Fully Abstract Trace Semantics for Low-level Isolation Mechanisms – Extended Version

Marco Patrignani *Dave Clarke*

Report CW 651, November 2013

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Fully Abstract Trace Semantics for Low-level Isolation Mechanisms – Extended Version

Marco Patrignani *Dave Clarke*

Report CW 651, November 2013

Department of Computer Science, K.U.Leuven

Abstract

Many software systems adopt isolation mechanisms of modern processors as software security building blocks. Reasoning about these building blocks means reasoning about elaborate assembly code, which can be very complex due to the loose structure of the code. One way to overcome this complexity is providing the code with a more structured semantics. This paper presents one such semantics, namely a *fully abstract* trace semantics, for an assembly language enhanced with protection mechanisms of modern processors. The trace semantics represents the behaviour of protected assembly code with simple abstractions, unburdened by low-level details, at the maximum degree of precision. Furthermore, it captures the capabilities of attackers to protected software and simplifies providing a secure compiler targeting the assembly language.

Fully Abstract Trace Semantics for Low-level Isolation Mechanisms – Extended version

Marco Patrignani [*] Dave Clarke
iMinds-DistriNet, Dept. Computer Science, KU Leuven
{first.last}@cs.kuleuven.be

Abstract

Many software systems adopt isolation mechanisms of modern processors as software security building blocks. Reasoning about these building blocks means reasoning about elaborate assembly code, which can be very complex due to the loose structure of the code. One way to overcome this complexity is providing the code with a more structured semantics. This paper presents one such semantics, namely a *fully abstract* trace semantics, for an assembly language enhanced with protection mechanisms of modern processors. The trace semantics represents the behaviour of protected assembly code with simple abstractions, unburdened by low-level details, at the maximum degree of precision. Furthermore, it captures the capabilities of attackers to protected software and simplifies providing a secure compiler targeting the assembly language.

1. Introduction

Many processors provide isolation mechanisms as software security building blocks. These are used to withstand low-level attackers who, through injected assembly code, can read the whole memory space and access secrets in memory, violate integrity constraints and so on. With isolation mechanisms, attackers cannot directly violate security properties of isolated software since the isolated memory is not accessible to them. Examples of these protection mechanisms are fine grained, program counter-based memory access control mechanism [4, 10, 11, 14, 18–20], which enforce security properties at process or lower levels (Ring-1). With this mechanism, the software to be secured is placed in a protected memory partition that shields it from the surrounding, potentially malicious code. The malicious code cannot read nor write the protected memory; it can only jump to specific addresses in protected memory in order to call functions of the protected code. This protection mechanism makes software more resilient against low-level attackers. However, this does not prevent an attacker from interacting with the protected code and violating security properties.

To enable reasoning about the behaviour of isolated software, this paper firstly introduces an assembly language enhanced with a fine grained, program counter-based memory access control. This protection mechanism was chosen due to its application in secure compilation [2, 15] and to its recent industrial implementation: the Intel SGX [11]. Then, this paper explores the design space of trace semantics and presents two different *fully abstract* [16] trace semantics for the protected assembly code. The paper provides a general proof strategy where full abstraction of the trace semantics can be proven simply. To the best of the authors' knowledge, this is the first such result for an assembly language extended with the said protection mechanism. Moreover, the results seem applicable to other isolation mechanisms, modulo technical changes.

The fully abstract trace semantics provides a number of benefits. Trace semantics uses simple abstractions to represent the behaviour of protected assembly code, unburdened by low-level details, at the maximum degree of precision. So, dually, it models the behaviour of an attacker to the protected code, since it captures precisely the capabilities of that attacker. Secondly, without the trace semantics, the behaviour of protected assembly code (and thus of an attacker) is given in terms of low-level contexts. While being precise, low-level contexts are notoriously difficult to reason about, since they offer no inductive nor coinductive structure. The fully abstract trace semantics allows low-level contexts to be disregarded, since low-level contexts and traces are proven to be equally precise. Finally, the field of secure compilation benefits from the trace semantics. Recently, Agten *et al.* [2] and Patrignani *et al.* [15] presented secure compilers to assembly code enhanced with the aforementioned protection mechanism. Both results assume that the assembly language has a fully abstract trace semantics, thus the results of this paper strengthen those secure compilation claims.

In summary, the contributions of this paper are:

- the formalisation of an assembly language modelling an architecture enhanced with a fine grained, program counter-based memory access control;
- two different trace semantics for that low-level model;
- the proof that both trace semantics are fully abstract in a general proof strategy.

Limitations of this work are twofold. Firstly, the trace semantics cannot express side-channel attacks. Secondly, the formalisation does not consider details of the architecture such as caches; yet this is a often assumed [2, 13, 15, 17].

The paper is organised as follows. Section 2 introduces background notions for the work. Section 3 formalises syntax and operational semantics of the assembly language. Section 4 describes two trace semantics for the language. Section 5 contains the proof of full abstraction of both trace semantics. Section 6 describes related work. Section 7 discusses future works and concludes. The appendices present the proofs of the main theorems.

[*] Marco Patrignani holds a Ph.D. fellowship from the Research Foundation Flanders (FWO).

2. Protected Programs and Trace Semantics

This section describes the memory access control mechanism. Then, it discusses how to turn a trace semantics for protected assembly code into a fully abstract one, thus turning an imprecise representation of the behaviour of the code into a precise one.

2.1 The Protection Mechanism, Informally

The assembly language is enhanced with an isolation mechanism: a fine-grained, program counter-based memory access control mechanism [4, 10, 14, 18–20]. We review this mechanism from the work of Strackx and Piessens [19]. However, the techniques presented in this paper can be easily adapted to reasoning about other isolation mechanisms [4, 10, 18]. The protection mechanism provides a secure environment for running code that must be protected from the code it interacts with. This mechanism assumes that the memory is logically divided into a *protected* and an *unprotected* partition. The protected partition is further divided into a *code* and a *data* section. The code section contains a variable number of *entry points*: the only addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected partition. For the sake of simplicity, code in protected memory cannot read the unprotected one; as discussed in Section 2.2 below. Based on the location of the program counter, instructions that violate the access control policy are ruled out. The table below summarises the access control model enforced by the protection mechanism.

From\ To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	w x
Unprotected	x			r w x

2.2 From Naïve to Fully Abstract Trace Semantics

Following is the notation used in all code examples throughout this paper. Every instruction in the code is preceded by the address where it is located. Call the left program P_L and the right one P_R. When P_L and P_R are executed, they are placed in a protected memory partition spanning from address 100 to 200, with a single entry point at address 100.

EXAMPLE 1. *Both P_L and P_R assign to $\mathtt{r}_0$ the result of $\mathtt{r}_0 - \mathtt{r}_1$ (line 1). If the result of the operation is not less than 0 (line 3), they respectively write the contents of $\mathtt{r}_4$ and $\mathtt{r}_5$ at the unprotected address 10 (lines 4,5) and call a function whose address is stored in $\mathtt{r}_2$ (line 6). Otherwise, they assign different values to $\mathtt{r}_{11}$ (line 7) and return 0 (lines 8,9).*

```
1 | 100 sub r0 r1
2 | 101 movi r3 106
3 | 102 jl r3
4 | 103 movi r3 10
5 | 104 movs r3 r4
6 | 105 call r2
7 | 106 movi r11 41
8 | 107 movi r0 0
9 | 108 ret
```

```
1 | 100 sub r0 r1
2 | 101 movi r3 106
3 | 102 jl r3
4 | 103 movi r3 10
5 | 104 movs r3 r5
6 | 105 call r2
7 | 106 movi r11 42
8 | 107 movi r0 0
9 | 108 ret
```

In order to describe the behaviour of the protected code of Example 1, this paper defines a traces semantics for it. The trace semantics gives a simple characterisation of the behaviour of that code as a set of sequences of labels. In this paper, trace semantics are devised to capture the execution of a *protected* program, which is a program allocated in the protected memory partition. The execution of a program is the evaluation of a sequence of instructions; in order to abstract over instructions, not all of them generate a label in a trace. Labels are generated only by `call` and `ret` instructions. If they are executed by unprotected code, they are named *calls* and *returnbacks*, if they are executed by protected code they are named *callbacks* and *returns* [2, 8]. Following is the syntax of labels.

$$L ::= a \mid \tau \qquad a ::= g? \mid g! \qquad g ::= \mathtt{call}\ p(\overline{v}) \mid \mathtt{ret}\ v$$

A label L can be an observable action a or a non-observable action τ. Decorations ? and ! indicate the direction of the observable action: from unprotected to protected code (?) or vice-versa (!). Address p is an address in memory, $\overline{v}$ is a list of the contents of all registers and v indicates the contents of register $\mathtt{r}_0$.

Informally, a trace semantics is fully abstract when its labels capture all that is being communicated between the protected and the unprotected code. The trace semantics hinted at above is not fully abstract due to a number of subtleties, as highlighted by the following traces for the code of Example 1. Omitted details are indicated using . . .; use · to separate actions of the same trace.

$$\overline{a_1} = \mathtt{call}\ 100(1, 2, \ldots)? \cdot \mathtt{ret}\ 0!$$
$$\overline{a_2} = \mathtt{call}\ 100(2, 1, 40, \ldots)? \cdot \mathtt{call}\ 40(\ldots)!$$

Traces $\overline{a_1}$ and $\overline{a_2}$ are generated by both P_L and P_R. Trace $\overline{a_1}$ does not capture the different value stored in $\mathtt{r}_{11}$ (line 7), which, even if it is not the returned value of the function, still constitutes an observable difference between P_L and P_R. Trace $\overline{a_2}$ does not capture the different value written at address 10 (line 5), which also constitutes a observable difference between P_L and P_R. Since $\overline{a_1}$ and $\overline{a_2}$ do not capture the observable differences between P_L and P_R, the trace semantics fails to be fully abstract.

As Curien stated [3], two ways to achieve full abstraction for a trace semantics exist. The first is to modify the labels so that they capture more precisely what is communicated between protected and unprotected code. In this case, labels should capture the values of all registers and what protected code writes in unprotected memory. The second is to change the operational semantics, to restrict what is communicated to what is captured by the labels. This is achieved by restricting the ways in which communication is performed, e.g. by preventing writeouts. Both approaches are presented in Section 4; they rely on an assembly language and on its operational semantics which are formalised in Section 3.

Read-outs. As mentioned in Section 2.1, protected memory cannot read the unprotected one. This is enforced by some protection mechanisms [10] while it is allowed yet discouraged by others [2, 14, 15, 19]. We briefly discuss the implications of read-outs, to keep the rest of the paper simple.

Consider two programs that read-out values at two different locations, disregard those values and return 0. They do not present an observable difference, but if read-outs generated a visible action, the two programs would have different traces. This would break full abstraction so read-outs generate a τ action.

Now consider two programs that read-out values at two different locations and return those values. The read values must be known in order to determine the traces of the two programs, so the traces become parametrised on the read values. This can be achieved in different ways, to keep the presentation of this paper simple, a more thorough discussion of the subject is left for future work.

3. Assembly Language Formalisation

This section formalises the syntax and operational semantics of an assembly language, followed by the definition of contextual equivalence.

3.1 Syntax

The language is run on an architecture that models a Von Neumann machine consisting of a program counter p, a register file r, a flags register f and memory space m. The program counter indicates

the address of the instruction that is executed. The register file contains 12 general purpose registers R_0 to R_{11} and a stack pointer register SP, which contains the address of the top of the current call stack. The flags register contains a zero flag ZF and a sign flag SF, which are set or cleared by arithmetic instructions and are used by branching instructions. For the sake of simplicity, assume the architecture targeted by the language works with ℓ bit-long words, where ℓ is a power of 2. This allows the formalisation presented to scale to architectures with words of different sizes.

$$
\begin{array}{rlrl}
\textit{Words} & w ::= i \in \mathcal{I} & \textit{Memories} & m ::= \emptyset \\
 & \mid v \in \mathcal{V} & & \mid m; a \mapsto w \\
\textit{Empty word} & \mathbf{0} ::= 0^\ell & \textit{Numbers} & n ::= n \in \mathbb{N} \\
\textit{Addresses} & a \in 0..2^\ell - 1 & \textit{Programs} & P ::= (m, s) \\
 & & \textit{Values} & \mathcal{V} ::= \{v \mid v = [0 \textit{ or } 1]^\ell \wedge \forall i \in \mathcal{I}.\, v \neq i\} \\
 & & \textit{Memory descriptors} & s ::= (a_b, n_c, n_d, n)
\end{array}
$$

Figure 1. Elements of the assembly language formalisation.

Fig. 1 presents elements of the formalisation. Words w are either instructions i or values v. Values v are elements of the set $\mathcal{V}$, they are sequences of bits 0 or 1 of length ℓ that are different from the encoding of instructions. The explanation of why values are encoded in this way is delayed to Example 3 in Section 4.1. Instructions i are elements of the set $\mathcal{I}$ and define the programming language executed on the architecture (Fig. 2). The empty word $\mathbf{0}$

`movl` $\mathtt{r_d}$ $\mathtt{r_s}$	Load the word from the memory address in register $\mathtt{r_s}$ into register $\mathtt{r_d}$.
`movs` $\mathtt{r_d}$ $\mathtt{r_s}$	Store the contents of register $\mathtt{r_s}$ at the address found in register $\mathtt{r_d}$.
`movi` $\mathtt{r_d}$ `k`	Load the constant value `k` into register $\mathtt{r_d}$. Note that $\mathtt{k} < 2^\ell$.
`add` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} + \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set the ZF flag accordingly.
`sub` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} - \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set both the ZF and the SF flags accordingly.
`cmp` $\mathtt{r_1}$ $\mathtt{r_2}$	Calculate $\mathtt{r_1} - \mathtt{r_2}$ and set both the ZF and the SF flags accordingly.
`jmp` $\mathtt{r_i}$	Jump to the address located in register $\mathtt{r_i}$.
`je` $\mathtt{r_i}$	If the ZF flag is set, jump to the address in register $\mathtt{r_i}$.
`jl` $\mathtt{r_i}$	If the SF flag is set, jump to the address in register $\mathtt{r_i}$.
`call` $\mathtt{r_i}$	Push the value of the program counter +1 onto the stack and jump to the address in register $\mathtt{r_i}$.
`ret`	Pop a value from the stack and jump to the popped location.
`halt`	Stop the execution with the result in register $\mathtt{r_0}$.

Figure 2. Instruction set $\mathcal{I}$.

is a sequence of 0's whose length is based on the architecture being considered. Addresses a are natural numbers, ranging from 0 to $2^\ell - 1$. Memories m are maps from addresses to words. Memory access, denoted as $m(a)$, is defined as follows: $m(a) = w$ if $a \mapsto w \in m$; it is undefined otherwise. Define the domain of a memory as $\mathtt{dom}(m) = \{a \mid a \mapsto w \in m\}$. If two memories m and m' have disjoint domains, they can be merged in another memory. Formally, if $\mathtt{dom}(m) \cap \mathtt{dom}(m') = \emptyset$, then $m + m' = \{a \mapsto w \mid a \mapsto w \in m \text{ or } a \mapsto w \in m'\}$. Memory descriptors s are quadruples: (a_b, n_c, n_d, n) that formalise the concepts of Section 2.1: a_b is the address where the protected memory partition starts, n_c and n_d are the sizes (in number of addresses) of the code and data section respectively and n is the number of entry points. Entry points are allocated starting from the base address a_b. Each entry point is $\mathcal{N}_e$ words long. Assume the entry points do not overflow the protected code section, thus the constraint $n \cdot \mathcal{N}_e < n_c$ holds for the all memory descriptors. Programs P are pairs of a memory m and a memory descriptor s.

3.2 Operational Semantics

Before introducing the semantics, a number of auxiliary notions are defined.

Fig. 3 defines the access control enforcement rules informally presented in Section 2.1. Read judgments $s \vdash \mathtt{predicate}(a, b, \cdots)$ as: "according to s, `predicate` holds for addresses a, b, $\cdots$".

$$
\frac{a_b \leq p < (a_b + n_c + n_d)}{s \vdash \mathtt{protected}(p)}\ \text{(Aux-protected)}
\qquad
\frac{p < a_b}{s \vdash \mathtt{unprotected}(p)}\ \text{(Aux-unprotected1)}
$$

$$
\frac{(a_b + n_c + n_d) \leq p}{s \vdash \mathtt{unprotected}(p)}\ \text{(Aux-unprotected2)}
\qquad
\frac{p = a_b + (n-1) \cdot \mathcal{N}_e}{s \vdash \mathtt{returnEntryPoint}(p)}\ \text{(Aux-returnEntry)}
$$

$$
\frac{p = a_b + m \cdot \mathcal{N}_e \quad m \in \mathbb{N} \quad m < n}{s \vdash \mathtt{entryPoint}(p)}\ \text{(Aux-entryPoint)}
\qquad
\frac{(a_b + n_c) \leq p \quad p < (a_b + n_c + n_d)}{s \vdash \mathtt{data}(p)}\ \text{(Aux-data)}
$$

$$
\frac{s \vdash \mathtt{protected}(p) \quad s \vdash \mathtt{protected}(a)}{s \vdash \mathtt{readAllowed}(p, a)}\ \text{(Aux-read-1)}
\qquad
\frac{s \vdash \mathtt{unprotected}(p) \quad s \vdash \mathtt{unprotected}(a)}{s \vdash \mathtt{readAllowed}(p, a)}\ \text{(Aux-read-2)}
$$

$$
\frac{s \vdash \mathtt{unprotected}(a)}{s \vdash \mathtt{writeAllowed}(p, a)}\ \text{(Aux-write-1)}
\qquad
\frac{s \vdash \mathtt{protected}(p) \quad s \vdash \mathtt{data}(a)}{s \vdash \mathtt{writeAllowed}(p, a)}\ \text{(Aux-write-2)}
$$

$$
\frac{s \vdash \mathtt{unprotected}(p) \quad s \vdash \mathtt{entryPoint}(p')}{s \vdash \mathtt{entryJump}(p, p')}\ \text{(Aux-entry)}
\qquad
\frac{s \vdash \mathtt{protected}(p) \quad s \vdash \mathtt{unprotected}(p')}{s \vdash \mathtt{exitJump}(p, p')}\ \text{(Aux-return)}
$$

$$
\frac{s \vdash \mathtt{protected}(p) \quad s \vdash \mathtt{protected}(p') \quad s \nvdash \mathtt{data}(p')}{s \vdash \mathtt{intJump}(p, p')}\ \text{(Aux-internal)}
\qquad
\frac{s \vdash \mathtt{unprotected}(p) \quad s \vdash \mathtt{unprotected}(p')}{s \vdash \mathtt{extJump}(p, p')}\ \text{(Aux-external)}
$$

Figure 3. Access control enforcement rules. Assume $s \equiv (a_b, n_c, n_d, n)$

Define functions $\mathtt{m_{sec}}(m, s)$ and $\mathtt{m_{ext}}(m, s)$, which return the protected and unprotected parts of a memory m according to the descriptor s, respectively as: $\mathtt{m_{sec}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathtt{protected}(a)\}$ and $\mathtt{m_{ext}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathtt{unprotected}(a)\}$.

In the semantics there are two call stacks, one for the protected code, called the *secure stack*, and one for the unprotected code, called the *insecure stack*. Each stack is preceded by a word containing the location of the current head of the stack, let $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$ indicate the address of the secure and insecure stack respectively. When the current stack is changed in the semantics, the stack pointer register SP is initialised to the right address using $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$. Given a memory descriptor $s = (a_b, n_c, n_d, n)$, the secure stack starts at the beginning of the protected data section and the insecure stack starts at the end of the protected memory partition. Thus $\mathsf{SP}_{\mathsf{sec}} = (a_b + n_c)$ and, initially, $\mathsf{SP}_{\mathsf{sec}} \mapsto (a_b + n_c + 1)$; analogously, $\mathsf{SP}_{\mathsf{ext}} = (a_b + n_c + n_d)$ and, initially, $\mathsf{SP}_{\mathsf{ext}} \mapsto$

$(a_b + n_c + n_d + 1)$. Call and return instructions are responsible of setting the SP register to the correct address when crossing boundaries between protected and unprotected memory. The value of the program counter is pushed onto the stack by a `call` instruction (the stack grows down), while a `ret` instruction pops one address from the top of the stack and jumps to that location. Updating the stack pointer SP is performed by the auxiliary function `setStack` (Fig. 4). In the rules, notation $m[a \mapsto w]$ indicates that memory m is updated to a new one that is equal to m except that the value stored at address a is w. Notation $r[\mathsf{R} \mapsto w]$ indicates that the register file r is updated to a new one that is equal to r except that the value stored in register R is w. Notation $r(\mathsf{R})$ indicates the value contained in register R in register file r. Given a jump between addresses p and p', the stack switch rules produce a new register file r' and a new memory m' based on old ones r and m. The memory is updated to store the top of the current stack, located in SP, in the address storing the top of the current stack: $\mathsf{SP}_{\mathsf{sec}}$ or $\mathsf{SP}_{\mathsf{ext}}$. When the stack is changed, the register file is updated to initialise SP to the top of the right stack: the address stored at $\mathsf{SP}_{\mathsf{sec}}$ or $\mathsf{SP}_{\mathsf{ext}}$.

$$\frac{\begin{array}{c}\text{(Stack-out-to-in)}\\ s \vdash \texttt{entryJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{ext}} \mapsto r(\mathsf{SP})] \quad r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{sec}})] \\ s \vdash \texttt{unprotected}\, r(\mathsf{SP}) \quad s \vdash \texttt{protected}\, r'(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r', m'}$$

$$\frac{\begin{array}{c}\text{(Stack-in-to-out)}\\ s \vdash \texttt{exitJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{sec}} \mapsto r(\mathsf{SP})] \quad r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}})] \\ s \vdash \texttt{protected}\, r(\mathsf{SP}) \quad s \vdash \texttt{unprotected}\, r'(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r', m'}$$

$$\frac{\begin{array}{c}\text{(Stack-no-change-i)}\\ s \vdash \texttt{intJump}(p, p') \quad s \vdash \texttt{protected}\, r(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r, m}$$

$$\frac{\begin{array}{c}\text{(Stack-no-change-e)}\\ s \vdash \texttt{extJump}(p, p') \quad s \vdash \texttt{unprotected}\, r(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r, m}$$

Figure 4. Stack switch enforcement rules.

The operational semantics is a small step semantics that describes how each instruction of the language transforms an execution state into a new one. Thus, the operational semantics handles programs in the whole memory: both the protected and unprotected partitions.

DEFINITION 1 (Execution state). *An execution state, denoted as Ω, is a quintuple $\Omega = (p, r, f, m, s)$, where p is a program counter, r is a register file, f is a flags register, m is a memory and s is a memory descriptor.*

Given $\Omega = (p, r, f, m, s)$, let $\lfloor\Omega\rfloor$ be the state $(p, r, f, \mathtt{m}_{\mathtt{sec}}(m, s), s)$ and $\lceil\Omega\rceil$ be the state $(p, r, f, \mathtt{m}_{\mathtt{ext}}(m, s), s)$. Relations $\xrightarrow{i} \subseteq \lfloor\Omega\rfloor \times \lfloor\Omega\rfloor$ and $\xrightarrow{e} \subseteq \lceil\Omega\rceil \times \lceil\Omega\rceil$ describe the evaluation of instructions that only affect the protected and unprotected parts of memory respectively. Fig. 5 presents the rules for $\xrightarrow{i}$, rules for $\xrightarrow{e}$ are obtained by replacing an `intJump` assumption with an `extJump` one. Let $m(p) = \texttt{inst}$ denote that `inst` is the word allocated in $m(p)$, where $\texttt{inst} \in \mathcal{I}$. Note that the program counter is set to -1 whenever the `halt` instruction is encountered, in order to capture termination. This way, no progress can be made, as $m(-1)$ does not return a valid instruction: the program is in a stuck state.

DEFINITION 2 (Stuck state). *A state $\Omega = (p, r, f, m, s)$ is stuck, denoted as $\Omega^{\perp}$, when the program counter does not point to a valid instruction: $m(p) \notin \mathcal{I}$.*

The operational semantics of the language is a binary relation over states $\rightarrow \subseteq \Omega \times \Omega$ defined by the rules of Fig. 6. Rules Eval-callback and Eval-returnback ensure that the address to be followed after a callback is stored in the secure stack and that the address returnback entry point A_{rb} is pushed on the insecure stack. Thus the unprotected code always jumps to the returnback entry point when returning from a callback. Code located at the returnback entry point must contain a `ret` instruction in order to correctly resume the execution. Rule Eval-movs-out ensures that the values that protected programs write in unprotected memory are not instructions.

$$\frac{\lfloor\Omega\rfloor \xrightarrow{i} \lfloor\Omega'\rfloor}{\Omega \rightarrow \Omega'}\ \text{(Eval-protected)} \qquad \frac{\lceil\Omega\rceil \xrightarrow{e} \lceil\Omega'\rceil}{\Omega \rightarrow \Omega'}\ \text{(Eval-unprotected)}$$

$$\frac{\begin{array}{c}\text{(Eval-movs-out)}\\ m(p) = (\texttt{movs}\ \mathtt{r_d}\ \mathtt{r_s}) \quad s \vdash \texttt{intJump}(p, p+1) \\ s \vdash \texttt{writeAllowed}(p, r(\mathtt{r_d})) \quad s \vdash \texttt{unprotected}(r(\mathtt{r_d})) \\ m' = m[r(\mathtt{r_d}) \mapsto r(\mathtt{r_s})] \quad r(\mathtt{r_s}) \in \mathcal{V}\end{array}}{(p, r, f, m, s) \rightarrow (p+1, r, f, m', s)}$$

$$\frac{\begin{array}{c}\text{(Eval-callback)}\\ m(p) = (\texttt{call}\ \mathtt{r_d}) \quad p' = m(r(\mathtt{r_d})) \quad s \vdash \texttt{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \quad m' = m[r(\mathsf{SP}) \mapsto p + 1] \\ p, r', m', s \vdash \texttt{setStack} \searrow p', r'', m'' \\ r''' = r''[\mathsf{SP} \mapsto r''(\mathsf{SP}) + 1] \quad m''' = m''[r'''(\mathsf{SP}) \mapsto \mathsf{A}_{\mathsf{rb}}]\end{array}}{(p, r, f, m, s) \rightarrow (p', r''', f, m''', s)}$$

$$\frac{\begin{array}{c}\text{(Eval-call)}\\ m(p) = (\texttt{call}\ \mathtt{r_d}) \quad p' = m(r(\mathtt{r_d})) \quad s \vdash \texttt{entryJump}(p, p') \\ p, r, m, s \vdash \texttt{setStack} \searrow p', r', m' \\ r'' = r'[\mathsf{SP} \mapsto r'(\mathsf{SP}) + 1] \quad m'' = m'[r''(\mathsf{SP}) \mapsto p + 1]\end{array}}{(p, r, f, m, s) \rightarrow (p', r'', f, m'', s)}$$

$$\frac{\begin{array}{c}\text{(Eval-returnback)}\\ m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) = \mathsf{A}_{\mathsf{rb}} \quad s \vdash \texttt{entryJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \quad p, r', m, s \vdash \texttt{setStack} \searrow p', r'', m'\end{array}}{(p, r, f, m, s) \rightarrow (p', r'', f, m', s)}$$

$$\frac{\begin{array}{c}\text{(Eval-return)}\\ m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) \quad s \vdash \texttt{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \quad p, r', m, s \vdash \texttt{setStack} \searrow p', r'', m'\end{array}}{(p, r, f, m, s) \rightarrow (p', r'', f, m', s)}$$

Figure 6. Operational semantics of whole programs. A_{rb} is the address of the returnback entry point.

The transitive closure of relation $\rightarrow$ is indicated with $\rightarrow^*$. A state Ω performing n reduction steps is indicated as $\Omega \rightarrow^n \Omega'$. The evaluation of program P is a sequence of steps that takes the initial state of P to another state.

DEFINITION 3 (Initial state). *The initial state of a program (m, s), denoted as $\Omega_0(m, s)$, is the state (p_0, r_0, f_0, m, s), where $s = (a_b, n_c, n_d, n)$, $p_0 = (a_b + n_c + n_d + 2)$, $r_0 = [\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}}); \mathtt{r_i} \mapsto 0\ _{\mathtt{i}=0..11}]$, and $f_0 = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0]$.*

The evaluation of P terminates if $\Omega_0(P) \rightarrow^* \Omega^{\perp}$; the result of the computation is stored in $\mathtt{r_0}$. If the evaluation of program P does not terminate, P diverges. A program P diverges, denoted as $P\Uparrow$, if it executes an unbounded number of reduction steps. Formally: $P\Uparrow$ if $\forall n \in \mathbb{N}, \exists \Omega'.\ \Omega_0(P) \rightarrow^n \Omega'$.

3.3 Contextual Equivalence

Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [16]. This notion relies on the concept of contexts, which is introduced before presenting the equivalence itself.

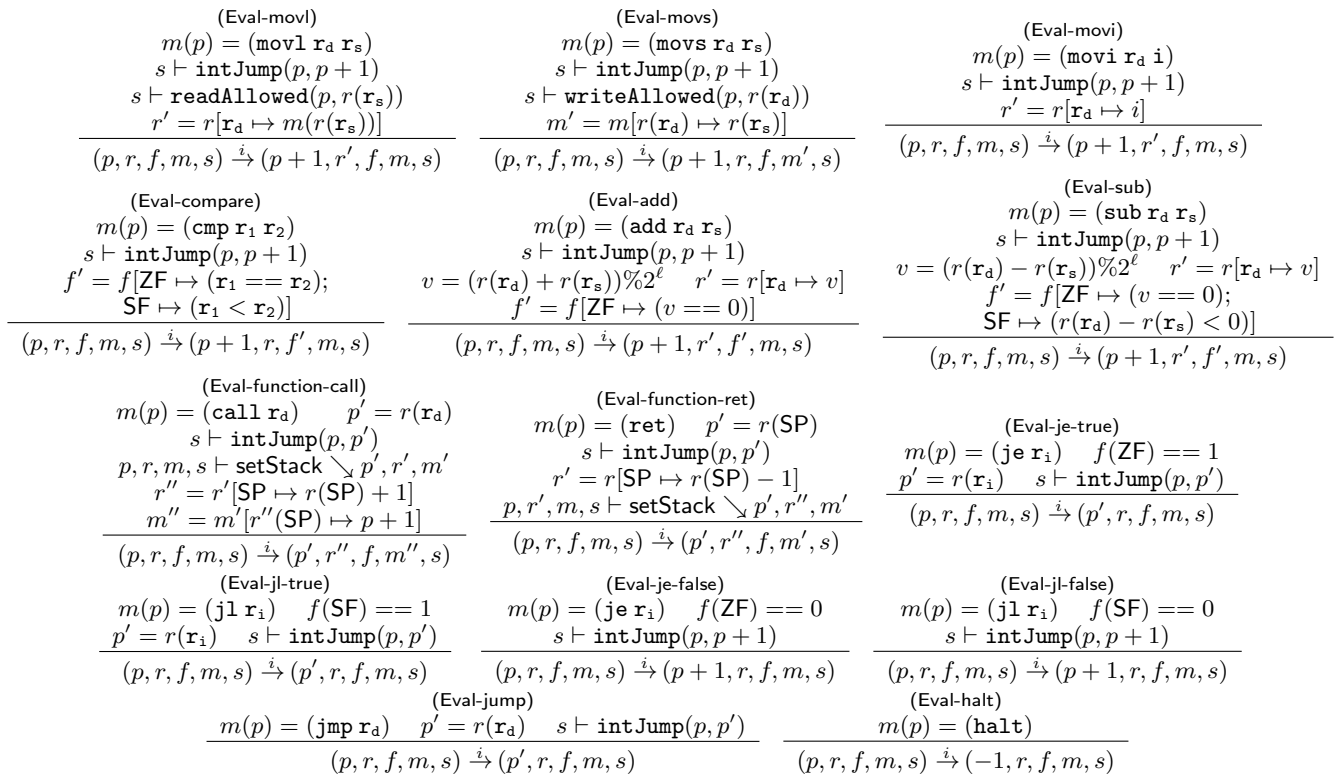

Figure 5. Operational semantics of instructions in the protected memory partition.

Since we consider programs P that are placed in protected memory and interact with arbitrary unprotected code, contexts model that unprotected code. Thus for any descriptor s, contexts $\mathbb{M}$ are partial memories with a hole: $\mathbb{M} = m[\cdot]$, where all addresses of $\mathbb{M}$ are unprotected. Formally, given s, $\forall a \in \texttt{dom}(\mathbb{M}), s \vdash \texttt{unprotected}(a)$. The hole models the possibility to combine a program P with the memory $\mathbb{M}$ iff they are compatible, denoted as $P \frown \mathbb{M}$, thus if the memories of P and $\mathbb{M}$ have disjoint domains. Let $\texttt{dom}(\mathbb{M}) = \texttt{dom}(m)$ if $\mathbb{M} = m[\cdot]$; formally, $P \frown \mathbb{M}$ if $P = (m', s)$ and $\texttt{dom}(m') \cap \texttt{dom}(\mathbb{M}) = \emptyset$. If P and $\mathbb{M}$ are compatible, the hole of $\mathbb{M}$ can be filled with P in order to model interaction between P and $\mathbb{M}$. Formally, if $P \frown \mathbb{M}$ then $\mathbb{M}[(m', s)] = (m' + m, s)$.

Programs P_1 and P_2 are contextually equivalent, denoted as $P_1 \simeq P_2$, when, for *all* contexts they interact with, P_1 diverges if and only if P_2 also diverges.

DEFINITION 4 (Contextual equivalence). *$P_1 \simeq P_2$ if* $\forall \mathbb{M}.\ P_1 \frown \mathbb{M} \wedge \mathbb{M}[P_1]\Uparrow \iff P_2 \frown \mathbb{M} \wedge \mathbb{M}[P_2]\Uparrow$.

An implication of this definition is that for P_1 and P_2 to be contextually equivalent they must have the same memory descriptor. For the sake of simplicity we will always assume the compatibility of a program and the context it is plugged in, shortening the above definition to: $P_1 \simeq P_2$ if $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \iff \mathbb{M}[P_2]\Uparrow$.

EXAMPLE 2 (Contextually equivalent programs). *The following P_L and P_R write the values of $\texttt{r}_1$ and $\texttt{r}_2$ respectively at the protected address 150 (line 2) and then return 0 (line 3). Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.*

```
1 100 movi r0 150
2 101 movs r0 r1
3 102 movi r0 0
4 103 ret
```

```
1 100 movi r0 150
2 101 movs r0 r2
3 102 movi r0 0
4 103 ret
```

The only difference between P_L and P_R is in the value stored at address 150. However, an unprotected program cannot read that value. Since that value does not affect the computation of P_L or P_R or the unprotected code, P_L and P_R are contextually equivalent.

Having defined the assembly language and its operational semantics, the paper introduces the two different trace semantics. Trace equivalence is also introduced, it will be proven the same as contextual equivalence in Section 5.

4. Trace Semantics

This section gives two different trace semantics for protected programs. The differences stem from the different ways to achieve full abstraction pointed out by Curien [3]. The first, Tr^L, possesses more expressive labels, while the second, Tr^S, relies on changes to the semantics of programs. Both are proven to be fully abstract w.r.t. the appropriate operational semantics in Section 5. Finally, this section defines when two programs are trace equivalent.

As for the operational semantics, a notion of execution states is required for the trace semantics as well. Execution states for Tr^L and Tr^S, denoted as Θ, are the same as Ω except that Θ do not deal with the whole memory but just with its protected partition. So, the memory m of (p, r, f, m, s) spans only the protected memory partition indicated by s. Additionally, Θ can be $(\mathsf{unknown}, m, s)$, an unknown state that models when code is executing in unprotected memory [8].

DEFINITION 5 (Initial state for traces). *The initial state for traces of a program* (m, s)*, denoted as* $\Theta_0(m, s)$*, is the state* $(\mathsf{unknown}, m, s)$.

4.1 Tr^{L}: Expressive Labels

Below are the labels exhibited by the Tr^{L} traces semantics.

$$\Lambda ::= \alpha \mid \tau_i \qquad \alpha ::= \surd \mid \gamma? \mid \delta!$$
$$\gamma ::= (r; f)\texttt{call}\ a \mid (r; f)\texttt{ret}\ a \qquad \delta ::= \texttt{write}(a, v).\delta \mid \gamma$$

A label Λ can be either an observable action α or a non-observable action τ_i. Action τ_i indicates the unobservable action occurred in protected memory. Observable actions include a tick $\surd$ indicating that the evaluation has terminated. Additionally, observable actions are function calls or returns to a certain address a, combined with the registers r and flags f. Registers and flags are in the labels as they convey information on the behaviour of programs. Observable, !-decorated actions can be prefixed by a number of write-outs of a value v to address a. They are only !-decorated since the protection mechanism forbids writes from unprotected to protected memory, as well as read-outs.

Fig. 7 presents the rules defining the relation $\Theta \overset{\overline{\alpha}}{\Longrightarrow} \Theta'$, which describe when a state Θ generates trace $\overline{\alpha}$ and results in state Θ'. The Tr^{L} traces of a program P is defined as follows: $\mathsf{Tr}^{\mathsf{L}}(P) = \{\overline{\alpha} \mid \exists \Theta'. \Theta_0(P) \overset{\overline{\alpha}}{\Longrightarrow} \Theta'\}$.

Example 3 below can now explain why protected code cannot write instructions in unprotected memory.

EXAMPLE 3 (Write-outs are not instructions). *In the following example, protected programs* can *write instructions in unprotected memory. Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.*

The following P_L *and* P_R *set* $\texttt{r}_0$ *to 20 and 10 respectively (line 1), then write the instruction* $\texttt{jmp}\ \texttt{r}_0$ *at address 20 and 10 respectively (line 2). Finally, they jump to the instruction they just wrote (line 3).*

```
1 | 100 movi r0 20
2 | 101 movs r0 "jmp r0"
3 | 102 call r0
```

```
1 | 100 movi r0 10
2 | 101 movs r0 "jmp r0"
3 | 102 call r0
```

When $\texttt{r}_0$ *is set to 20 (resp. 10), the instruction* $\texttt{jmp}\ \texttt{r}_0$ *written at address 20 (resp. 10) will diverge. Thus,* P_L *and* P_R *are contextually equivalent, since no context can differentiate between them. However,* P_L *and* P_R *are trace inequivalent, since the following is a trace of* P_L *and not of* P_R*.*

$$(\cdots)\texttt{call}\ 100? \cdot \texttt{write}(20, \text{“}\texttt{jmp}\ \texttt{r}_0\text{”}).(\cdots)\texttt{call}\ 20!$$

Since we are interested in programs that do not extend the functionality of code in unprotected memory (and since these programs are the majority of assembly programs), protected code cannot write instructions in unprotected memory. Thus the programs of this example are not valid.

4.2 Tr^{S}: Changes to the Semantics

The Tr^{S} semantics presents different labels from Tr^{L}, which are driven by changes to the operational semantics of protected programs. These changes are formalised in Fig. 8; smaller changes to other rules are omitted. When the program counter jumps between the protected and the unprotected memory partitions, or vice-versa, flags are set to 0; in case of a `return`, all registers but R_0 are also set to 0. Additionally, write-outs are prohibited.

Following are the labels of Tr^{S}; we drop the greek letter notation and use latin letters in order to immediately differentiate between Tr^{L} and Tr^{S}. Flags do not appear in traces because they are always set to 0, as are all registers but R_0 in case of a `return`. Write-outs

$$\text{(Trace-internal)}\quad \frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \quad s \vdash \texttt{intJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\tau_i} (p',r',f',m',s)}$$

$$\text{(Trace-internal-tick)}\quad \frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \quad s \vdash \texttt{protected}(p) \quad (p',r',f',m',s)^{\perp}}{(p,r,f,m,s) \xrightarrow{\surd} (p',r',f',m',s)}$$

$$\text{(Trace-writeout-accumulate)}\quad \frac{s \vdash \texttt{intJump}(p,p+1) \quad m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{d})) \quad r(\texttt{r}_\texttt{s}) \in \mathcal{V} \quad (p+1,r,f,m,s) \overset{\delta!}{\Longrightarrow} (p',r',f',m',s)}{(p,r,f,m,s) \xrightarrow{\texttt{write}(r(\texttt{r}_\texttt{d}),r(\texttt{r}_\texttt{s})).\delta!} (p',r',f',m',s)}$$

$$\text{(Trace-writeout-discard)}\quad \frac{s \vdash \texttt{intJump}(p,p+1) \quad m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{d})) \quad (p+1,r,f,m,s) \overset{\surd}{\Longrightarrow} (p',r',f',m',s)}{(p,r,f,m,s) \xrightarrow{\surd} (p',r',f',m',s)}$$

$$\text{(Trace-call)}\quad \frac{s \vdash \texttt{entryPoint}(p)}{(\mathsf{unknown},m,s) \xrightarrow{(r;f)\texttt{call}\ p?} (p,r,f,m,s)}$$

$$\text{(Trace-returnback)}\quad \frac{s \vdash \texttt{returnEntryPoint}(p)}{(\mathsf{unknown},m,s) \xrightarrow{(r;f)\texttt{ret}\ p?} (p,r,f,m,s)}$$

$$\text{(Trace-callback)}\quad \frac{s \vdash \texttt{exitJump}(p,p') \quad m(p) = (\texttt{call}\ p') \quad r' = r[\mathsf{SP} \mapsto r(\mathsf{SP})+1] \quad m' = m[r(\mathsf{SP}) \mapsto p+1]}{(p,r,f,m,s) \xrightarrow{(r;f)\texttt{call}\ p'!} (\mathsf{unknown},m',s)}$$

$$\text{(Trace-return)}\quad \frac{p' = m(\mathsf{SP}) \quad s \vdash \texttt{exitJump}(p,p') \quad m(p) = (\texttt{ret})}{(p,r,f,m,s) \xrightarrow{(r;f)\texttt{ret}\ p'!} (\mathsf{unknown},m,s)}$$

$$\text{(Trace-refl)}\quad \frac{}{\Theta \overset{\epsilon}{\Longrightarrow} \Theta} \qquad \text{(Trace-tau-i)}\quad \frac{\Theta \xrightarrow{\tau_i} \Theta'}{\Theta \overset{\epsilon}{\Longrightarrow} \Theta'} \qquad \text{(Trace-trans)}\quad \frac{\Theta \overset{\overline{\alpha}}{\Longrightarrow} \Theta'' \quad \Theta'' \overset{\overline{\alpha}'}{\Longrightarrow} \Theta'}{\Theta \overset{\overline{\alpha}\overline{\alpha}'}{\Longrightarrow} \Theta'} \qquad \text{(Trace-action)}\quad \frac{\Theta \xrightarrow{\alpha} \Theta'}{\Theta \overset{\alpha}{\Longrightarrow} \Theta'}$$

Figure 7. Rules of the Tr^{L} trace semantics.

are prohibited, so there are no labels that capture them.

$$L ::= a \mid \tau_i \quad a ::= \surd \mid g? \mid g! \quad g ::= \texttt{call}\ p\ (r) \mid \texttt{ret}\ p\ r(\texttt{r}_0)$$

Rules defining the relation $\Theta \overset{\overline{a}}{\Longrightarrow} \Theta'$ for Tr^{S} are omitted. They are a subset of the ones defined in Fig. 7, with a slight modification to the syntax of generated labels. The Tr^{S} traces of a program P is defined as follows: $\mathsf{Tr}^{\mathsf{S}}(P) = \{\overline{a} \mid \exists \Theta.\ \Theta_0(P) \overset{\overline{a}}{\Longrightarrow} \Theta\}$.

4.3 Trace Equivalence

The notion of trace equivalence is presented generically for both trace semantics under consideration. Use $\mathsf{Tr}(P)$ to indicate the traces of a program P, be it expressed through Tr^{L} or Tr^{S}. Two programs P_1 and P_2 are trace-equivalent, denoted as $P_1 \simeq_{\mathrm{T}} P_2$, if their traces are the same and they have the same memory descriptor.

DEFINITION 6 (Trace equivalence). $P_1 \simeq_{\mathrm{T}} P_2$ *if* $\mathsf{Tr}(P_1) = \mathsf{Tr}(P_2)$ *and* $P_1 = (m_1, s)$ *and* $P_2 = (m_2, s)$.

Following are two examples of trace equivalent and inequivalent programs. For the sake of simplicity, we use the Tr^{L} semantics and

$$\frac{s \vdash \texttt{unprotected}(p) \quad s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{writeAllowed}(p,a)}\ \text{(Aux-write-1')}$$

$$\frac{\begin{array}{c}s \vdash \texttt{entryJump}(p,p') \quad m' = m[\mathsf{SP}_{\mathsf{ext}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{sec}})] \quad f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \texttt{unprotected}\, r(\mathsf{SP}) \quad s \vdash \texttt{protected}\, r'(\mathsf{SP})\end{array}}{p,r,f,m,s \vdash \texttt{setStack} \searrow p', r', f', m'}\ \text{(Stack-out-to-in')}$$

$$\frac{\begin{array}{c}s \vdash \texttt{exitJump}(p,p') \quad m' = m[\mathsf{SP}_{\mathsf{sec}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}})] \quad f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \texttt{protected}\, r(\mathsf{SP}) \quad s \vdash \texttt{unprotected}\, r'(\mathsf{SP})\end{array}}{p,r,f,m,s \vdash \texttt{setStack} \searrow p', r', f', m'}\ \text{(Stack-in-to-out')}$$

$$\frac{\begin{array}{c}m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) \quad s \vdash \texttt{exitJump}(p,p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1; \mathsf{R_i} \mapsto 0_{\mathsf{i}=0..11}] \\ p,r',f,m,s \vdash \texttt{setStack} \searrow p', r'', f', m'\end{array}}{(p,r,f,m,s) \to (p',r'',f',m',s)}\ \text{(Eval-return')}$$

Figure 8. Changes to auxiliary functions and to the operational semantics for Tr^{S}.

indicate arbitrary values for registers and flags with notation (r, f) and an unprotected address with p.

EXAMPLE 4 (Traces of previous examples). *The code of Example 1 is* not *trace equivalent; the following trace is generated by P_L but not by P_R:*

$$(r; f)\texttt{call}\ 100? \cdot (\ldots, 41; f)\texttt{ret}\ p! \cdot \surd$$

The code of Example 2 is trace equivalent since the trace semantics of both P_L and P_R is a set whose sequences are concatenations of the following trace, each element of the sequence being parametrised on r and f:

$$(r; f)\texttt{call}\ 100? \cdot (r[\mathsf{R}_0 \mapsto 0]; f)\texttt{ret}\ p!$$

5. Full Abstraction of the Trace Semantics

This section presents the general proof strategy through which both Tr^{L} and Tr^{S} are proven to be fully abstract w.r.t. the corresponding operational semantics and discusses the benefits of full abstraction of the trace semantics.

A fully abstract trace semantics is both sound and complete with respect to the operational semantics. Soundness states that the trace semantics captures all details of the operational semantics. Thus, for all contexts, two trace equivalent programs cannot be told apart. The universal quantification over contexts makes this more difficult to prove. Completeness states that abstractions of the trace semantics are preserved in the operational semantics. This is achieved by proving the contrapositive of the completeness statement. For this, the proof consists of devising an algorithm that constructs a program, a "witness", that tells two trace inequivalent programs apart [2, 7, 15]. Since the language is low-level and untyped, this is not particularly complicated.

Full abstraction of trace semantics is formally stated as: $P_1 \simeq_{\mathrm{T}} P_2 \iff P_1 \simeq P_2$; its proof is split in two cases, one for each direction of the co-implication.

THEOREM 1 (Soundness). $P_1 \simeq_{\mathrm{T}} P_2 \Rightarrow P_1 \simeq P_2$.

Call the *interface* of a state its registers, flags and unprotected memory. Two states Ω_1 and Ω_2 have the same interface, denoted as $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, if they have the same registers, flags and unprotected memory. Formally, $\Omega_1 \mathrel{\mathring{=}} \Omega_2$ if $\Omega_1 = (p_1, r, f, m_1, s_1)$ and $\Omega_2 = (p_2, r, f, m_2, s_2)$ and $\mathtt{m}_{\mathtt{ext}}(m_1, s_1) = \mathtt{m}_{\mathtt{ext}}(m_2, s_2)$. Given $\Omega = (p, r, f, m, s)$, redefine $\lfloor\Omega\rfloor$ to be state $\Theta = (p, r, f, \mathtt{m}_{\mathtt{sec}}(m, s), s)$ if $s \vdash \texttt{protected}(p)$ and $(\texttt{unknown}, m, s)$ otherwise.

The proof of Theorem 1 states that an unprotected program interacting with P_1 cannot distinguish it from P_2. This is because both programs offer the same interface to the unprotected program. So this proof depends on an interface-preservation lemma (Lemma 1) which must be proven for each trace semantics since it depends on the labels of each trace semantics. Lemma 1 enunciates that two states with the same interface still have the same interface after they perform the same observable action. Thus unprotected programs do not see differences, in terms of flags, registers and unprotected memory, between P_1 and P_2.

LEMMA 1 (Interface preservation after same observable action). *If $\Theta_1 \overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\!\rightarrow \overset{\alpha}{\longrightarrow}\!\!\!\!\!\rightarrow\ \Theta_1'$ and $\Theta_1 = \lfloor\Omega_1\rfloor$ and $\Omega_1 \to^* \Omega_1'$ and $\Theta_1' = \lfloor\Omega_1'\rfloor$ and $\Theta_2 \overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\!\rightarrow \overset{\alpha}{\longrightarrow}\!\!\!\!\!\rightarrow\ \Theta_2'$ and $\Theta_2 = \lfloor\Omega_2\rfloor$ and $\Omega_2 \to^* \Omega_2'$ and $\Theta_2' = \lfloor\Omega_2'\rfloor$ and $\Omega_1 \mathrel{\mathring{=}} \Omega_2$ then $\Omega_1' \mathrel{\mathring{=}} \Omega_2'$.*

THEOREM 2 (Completeness). $P_1 \simeq P_2 \Rightarrow P_1 \simeq_{\mathrm{T}} P_2$.

Completeness is equivalently stated as: $P_1 \not\simeq_{\mathrm{T}} P_2 \Rightarrow P_1 \not\simeq P_2$. This is proven by devising an algorithm that takes as input two different traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ and the two programs P_1 and P_2 generating them and outputs a program P that interacts with P_1 and P_2 and is able to differentiate between them. The two different traces are generated as follows. Since $P_1 \not\simeq_{\mathrm{T}} P_2$, we have that $\mathsf{Tr}(P_1) \neq \mathsf{Tr}(P_2)$, thus there exists a trace $\overline{\alpha}$ that belongs to either only $\mathsf{Tr}(P_1)$ or only $\mathsf{Tr}(P_2)$. Assume wlog that $\overline{\alpha} \in \mathsf{Tr}(P_1)$. The trace $\overline{\alpha}$ can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha_d}$ such that $\overline{\alpha}$=$\overline{\alpha_s}\overline{\alpha_d}$, and so that there exists a trace $\overline{\alpha'} \in \mathsf{Tr}(P_2)$ that can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha_d'}$ such that $\overline{\alpha'}$=$\overline{\alpha_s}\overline{\alpha_d'}$ and $\overline{\alpha_d} \neq \overline{\alpha_d'}$. Trace $\overline{\alpha'}$ always exists, it could be an empty trace, it could be composed by an empty $\overline{\alpha_s}$ and, possibly, by an empty $\overline{\alpha_d'}$. The traces input for the algorithm are $\overline{\alpha_1}$ = $\overline{\alpha_s}\overline{\alpha_d}$ and $\overline{\alpha_2}$ = $\overline{\alpha_s}\overline{\alpha_d'}$.

Proof Sketch. This sketch shows how to generate P so that it interacts with P_1 or P_2 and tells them apart. Let div be a simple program written at some address X: `movi r0 X; jmp r0`. Differentiation between P_1 and P_2 is done by `halting` in one case and executing div in the other.

Input traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are sequences of actions where even-numbered ones are messages from P to either P_1 or P_2 and odd-numbered ones are messages from either P_1 or P_2 to P. By hypothesis there exists a different action in the traces, that action is at an odd-numbered index because it is generated by P_1 or P_2; P does not change so it generates the same actions. Thus there exists j for which $\overline{\alpha_1}(j) \neq \overline{\alpha_2}(j)$. Once executed, P replicates actions from the traces until the jth, then it performs the differentiation.

Even-numbered actions in $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are the same until the jth one. They are function calls or returns implemented in P. To replicate them, P sets flags and registers as in the action and then performs a `call` to the right address or a `return`. This code is placed at the current code location, which is either the initial address p_0 or the address indicated by a callback from P_1 or P_2.

Odd-numbered actions are carried out in P_1 or P_2. If these actions are the same, then no code is added to P. If they are different, the differentiating code stated above is placed at the current code location. If both actions are callbacks, the current code location is updated to the address mentioned in the callback. □

This general proof strategy is presented for both Theorem 1 and Theorem 2. The generalised approach is tailored to each semantics only in the relatively simple proof of Lemma 1. Since Theorem 1 and 2 hold for both Tr^{L} and Tr^{S}, both semantics are fully abstract w.r.t. the corresponding operational semantics.

5.1 Benefits of Fully Abstract Trace Semantics

Let us now highlight two of the benefits of fully abstract trace semantics.

Without the trace semantics, the capabilities of an attacker towards protected code are expressed as sequences of instructions, which can be difficult to reason about. With the trace semantics, the capabilities of that attacker are captured via the simple notion of traces. The full abstraction property ensures that traces express precisely all the capabilities of an attacker, without leaving any out.

The second benefit is in the field of secure compilation. Given two programs C_1 and C_2 in a language and indicate their compilation to an assembly language with $C_1^{\downarrow}$ and $C_2^{\downarrow}$ respectively. Proving the compilation scheme secure is formally stated as $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$ [1]. The more complex direction of this proof is $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq C_2^{\downarrow}$, but it can be simplified by adopting a fully abstract trace semantics for the assembly language. That proof is simpler in this way than by adopting the notion of contextual equivalence and performing induction on all possible low-level contexts. The complex direction becomes $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq_{\mathrm{T}} C_2^{\downarrow}$, whose contrapositive is $C_1^{\downarrow} \not\simeq_{\mathrm{T}} C_2^{\downarrow} \Rightarrow C_1 \not\simeq C_2$. This can be proven similarly to Theorem 2, as Agten *et al.* [2] and Patrignani *et al.* [15] did. Both implicitly use the second trace semantics where the compiler adds code that enforces the changes to the operational semantics.

6. Related Work

Full abstraction has been largely studied as a way to state the correctness of a denotational semantics with respect to an operational one [16]. This technique has been adopted for different programming languages paradigms, such as the λ-calculus [12] and the π-calculus [6]. For example, Ghica and Tzevelekos [5] provided a fully abstract the trace semantics, with regards to a game operational semantics, of a C-like language that, unlike this work, does not present a protection mechanism.

Fine grained, program counter-based memory access control mechanisms have been implemented in several software [10, 18–20] and hardware forms [4, 14] and recently by Intel in the SGX processor [11]. From the theoretical point of view, assembly languages extended with these protection mechanisms have been recently studied as target languages for secure compilation schemes [2, 15]. The language and trace semantics of this paper are inspired by those works. Besides elaborating on details of their formalisation, this work provides the first proof of full abstraction of the trace semantics.

A different research area studies logics for assembly languages: Hoare logics [17] or separation logics [9]. Jensen *et al.* [9] present a summary of the most recent advances in the latter. That research area focusses on providing reasoning facilities for assembly code, while this paper focusses on reasoning on the security of assembly code.

A different protection mechanism that could be employed at the assembly level is a typed assembly language [13]. To the best of the authors' knowledge, no fully abstract trace semantics has been provided for such languages.

7. Conclusion and Future Work

This paper studied the characterisation of the behaviour of isolated programs. To this extent, it formalised an assembly language extended with a fine grained, program counter-based memory access control mechanism. Then, it provided two different trace semantics for that language and a general proof strategy where both semantics are proven to be fully abstract. These semantics model the capabilities of attackers that inject malicious assembly code and they simplify proving secure compilation to the assembly language.

Providing a fully abstract trace semantics for a machine with multiple instances of an isolation mechanism or with multiple cores seem natural extensions to this work. The latter seems crucial in order to provide a secure compiler for concurrent programs to machines using the protection mechanism presented.

Acknowledgements. The authors would like to thank Tarmo Uustalu and the anonymous reviewers for useful feedback on an earlier draft.

References

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999.

[2] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *CSF '12*, pages 171 – 185. IEEE, 2012.

[3] Pierre-Louis Curien. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci.*, 172:301–310, April 2007.

[4] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS'12*, 2012.

[5] Dan R. Ghica and Nikos Tzevelekos. A system-level game semantics. *Electr. Notes Theor. Comput. Sci.*, 286:191–211, 2012.

[6] Alan Jeffrey and Julian Rathke. Full abstraction for polymorphic pi-calculus. In *FOSSACS'05*, pages 266–281. Springer-Verlag, 2005.

[7] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.

[8] Alan Jeffrey and Julian Rathke. Java Jr.: fully abstract trace semantics for a core java language. In *ESOP'05*, pages 423–438. Springer-Verlag, 2005.

[9] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. *SIGPLAN Not.*, 48(1):301–314, January 2013.

[10] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, 2008.

[11] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[12] Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[14] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the 22nd USENIX conference on Security symposium*. USENIX Association, 2013.

[15] Marco Patrignani, Dave Clarke, and Frank Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *(APLAS'13)*, volume 8301 of *LNCS*, pages 176–191, 2013.

[16] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[17] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Electr. Notes Theor. Comput. Sci.*, 156:151–168, 2006.

[18] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[19] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS 2012*, pages 2–13. ACM Press, October 2012.

[20] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm*, pages 344–361, 2010.

A. Algorithm description

Let us now give a description of the algorithm mentioned for Theorem 2.

We drop the differentiation via `div` and `halt` and introduce differentiation via different values (1 and 2 respectively) stored in $\texttt{r}_0$, followed by `halt`. The two techniques are equivalent, the second one leads to an more easily implementable algorithm.

Moreover we assume that there is always enough memory to write the algorithm.

In the output code P, there must be four functions in order to set the flags to the right combinations. These function are of the form:

- store $\texttt{r}_1$ and $\texttt{r}_2$ in memory;
- set $\texttt{r}_1$ and $\texttt{r}_2$ to the right values that set the flag in the wanted combination (e.g, for SF=0; ZF=1, set $\texttt{r}_1$=1 and $\texttt{r}_2$=1);
- execute `cmp` $\texttt{r}_1$ $\texttt{r}_2$;
- restore $\texttt{r}_1$ and $\texttt{r}_2$ to the corresponding previous values;
- `ret`.

The algorithm keeps track of where to write instructions in P with a stack: the *current address stack* c. Initially, the top of c is set to p_0.

The algorithm scans the traces $\overline{a_1}$ and $\overline{a_2}$. By construction, each even-numbered label is !-decorated; each odd-numbered label is ?-decorated. The algorithm is split in two subroutines based on what kind of actions it is examining. Each subroutine analyses one action from each trace and then calls the other subroutine on the following actions until the differentiation is achieved; in that case the algorithm terminates.

?-decorated actions. These actions are generated by the unprotected code. The algorithm must output P such that P generates those traces. Thus, at location c, the algorithm writes code depending on what action is being considered.

$(r; f)\texttt{call}\ p?$ Firstly, the algorithm writes a `call` to the function that sets the flags to f. Then the top of c is incremented by 1. Then, all twelve registers are set to the values of r, thus for all $i = 0..11$, the following instruction is written: $\texttt{movi}\ \texttt{r}_\texttt{i}\ r_i$. Eventually, if the value to be written in a register is larger than the constant allowed by `movi`, an `add` instruction is used. Then the top of c is incremented by 12. Then based on which register contains the value p that is where the call is directed, instruction $\texttt{call}\ \texttt{r}_\texttt{p}$ is written. Then the top of c is incremented by 1.

$(r; f)\texttt{ret}?$ Firstly, the algorithm writes a `call` to the function that sets the flags to f. Then the top of c is incremented by 1. Then, all twelve registers are set to the values of r, thus for all $i = 0..11$, the following instruction is written: $\texttt{movi}\ \texttt{r}_\texttt{i}\ r_i$. Then the top of c is incremented by 12. Then instruction `ret` is written. Then the top of c is incremented by 1.

!-decorated actions. These actions are generated by protected code. If the labels are the same, then no code is added to P.

callbacks. If both actions are of the form $(r; f)\texttt{call}\ p?$, then p is pushed on top of the stack c.

returns. If both actions are of the form $(r; f)\texttt{ret}?$, then the top of the stack c is popped.

writeouts. The algorithm adds no code to P.

If the labels are different, then the algorithm writes the differentiating code at address c in P. Differences in the labels can be of the following type:

different length. Thus one label is $\surd$ and the other one is $\alpha \neq \surd$. In this case, given that α is generated by program P_i, the algorithm writes $\texttt{movi}\ \texttt{r}_0\ i$ and `halt` at the address indicated by c.

different actions. Assume wlog that $\alpha_1 = (r; f)\texttt{ret}!$ and $\alpha_2 = (r; f)\texttt{call}\ 10!$. Then the algorithm writes instructions $\texttt{movi}\ \texttt{r}_0\ 1$; `halt` at c and instructions $\texttt{movi}\ \texttt{r}_0\ 2$; `halt` at address 10.

Assume wlog that $\alpha_1 = \texttt{write}(a, v).\delta!$ and $\alpha_2 = \delta!$. Then, *before the code written in P for δ!*, the following code is written to P: $\texttt{movi}\ \texttt{r}_0\ a$; $\texttt{movi}\ \texttt{r}_1\ u$; $\texttt{movs}\ \texttt{r}_0\ \texttt{r}_1$, where u is a different value from v. At address 20, the algorithm writes the instructions for the following pseudo-code: read the value of a, subtract v, if the result is 0, then $\texttt{movi}\ \texttt{r}_0\ 1$; `halt`, otherwise $\texttt{movi}\ \texttt{r}_0\ 2$; `halt`.

different values in the same action. Assume wlog that $\alpha_1 = (r; 0, 1)\texttt{ret}!$ and $\alpha_2 = (r; 0, 0)\texttt{ret}!$. Then the differentiating code is the following: perform a jump (via `jl` in this case) to an address a in case the flag is 1. At address a, instructions $\texttt{movi}\ \texttt{r}_0\ 1$; `halt` are written. Right after the jump, instructions $\texttt{movi}\ \texttt{r}_0\ 2$; `halt` are written.

Assume wlog that $\alpha_1 = (1, \ldots; f)\texttt{ret}!$ and $\alpha_2 = (2, \ldots; f)\texttt{ret}!$. Then the differentiating code is the following: $\texttt{movi}\ \texttt{r}_1\ 1$; $\texttt{sub}\ \texttt{r}_0\ \texttt{r}_1$. Now the problem is reduced to different values in flags, so the previous approach can be used.

Assume wlog that $\alpha_1 = (r; f)\texttt{call}\ 10!$ and $\alpha_2 = (r; f)\texttt{call}\ 20!$. Then the algorithm writes instructions $\texttt{movi}\ \texttt{r}_0\ 1$; `halt` at address 10 and instructions $\texttt{movi}\ \texttt{r}_0\ 2$; `halt` at address 20.

Assume wlog that $\alpha_1 = \texttt{write}(a_1, v_1).\delta!$ and $\alpha_2 = \texttt{write}(a_2, v_2).\delta!$. The same procedure stated in the last paragraph for the previous point is applied.

B. Proofs

We overload the hole-filling notation and allow a hole to be filled by a state $\Omega = (p, r, f, m, s)$ as follows: $\mathbb{M}[\Omega] = (p, r, f, m + m', s)$, if $(m, s) \frown \mathbb{M}$.

Proof. Proof of Theorem 1.

By Definition 4 the thesis $P_1 \simeq P_2$ becomes $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \iff \mathbb{M}[P_2]\Uparrow$.

The proof is split in two cases, one for each side of the co-implication.

1. $\Rightarrow$

   The thesis is $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \Rightarrow \mathbb{M}[P_2]\Uparrow$.
   As seen in Section 3.3 the thesis is $\forall \mathbb{M}.\ \mathbb{M}[\Omega_0(P_1)]\Uparrow \Rightarrow \mathbb{M}[\Omega_0(P_2)]\Uparrow$.
   Let $\Omega_1 = \mathbb{M}[\Omega_0(P_1)]$ and $\Omega_2 = \mathbb{M}[\Omega_0(P_2)]$.
   The thesis is $\forall \mathbb{M}.\ \forall n \in \mathbb{N}.\ \Omega_1 \to^n \Omega_1' \Rightarrow \forall m \in \mathbb{N}.\ \Omega_2 \to^m \Omega_2'$.
   The proof proceeds by induction on m.

   Base case: $m = 0$. Straightforward: $\Omega_2 \to^0 \Omega_2$.

   Inductive case: $m = h + 1$. The thesis is: $\Omega_2 \to^{h+1} \Omega_2'$. The inductive hypothesis (IH) is: $\forall \mathbb{M}.\ \forall n \in \mathbb{N}.\ \mathbb{M}[\Omega_0(P_1)] \to^n \Omega_1' \Rightarrow \mathbb{M}[\Omega_0(P_2)] \to^h \Omega_2^h$.
   By IH we have that: $\exists \Omega_1.\ \mathbb{M}[\Omega_0(P_1)] \to^h \Omega_1^h \to^{n-h} \Omega_1'$.
   Let $\Omega_1^h = (p_1, \ldots)$ and $\Omega_2^h = (p_2, \ldots)$.
   There are two cases based on p_1 and p_2 and on the last executed action.
   The last executed action is an action α such that: $\mathbb{M}[\lfloor \Omega_0(P_1) \rfloor] \xRightarrow{\overline{\alpha}\alpha} \lfloor \Omega_1^a \rfloor$ and $\exists k \in \mathbb{N}.\ \Omega_1^a \to^k \Omega_1^h$.

By hypothesis $P_1 \simeq_{\mathrm{T}} P_2$, $\mathbb{M}[\lfloor \Omega_0(P_2) \rfloor] \overset{\overline{\alpha}\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_2^a \rfloor$ and $\exists k \in \mathbb{N}.\ \Omega_2^a \rightarrow^k \Omega_2^h$.

(a) $s_1 \vdash \texttt{protected}(p_1)$ and $s_2 \vdash \texttt{protected}(p_2)$ and $\alpha = \gamma?$.

There are two cases: $\exists \alpha.\ \lfloor \Omega_1^h \rfloor \overset{\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$ or $\nexists \alpha.\ \lfloor \Omega_1^h \rfloor \overset{\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \Rightarrow \lfloor \Omega_1^{h'} \rfloor$.

i. $\exists \alpha.\ \lfloor \Omega_1^h \rfloor \overset{\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$.

By hypothesis $P_1 \simeq_{\mathrm{T}} P_2$, $\lfloor \Omega_2^h \rfloor \overset{\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_2^{h'} \rfloor$.
This, in conjunction with IH, implies the thesis $\Omega_2 \rightarrow^{h+1} \Omega_2'$.

ii. $\nexists \alpha.\ \lfloor \Omega_1^h \rfloor \overset{\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$.

There are two cases: either $\lfloor \Omega_1^h \rfloor \overset{\surd}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$ or $\lfloor \Omega_1^h \rfloor \overset{\surd}{\not\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$.

A. $\lfloor \Omega_1^h \rfloor \overset{\surd}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$.
This case cannot happen as it contradicts the assumption $\forall n \in \mathbb{N}.\ \mathbb{M}[\Omega_0(P_1)] \rightarrow^n \Omega_1'$.

B. $\lfloor \Omega_1^h \rfloor \overset{\surd}{\not\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^{h'} \rfloor$.
Let $\Omega_1^h = (p_1, r_1, f_1, m + m_1, s_1)$ and $\Omega_2^h = (p_2, r_2, f_2, m + m_2, s_2)$.

By hypothesis $P_1 \simeq_{\mathrm{T}} P_2$, $\lfloor \Omega_2^h \rfloor \overset{\surd}{\not\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_2^{h'} \rfloor$.
By inspecting rules for generating τ in traces (the only possible rule that applies in this case), we have that $m_1(p_1) = i_1 \in \mathcal{I}$ and $m_2(p_2) = i_2 \in \mathcal{I}$.

The thesis holds because $\Omega_2 \rightarrow^h \Omega_2^h \xrightarrow{i_2} \Omega_2'$.

(b) $s_1 \vdash \texttt{unprotected}(p_1)$ and $s_2 \vdash \texttt{unprotected}(p_2)$ and $\alpha = \gamma!$ or there was no α.

By IH $\exists l \leq h.\ \mathbb{M}[\Omega_0(P_1)] \rightarrow^l \Omega_1^l$ and $\lfloor \Omega_0(P_1) \rfloor \overset{\overline{\alpha}\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_1^l \rfloor$.

By hypothesis $P_1 \simeq_{\mathrm{T}} P_2$, $\exists l \leq h.\ \mathbb{M}[\Omega_0(P_2)] \rightarrow^l \Omega_2^l$.

Additionally, $\lfloor \Omega_0(P_2) \rfloor \overset{\overline{\alpha}\alpha}{\Longrightarrow\!\!\!\!\!\rightarrow} \lfloor \Omega_2^l \rfloor$.

By Lemma 1, $\Omega_1^l = (p^l, r^l, f^l, \mathbb{M}^l + m_1, s_1)$ and $\Omega_2^l = (p^l, r^l, f^l, \mathbb{M}^l + m_2, s_2)$.

Additionally, $\Omega_1^l \rightarrow^{h-l} \Omega_1^h$ and $\Omega_2^l \rightarrow^{h-l} \Omega_2^h$.

Since $\mathbb{M}^l$ is the same for both P_1 and P_2, the $(h-l)$-steps they perform in unprotected memory are the same for Ω_1^l and Ω_2^l.

Thus $\Omega_1^h = (p^h, r^h, f^h, \mathbb{M}^h + m_1, s_1)$ and $\Omega_2^h = (p^h, r^h, f^h, \mathbb{M}^h + m_2, s_2)$.

As stated in Section 3.3 $p \in \texttt{dom}(\mathbb{M}^h)$ implies that $s_1 \vdash \texttt{unprotected}(p^h)$ and $s_2 \vdash \texttt{unprotected}(p^h)$.

By hypothesis $\forall n \in \mathbb{N}.\ \mathbb{M}[\Omega_0(P_1)] \rightarrow^n \Omega_1'$: we have that $\Omega_1^h \xrightarrow{i} \Omega_1'$ and that $\mathbb{M}^h(p^h) \cong i \in \mathcal{I}$.

This implies the thesis: $\Omega_2 \rightarrow^{h+1} \Omega_2'$ since $\Omega_2 \rightarrow^h \Omega_2^h \xrightarrow{i} \Omega_2'$.

2. $\Leftarrow$ As in case 1, *mutatis mutandis*.

□

Proof. Proof of Lemma 1 for Tr^{L} (Implies the proof the same lemma for Tr^{S}).

Straightforward induction on $\overline{\alpha}$ that leads to a case analysis on α.

$\surd$. Straightforward: the thesis is $\Omega_1 \stackrel{\circ}{=} \Omega_2$, which is among the hypotheses.

?-decorated actions. These actions are done in the same unprotected memory. Moreover they are generated by the same instructions (even if they can be at different addresses): `call` for a label of the form $(r, f)\texttt{call}\ p?$ and `ret` for a label of the form $(r, f)\texttt{ret}?$. Neither of those instructions touch the memory, and they only modify registers and flags in the same way.

Thus, also in this case, the thesis $\Omega_1' \stackrel{\circ}{=} \Omega_2'$ holds.

!-decorated actions. These are the subcases for this one, one for each kind of label that can be !-decorated.

$(r; f)\texttt{call}\ p!$ This label is generated by a `call` instruction which, in a single step, does not change registers (besides SP) nor flags, nor unprotected memory. As the value of SP is captured in r, and r is the same for both Ω_1 and Ω_2, the interface does not change. Thus, the thesis $\Omega_1' \stackrel{\circ}{=} \Omega_2'$ holds.

$(r; f)\texttt{ret}\ p!$ This label is generated by a `ret` instruction which, in a single step, does not change registers (besides SP) nor flags, nor unprotected memory. The same considerations made for SP before hold here, thus the thesis $\Omega_1' \stackrel{\circ}{=} \Omega_2'$ holds.

Additionally those labels can be preceded by an arbitrary number of $\texttt{write}(a, v)!$. This prefix is generated by a `movs` instruction which, in a single step, does not change registers nor flags. Address a is unprotected, so the unprotected memory is changed but in the same way. Thus, these prefixes do not affect the thesis, which holds.

τ actions Unobservable actions can be generated in protected memory: τ_i.

Changes to registers and flags made by those actions are captured in the following, observable action α. Since α is the same for both Θ_1 and Θ_2, registers and flag are changed in the same way.

Changes to unprotected memory that are done by the protected code are captured by α, thus the protected memory is changed in the same way.

The proof of Lemma 1 for Tr^{S} is a subset of the proof for Tr^{L} since the labels of Tr^{S} are a subset of the labels of Tr^{L}.

□