret` instruction, one gets that $n \geq 1$ and $m \geq 1$. According to rules (Seq) and (Ter), the times are the sum over each small-step timing, giving $c^1 = \sum_{i=0}^{n-1} c_i^1$ and $c^2 = \sum_{i=0}^{m-1} c_i^2$. And, according to (Ter), one gets that $\mathsf{epselect}(s_n^1) = \mathsf{epselect}(s_m^2) = \epsilon$ as both are terminating states according to assumptions 3 and 4.

The goal is now to apply Lemma 10 (Security of Typable Sequences) on the state sequences (10) and (11), where (11) is truncated by the last small-step execution, such that s_{m-1}^2 is the last state. We thus check the prerequisites of Lemma 10. By the observations above, prerequisites 2, 5 and 6 are satisfied. As P is typable with at least one security environment, let *se* be the smallest such. Prerequisite 7 is then given by the typability assumption and this smallest *se*. Prerequisites 4 is given by assumption 1, prerequisite 1 is given by assumption 2.

Prerequisite 3, that s^2_{m-1} is in a $\mathcal{L}$ security environment, is seen to be satisfied, as it has to point to a `ret` instruction in a $\mathcal{L}$ security environment. A `ret` instruction can only be in a $\mathcal{H}$ security environment by being inside a region that involves branchings on secrets. This is not possible by Assumption 1.

Now that Lemma 10 is applicable, its application as described gives a $d \leq n$, such that

1. $\mathsf{epselect}(s^1_d) = \mathsf{epselect}(s^2_{m-1})$
2. $s^1_d \approx_{\mathtt{sda}_{ep^1_d},\mathtt{md}_{ep^1_d},\mathtt{rda}_{ep^1_d},\mathtt{srda}_{ep^1_d}} s^2_{m-1}$, where $ep^1_d = \mathsf{epselect}(s^1_d)$
3. $\sum_{i=0}^{d-1} c^1_i = \sum_{i=0}^{m-2} c^2_i$

As state s^1_n is a terminating state as well, we get that s^1_{n-1} has to be a `ret` instruction. As this one is unique by Assumption 1, we get $d = n-1$ by the assumption about the execution sequence (10) and thus $s^1_{n-1} \approx_{\mathtt{sda}_{ep},\mathtt{md}_{ep},\mathtt{rda}_{ep},\mathtt{srda}_{ep}} s^2_{m-1}$, where $ep = \mathsf{epselect}(s^1_{n-1})$, and $\sum_{i=0}^{n-2} c^1_i = \sum_{i=0}^{m-2} c^2_i$.

As argued before, $\mathsf{epselect}(s^1_{n-1})$ and $\mathsf{epselect}(s^2_{m-1})$ have to point to a `ret` instruction and as such be in a $\mathcal{L}$ security environment. This, together by the results gained after applying Lemma 10, lead to that the prerequisites of Lemma 4 (Step Consistent) are satisfied. Applying Lemma 4 on s^1_{n-1} and s^2_{m-1} allows to conclude that

1. $s^1_n \approx_{\mathtt{sda}_{ep'},\mathtt{md}_{ep'},\mathtt{rda}_{ep'},\mathtt{srda}_{ep'}} s^2_m$, where $ep' = \mathsf{epselect}(s^1_n)$
2. $\mathsf{epselect}(s^1_n) = \mathsf{epselect}(s^2_m) = \mathsf{ep}_f$
3. $c^1_{n-1} = c^2_{m-1}$

From 1 and 2 we directly obtain Goal I and by combining 3 with the previous result that $\sum_{i=0}^{n-2} c^1_i = \sum_{i=0}^{m-2} c^2_i$, we obtain Goal II. □

References

1. Atmel: ATmega 8 Datasheet (02 2013), http://www.atmel.com/images/atmel-2486-8-bit-avr-microcontroller-atmega8_l_datasheet.pdf, rev.2486AA–AVR–02/2013
2. Atmel: Atmel AVR 8-bit Instruction Set: Instruction Set Manual (07 2014), http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf, rev. 0856J–AVR–07/2014