Fully Abstract Trace Semantics for Protected Module Architectures

Marco Patrignani[a,1], Dave Clarke[b,a]

[a] iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium
[b] Dept. of Information Technology, Uppsala University, Sweden

Abstract

Protected module architectures (PMA) are an isolation mechanism that emerging processors provide as security building blocks for modern software systems. 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 well-structured semantics. This paper presents one such semantics, namely a *fully abstract* trace semantics, for an assembly language enhanced with PMA. 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 code and simplifies the formulation of a secure compiler targeting PMA-enhanced assembly language.

Keywords: Fully abstract semantics, trace semantics, untyped assembly language, protected modules architectures, formal languages

1. Introduction

Emerging processors, such as the Intel SGX [1], provide isolation mechanisms as software security building blocks. These are used to withstand low-level attackers who, generally through injected assembly code, can read the whole memory space and can thus access secrets in memory, violate integrity constraints and so on. When these isolation mechanisms are in place, attackers cannot directly violate security properties of isolated software since the isolated memory is not accessible to them. Examples of these protection mechanisms are protected module architectures (PMA) [1, 2, 3, 4, 5, 6, 7], which enforce security properties at process or lower levels (Ring -1). With PMA, the software to be secured is placed in a protected memory partition (a protected module) that

Email addresses: `first.last@cs.kuleuven.be` (Marco Patrignani), `first.last@it.uu.se` (Dave Clarke)
[1]Corresponding author.

shields it from the surrounding, potentially malicious code. The malicious code can neither 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. Thus, PMA makes software more resilient against low-level attackers. However, this does not prevent an attacker from violating security properties of protected code by interacting with it.

Describing the interaction between protected and unprotected code or (dually) of an attacker to protected code can be done by using contextual equivalence. However, while being precise, contextual equivalence is notoriously difficult to reason about [8]. An alternative characterisation of the behaviour of protected code has the form of *fully abstract* trace semantics. Such a semantics uses simple abstractions to represent the behaviour of protected assembly code, unburdened by low-level details, while remaining at the maximum degree of precision. Dually, it models the behaviour of attackers to protected code, since it captures precisely the capabilities of those attackers.

The fully abstract trace semantics has the following benefits. Firstly, it allows contextual equivalence to be disregarded, since contextual and trace equivalence are proven to be equally precise. The full abstraction property ensures that traces express precisely all the capabilities of an attacker. Without the trace semantics, the capabilities of an attacker towards protected code are expressed by means of contexts: complex sequences of assembly instructions. With the trace semantics, the capabilities of that attacker are captured via the simple notion of traces, which provide a clearer abstraction than contexts.

Secondly, the fully abstract trace semantics fulfils the claims of recent secure compilation works targeting PMA-enhanced assembly languages. Given two programs C_1 and C_2 written in a language L, indicate their compilation to an assembly language with $C_1^{\downarrow}$ and $C_2^{\downarrow}$ respectively. One way of proving the compilation scheme secure is formally stated as $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$ [9]. 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, as in the works of Agten *et al.* [10] and Patrignani *et al.* [11, 12]. These works presented secure compilers to PMA-enhanced assembly code that depend on the assembly language having a fully abstract trace semantics such as one of those presented in this paper.

Finally, the trace semantics allows some limitations of the aforementioned secure compilers to be forgone. Currently, securely-compiled function calls can have a number of parameters based on what the registers allow. To overcome this limitation (or to pass large data that does not fit in a register value), additional parameters can be spilled on the stack in unprotected memory. To allow this spilling, the trace semantics needs to capture reading and writing outside of the protected memory. While none of the previous did, the trace semantics of this paper considers both operations.

This paper initially presents the PMA protection mechanism and informally describes how to devise a fully abstract trace semantics for PMA-enhanced assembly code (Section 2). Then it introduces $\mathcal{A}$+I: an assembly language en-

hanced with PMA (Section 3). This paper then investigates how different operations across PMA boundaries are supported by trace semantics. It explores the design space of trace semantics for $\mathcal{A}$+I and presents two different fully abstract trace semantics for it (Section 4): one where cross-boundary operations are restricted to function calls (Section 4.1) and one where they are unrestricted (Section 4.2). This paper extends the authors' previous work [13] by considering additional behaviour in traces in the form of protected code reading unprotected memory (whose complications are explained in Section 2.2). This paper then provides a general strategy to simplify the proof of full abstraction of the trace semantics (Section 5). Finally, it reviews related work (Section 6) and concludes (Section 7). Limitations of this work are threefold. 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 commonly found assumption [10, 11, 14, 15]. Thirdly, the second trace semantics relies on an assumption on the partitioning of unprotected code that is not readily fulfilled by certain PMA implementations; Section 7 discusses this limitation.

2. Protected Programs and Trace Semantics

This section describes the PMA memory access control mechanism and the behaviour of $\mathcal{A}$+I code (Section 2.1). Then it discusses the pitfalls to avoid in order to obtain a fully abstract trace semantics for $\mathcal{A}$+I code (Section 2.2).

2.1. The PMA Protection Mechanism, Informally

The assembly language is enhanced with a protected module architecture (PMA). This isolation mechanism enforces a fine-grained, program counter-based memory access control mechanism [2, 3, 4, 5, 6, 7]. We review this mechanism from the work of Strackx and Piessens [6], upon which our results are based. The techniques presented in this paper can nevertheless be easily adapted to reasoning about other isolation mechanisms [2, 3, 5]. 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 read-only *code* and a non-executable *data* section. The code section contains a variable number of *entry points*: the only addresses which instructions in unprotected memory can jump to and execute. The data section is accessible only from the protected partition. Based on the location of the program counter, instructions that violate the access control policy cause the execution to `halt` [10, 11].

The table below summarises the access control model enforced by PMA.

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

Following are some code snippets that exemplify how the PMA access control mechanism works, and, introduce the syntax of the $\mathcal{A}$+I along the way. All $\mathcal{A}$+I examples throughout the paper assume the presence of a protected memory section spanning from address 100 to 200, with a *single* entry point at address 100. In the examples, call the code located in the protected section P_s and the code located in the unprotected section P_u. Every instruction is preceded by the address where it is located; execution starts at address 0.

Example 1 (No execution of code in the protected code section). *P_u initialises register* $\mathtt{r}_0$ *to 101 (line 1) and then jumps to that address (line 2).*

```
1 | 0    movi r0 101  // unprotected code
2 | 1    jmp r0
3 | ...
4 | 100  add r0 r1    // protected code
5 | 101  ret
```

Since address 101 is not an entry point of the protected memory section, the jump of P_u does not succeed.

Example 2 (No reading/writing the protected code section). *P_u initialises register* $\mathtt{r}_0$ *to 101 (line 1) and register* $\mathtt{r}_1$ *to 20 (line 2), then it writes the contents of* $\mathtt{r}_1$ *to the address in* $\mathtt{r}_0$ *(line 3).*

```
1 | 0    movi r0 101  // unprotected code
2 | 1    movi r1 20
3 | 2    movs r0 r1
4 | ...
5 | 100  add r0 r1    // protected code
6 | 101  ret
```

Since address 101 is protected, P_u cannot write there, so execution is halted, as in Example 1. Analogously, if the instruction of line 2 were replaced with `movl` $\mathtt{r}_0$ $\mathtt{r}_1$*, the execution halts. In that case, P_u would be attempting to read the protected memory section, while it does not have that privilege.*

Example 3 (Interoperation between protected and unprotected code). *P_u initialises register* $\mathtt{r}_0$ *to 12 (line 1), register* $\mathtt{r}_1$ *to 10 (line 2), register* $\mathtt{r}_5$ *to 100 (line 3) and then calls to the protected function located at address 100 (line 4), storing address* 4 *on the call stack (implicit). P_s subtracts registers* $\mathtt{r}_0$ *and* $\mathtt{r}_1$ *(line 6) and, if the result is greater than or equal to zero, returns that result (line 9). Otherwise, if the result is less than zero, P_s jumps to address 104 (lines 7, 8), and returns 0 (lines 10, 11). Execution then continues in unprotected memory at address 4 (line 5, omitted), which is the address popped from the call stack (implicit).*

```
1 | 0    movi r0 12   // unprotected code
2 | 1    movi r1 10
3 | 2    movi r5 100
4 | 3    call r5
5 | ...
6 | 100  sub r0 r1    // protected code
```

```
 7 101  movi r3 104
 8 102  jl r3
 9 103  ret
10 104  movi r0 0
11 105  ret
```

To provide a better understanding of the PMA memory layout, Figure 1 below provides a graphical representation of the layout of this example.

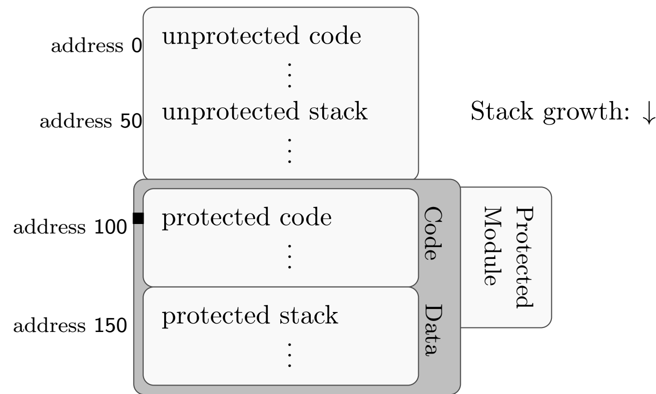

Figure 1: Memory layout for the code of Example 3.

2.2. From Naïve to Fully Abstract Trace Semantics

As seen in Example 3, the description of the behaviour of protected $\mathcal{A}$+I code can be rather burdensome as it is expressed in terms of the external code and each protected instruction. A trace semantics can give a simpler description of the behaviour of protected $\mathcal{A}$+I code in terms of a set of *sequences of labels*. These labels capture how communication between protected and unprotected code happens and what is communicated. In this paper, trace semantics are devised to capture the behaviour of a *protected* program, which is a program allocated in the protected memory partition. Example 4 presents a trace-based description of the behaviour of the protected code of Example 3. After showing the limitations of the initial trace semantics, this section presents the pitfalls that arise when considering writes (Section 2.2.1) and reads (Section 2.2.2) to unprotected memory.

Example 4 (Describing behaviour with traces). *Consider only the protected code of the snippet from Example 3.*

```
1 100  sub r0 r1   // protected code
2 101  movi r3 104
3 102  jl r3
4 103  ret
5 104  movi r0 0
6 105  ret
```

Since there is a single entry point to this code, located at address 100, this code just represents a single function. A possible behaviour of this code can be expressed as follows (· is used to separate actions of the same trace):

$$\texttt{call}\ 100\ \texttt{r}_0, \ldots, \texttt{r}_{11} \cdot \texttt{ret}\ \texttt{r}_0$$

To describe the behaviour of the code of Example 4 as a trace, we identify the actions that are observable from the point of view of code interacting with the snippet above: `call` and `ret`. These actions are the labels of the trace semantics; they are generated by `call` and `ret` instructions. Not all instructions generate a visible label in a trace, only those whose effect can be observed from the unprotected code.

Following is the syntax of labels of a trace semantics for protected $\mathcal{A}$+I code.

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

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 in a call and v indicates the contents of register $\texttt{r}_0$ in a return. Calls and returns executed by unprotected code are named *calls* and *returnbacks*, dually, if they are executed by protected code they are named *callbacks* and *returns* [10, 16].

This paper aims at providing a fully abstract trace semantics, thus implying that the trace semantics is the most precise. Informally, a trace semantics is fully abstract when its labels capture all that is being communicated between the protected and the unprotected code but no more. A trace semantics following the discussion above would not be fully abstract due to a number of subtleties, as highlighted in Example 5.

Example 5 (Limitation of the aforementioned trace semantics). *Consider the two protected $\mathcal{A}$+I programs below, call the left one P_L and the right one P_R. When presenting snippets side by side, differences are highlighted in a blue font.*

```
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
```

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

With the trace semantics hinted at after Example 4, the behaviours of P_L and P_R coincide, as they generate the same traces. However, P_L and P_R can be distinguished by an external observer, and the traces they generate should reflect this. Consider trace $\overline{a_1}$, which is generated by both P_L and P_R (omitted details are indicated using ...).

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

$\overline{a_1}$ does not capture the different values contained in $\mathtt{r}_{11}$ (line 7), which, even if they are not the returned values of the function, still constitutes an observable difference between P_L and P_R.

Trace $\overline{a_2}$ is also generated by both P_L and P_R.

$$\overline{a_2} = \mathit{call}\ 100(2, 1, 40, \ldots)? \cdot \mathit{call}\ 40(\ldots)!$$

$\overline{a_2}$ 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.

Let us now consider writing and reading to unprotected memory.

2.2.1. Writeouts

Protected code writing a value into the unprotected memory partition is called a *writeout*. Since such values can be observed by unprotected code, writeouts need to be captured in traces. This is done with a writeout label of the following form: $\mathtt{write}(a, v)$ stating what was written (v) and where (a). Following are the subtleties that need to be considered when introducing writeouts into the trace semantics (Examples 6 to 9). In the first case the problem is that the write is not observable, while in the second case the problem is the ordering of writout labels. In the remaining cases the problem is that control is not returned to the external code, which means that it will not be able to detect the difference introduced by the writeout.

Example 6 (Invisible writeouts). *The following P_L and P_R read a value from an unprotected address 10 and 20, respectively (line 2), and then rewrite the same value back to the same address (line 3).*

```
1 100 movi r0 10
2 101 movl r1 r0
3 102 movs r0 r1
4 103 movi r0 0
5 104 ret
```

```
1 100 movi r0 20
2 101 movl r1 r0
3 102 movs r0 r1
4 103 movi r0 0
5 104 ret
```

The writeouts of P_L and P_R are invisible. *In fact, they do not alter the contents of unprotected memory, since address 10 (20, resp.) already contains the written value. Thus, P_L and P_R are contextually equivalent. However, they are not trace equivalent, since the following is a trace of P_L and not of P_R:*

$$\mathtt{call}\ 100(\cdots)? \cdot \mathtt{read}(10, 0)\mathtt{write}(10, 0)\mathtt{ret}\ 0!$$

Notice that if the readout were absent, the writeout would distinguish between P_L and P_R, as there are unprotected memories whose existing value at address 10 (20, resp.) differs from what is written by P_L or P_R.

To address this concern, the readout information must be accumulated and used to detect when a writeout is not introducing an observable difference in unprotected memory.

Example 7 (Order independence of writeouts). *The following P_L and P_R write 0 to addresses 10 and 20 in unprotected memory (lines 4 and 5). The only difference between the two is that P_L writes to address 10 then to address 20 while P_R does the same writes in the opposite order.*

```
1 100 movi r1 10
2 101 movi r2 20
3 102 movi r0 0
4 103 movs r1 r0
5 104 movs r2 r0
6 105 ret
```

```
1 100 movi r1 10
2 101 movi r2 20
3 102 movi r0 0
4 103 movs r2 r0
5 104 movs r1 r0
6 105 ret
```

These programs are contextually equivalent, but if their labels are generated by the orders of the instructions, they will have different labels, since the following will be a trace of P_L and not of P_R.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10,0)\texttt{write}(20,0)\texttt{ret}\ 0!$$

To address this concern, writeouts need to be sorted when they are added to a trace. A more precise discussion over this solution is delayed until Example 15 since the solution is affected by the solutions of other

Example 8 (No writeouts with termination). *The following P_L and P_R write 0 and 1 respectively to address 10 in unprotected memory (line 3) and then terminate (line 4).*

```
1 100 movi r1 10
2 101 movi r0 0
3 102 movs r1 r0
4 103 halt
```

```
1 100 movi r1 10
2 101 movi r0 1
3 102 movs r1 r0
4 103 halt
```

The only difference between P_L and P_R is the value written at address 10. However, the unprotected code cannot detect this difference since execution is halted before control is returned to it. Thus, P_L and P_R are contextually equivalent. If the writeout would appear in the traces, P_L and P_R would be trace-inequivalent, since the trace below would belong to P_L and not to P_R.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10,0)\surd$$

Consequently, writeouts do not appear if the protected program halts afterwards.

Example 9 (Writeouts are not executable). *The following P_L and P_R set $\texttt{r}_0$ to 20 and 10 respectively (line 1), then write the instruction $\texttt{jmp r}_0$ at addresses 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 $\mathtt{r_0}$ *is set to 20 (resp. 10), the instruction* `jmp` $\mathtt{r_0}$ *written at address 20 (resp. 10) will diverge when called. 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*, since a trace of* P_R *would contain a* $\mathtt{write}(10, \text{“}\mathtt{jmp\ r_0}\text{”})\mathtt{call}\ 10\ (10, \cdots)!$ *label.*

$$\mathtt{call}\ 100\ (\cdots)? \cdot \mathtt{write}(20, \text{“}\mathtt{jmp\ r_0}\text{”})\mathtt{call}\ 20\ (20, \cdots)!$$

The solution to this concern is to split the unprotected memory in a code and a data section and to allow writeouts only to the unprotected data section. A more complete analysis of the solution is delayed until Example 16.

2.2.2. Readouts

A *readout* occurs when protected code reads unprotected memory. Not all PMA implementations allow readouts, they are forbidden in some implementations [3] and discouraged by others [4, 6]. When protected code can perform readouts, devising a fully abstract trace semantics is challenging. The readout label $\mathtt{read}(a, v)$ states that a value v was read from address a. It is not obvious to decide when such a label should appear and the following examples present when the readout label should appear in traces or not (Examples 10 to 16).

Example 10 (Unobservable readouts). *Consider the two protected* $\mathcal{A}$+I *programs below.*

```
1 100 movi r0 10
2 101 movl r1 r0
3 102 movi r1 0
4 103 movi r0 0
5 104 ret
```

```
1 100 movi r0 20
2 101 movl r1 r0
3 102 movi r1 0
4 103 movi r0 0
5 104 ret
```

P_L *and* P_R *read the contents of unprotected addresses 10 and 20, respectively, and store the result in register* $\mathtt{r_1}$ *(line 2), then they set registers* $\mathtt{r_0}$ *and* $\mathtt{r_1}$ *to 0 (lines 3,4) and return (line 5). In this case, the value read does not influence the behaviour of* P_L *or* P_R*, which behave the same, so the readout should not appear in their traces.*

Example 11 (Readouts reduce to a constant). *Consider the two protected* $\mathcal{A}$+I *programs below:*

```
1 100 movi r1 10
2 101 movl r0 r1
3 ···      // manipulate r0
4          // until it contains k
5 102 ret
```

```
1 100 movi r1 10
2 101 movi r0 k
3 102 ret
```

Here, P_L *reads the contents of address 10 into* $\mathtt{r_0}$ *(line 2), performs computations until* $\mathtt{r_0}$ *contains a constant value* k *(omitted lines), independent of the value*

read, and then returns (line 5). P_R simply initialises $\mathtt{r}_0$ to k (line 2) and returns (line 3).

These programs are contextually equivalent, both always return k, however, P_L also performs a readout. If this readout appears in traces, it would be a failure of full abstraction, since the traces of P_R do not have such a label. The problem here is that the omitted code of P_L always reduces the contents of $\mathtt{r}_0$ to a constant, no matter what values it contained beforehand. The trace semantics must be able to identify that the value read do not affect the execution of the program and thus not include the read label in this case.

Example 12 (Observable readouts). *Consider the two protected $\mathcal{A}$+I programs below.*

```
1 100  movi r0 10
2 101  movl r1 r0
3 102  movi r0 0
4 103  sub r0 r1
5 104  movi r0 108
6 105  je r0
7 106  movi r0 30
8 107  call r0
9 108  ret
```

```
1 100  movi r0 20
2 101  movl r1 r0
3 102  movi r0 0
4 103  sub r0 r1
5 104  movi r0 108
6 105  je r0
7 106  movi r0 30
8 107  call r0
9 108  ret
```

In this case P_L and P_R read the contents of unprotected addresses 10 and 20, respectively, in register $\mathtt{r}_1$ (line 2). Then, if those values are less than 0 (lines 3, 4) they jump to address 108 (lines 5, 6) and return (line 9), otherwise they call to a function at address 30 (lines 7, 8).

The value read in unprotected memory constitutes an observable difference between P_L and P_R, as it alters the execution flow. Thus, the readout value should itself be present in the trace.

The problem in this case is detecting when does a read affect tha behaviour of a program. A read affects the behaviour of a program if some future behaviour of the program depends on the value read; when different values are read, the behaviour of the programs varies. On the other hand, if a read does not affect the behaviour, any value can be read and the program behaves the same. By viewing readout values as inputs, in the former case we can say that different inputs make a program have different behaviours (as in Example 12, while in the latter case different inputs do not vary the behaviour of a program (as in Examples 10 to 11).

The concept described above is analogous to *non-interference* [17, 18]. Non-interference is a property of systems whose input can be classified to be either low or high security (for non sensible and classified material respectively). A system is non-interfering if for a given set of low inputs it will produce the same low outputs, regardless of what the high level inputs are.

In this setting, if we treat readouts as high inputs and future traces as low outputs, we can apply non-interference notions to detect whether a readout affects a program. A readout does not affect a program if it is non-interfering, i.e. for any readout value (high input) the future traces (low output) do not

vary. The trace semantics can use the non-interference information to decide whether a readout label should appear on traces or not. In Examples 10 to 11, the readouts are non-interfering, whatever value is read, the behaviour of the program does not vary, thus the trace semantics can exclude these readouts from traces. However, in Example 12 if the value read is 0, the program will behave differently than if it is not 0, so the readout is interfering. Here the trace semantics can tell that the readout must be included in the trace.

The main difference between the way non-interference is used in the literature and in this work is in the treatment of readout values. These values are in the external memory, thus intuitively low security, and they should be kept immutable. However, in order to apply non-interference correctly, they have to vary, thus they are regarded as high security.

Example 13 (Unobservable readouts after writeout). *The following P_L and P_R write 0 to address 10 (line 3), then P_R reads from address 10 (line 4).*

```
1 100 movi r1 10
2 101 movi r0 0
3 102 movs r1 r0
4
5 103 ret
```

```
1 100 movi r1 10
2 101 movi r0 0
3 102 movs r1 r0
4 103 movl r0 r1
5 104 ret
```

These programs are thus contextually equivalent, but the following is a trace of P_R and not of P_L.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10, 0)\texttt{read}(10, 0)\texttt{ret}\ 0!$$

To address this concern, reads to an address that was the subject of a writeout should not appear on traces. In fact, the readout value cannot be different from the writeout one, and that information is already known to protected programs.

Example 14 (Multiple readouts). *The following P_L and P_R read from address 10 in unprotected memory (line 2).*

```
1 100 movi r1 10
2 101 movl r1 r0
3
4 102 ret
```

```
1 100 movi r1 10
2 101 movl r1 r0
3 102 movl r1 r0
4 103 ret
```

The only difference between the two is that P_R reads from address 10 twice, but this does not affect its behaviour, since the same value is read. Thus, these programs are contextually equivalent, but the following is a trace of P_L and not of P_R.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{ret}\ 0!$$

To address this concern, multiple readouts to the same address should thus be filtered, only one must be present in the traces.

Example 15 (Order independence of readouts). *In the following, P_L reads the contents of unprotected address 10 into register $\texttt{r}_1$ (line 2), then it reads the contents of unprotected address 20 into register $\texttt{r}_2$ (line 4). Finally, it calls to*

a function located at address 20 (line 5, the value of register $\texttt{r}_0$*).* P_R *does the same, but first its reads happen in the reversed order: first it reads address 20 into register* $\texttt{r}_2$ *(line 2), then address 10 into register* $\texttt{r}_1$ *(line 2).*

```
1 100  movi r3 10
2 101  movl r1 r3
3 102  movi r0 20
4 103  movl r2 r0
5 104  call r0
```

```
1 100  movi r0 20
2 101  movl r2 r0
3 102  movi r3 10
4 103  movl r1 r3
5 104  call r0
```

These programs are contextually equivalent, but the traces they create are different. The order in which the readouts are executed and accumulated on the traces makes it so that the following trace is generated by P_L *and not by* P_R*.*

$$\texttt{call}\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{read}(20, v')\texttt{call}\ 20\ (20, v, v', 10, \cdots)!$$

To address this and the concern of Example 7, readouts and writeouts can be sorted based on the address to which the operation is performed.

This introduces a sort of normal form for traces, which consist of a sorted prefix of readouts and writeouts followed by a call or a return. The normal form effectively merges the solutions to Examples 13 to 14.

Example 16 (Readouts are not executable). *In the following,* P_L *always halts while* P_R *reads the contents of address 10 into* $\texttt{r}_1$ *(lines 1, 2). If the value read is not an instruction (line 3, omitted for the sake of simplicity),* P_R *jumps there (line 4), otherwise it halts (line 6).*

```
1 100  halt
2
3
4
5
6
7
8
9
10
```

```
1 100  movi r3 107
2 101  movi r0 10
3 102  movl r1 r0
4 // load the encoding for halt in r2
5 103  movi r2  "halt"
6 104  cmp r1 r2
7 // if address 10 contains halt
8 105  je r3
9 106  call r1 // jump to address 10
10 107  halt // otherwise, halt
```

These two programs are contextually equivalent: they always terminate. However, P_R *generates the following trace, which* P_L *does not:*

$$\texttt{call}\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{call}\ 10\ (10, v, \cdots)!$$

The problem is that the trace above will always be followed by termination (in unprotected code), which unprotected code cannot observe. This is due to P_R *reading executable unprotected code and* P_R *behaving differently based on the value read.*

To address this concern and Example 9, unprotected code is split in a code and a data section, just as protected code is. Writeouts and readouts can only be performed on the data section of unprotected code, so protected code cannot read nor write executable unprotected code.

From the threat modeling perspective, this assumption somewhat reduces the attacker's power, since she is not able to execute the values written by the protected code. However, this assumption seems reasonable, since most times we are interested in modelling the behaviour of code that uses readouts for parameters and not to execute readout values. Future work will consider writeouts and readouts of executable unprotected code.

As Curien stated [19], two ways to achieve full abstraction for a trace semantics exist. The first 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 readouts and writeouts. The second 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 flags as well as what protected code reads and writes in unprotected memory. Both approaches are presented in Section 4. These are based on the $\mathcal{A}$+I assembly language and its operational semantics, which are given in Section 3.

3. $\mathcal{A}$+I Assembly Language Formalisation

This section formalises the syntax (Section 3.1) and operational semantics (Section 3.2) of $\mathcal{A}$+I, a PMA-enhanced assembly language, and concludes with the definition of contextual equivalence for $\mathcal{A}$+I programs (Section 3.3).

3.1. Syntax

$\mathcal{A}$+I programs 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 next. The register file contains 12 general purpose registers $\mathtt{r}_0$ to $\mathtt{r}_{11}$ 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, respectively.

$$
\begin{array}{rlrl}
\textit{Words} & w ::= n \in \mathbb{N} \cup -1 & \textit{Memories} & m ::= \emptyset \\
\textit{Instructions} & i \in \mathcal{I} \subset \textit{Words} & & \mid m; a \mapsto w \\
\textit{Addresses} & a \in n \in \mathbb{N} & \textit{Programs} & P ::= (m, s) \\
\textit{Memory descriptors} & s ::= (a_b, n_c, n_d, n, a_{uc}, a_{ud}) & &
\end{array}
$$

Figure 2: Elements of the $\mathcal{A}$+I formalisation.

Figure 2 presents elements of the formalisation. Addresses a are natural numbers. Words are natural numbers plus -1, which serves as a value that is not in the address range in order to stop computation (As described in Definition 2

below). Memories m are infinite 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 $\texttt{dom}(m) = \{a \mid a \mapsto w \in m\}$. If two memories m and m' have disjoint domains, they can be merged into another memory. Formally, if $\texttt{dom}(m) \cap \texttt{dom}(m') = \emptyset$, then $m + m' = \{a \mapsto w \mid a \mapsto w \in m \text{ or } a \mapsto w \in m'\}$. Memory descriptors s are sextuples: $(a_b, n_c, n_d, n, a_{uc}, a_{ud})$ 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. Additionally, a_{uc} states where the code section of the unprotected code starts and a_{ud} sttes where the data section of the unprotected code starts (and where the unprotected code section ends). This partitioning of unprotected code is not required by PMA architectures but it helps devising a fully abstract trace semantics, as explained previously. Entry points are allocated starting from the base address a_b. Each entry point is $\mathcal{N}_e$ words long. Assume that 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. Instructions i are elements of the set $\mathcal{I}$ and define the programming language executed on the architecture (Figure 3).

Instruction	Description
$\texttt{movl r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Load the word from the address in register $\texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$.
$\texttt{movs r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Store the contents of register $\texttt{r}_\texttt{s}$ at the address found in register $\texttt{r}_\texttt{d}$.
$\texttt{movi r}_\texttt{d}\ \texttt{k}$	Load the constant value $\texttt{k}$ into register $\texttt{r}_\texttt{d}$.
$\texttt{add r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} + \texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$ and set the ZF flag accordingly.
$\texttt{sub r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} - \texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$ and set both the ZF and the SF flags accordingly.
$\texttt{cmp r}_\texttt{s}\ \texttt{r}_\texttt{d}$	Calculate $\texttt{r}_\texttt{s} - \texttt{r}_\texttt{d}$ and set both the ZF and the SF flags accordingly.
$\texttt{jmp r}_\texttt{i}$	Jump to the address located in register $\texttt{r}_\texttt{i}$.
$\texttt{je r}_\texttt{i}$	If the ZF flag is set, jump to the address in register $\texttt{r}_\texttt{i}$.
$\texttt{jl r}_\texttt{i}$	If the SF flag is set, jump to the address in register $\texttt{r}_\texttt{i}$.
$\texttt{call r}_\texttt{i}$	Push the value of the program counter $+1$ onto the stack and jump to the address in register $\texttt{r}_\texttt{i}$.
ret	Pop a value from the stack and jump to the popped location.
halt	Stop the execution with the result in register $\texttt{r}_\texttt{0}$.

Figure 3: Instruction set $\mathcal{I}$.

3.2. Operational Semantics

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

Figure 4 defines the access control enforcement rules informally presented in Section 2.1. Read judgments $s \vdash \texttt{predicate}(a, b, \cdots)$ as: "according to memory descriptor s, $\texttt{predicate}$ holds for addresses a, b, $\cdots$".

Define functions $\texttt{m}_\texttt{sec}(m, s)$ and $\texttt{m}_\texttt{ext}(m, s)$, which return the protected and

$$\frac{a_b \le p < (a_b + n_c + n_d)}{s \vdash \texttt{protected}(p)}\ \text{(Aux-protected)} \qquad \frac{p < a_b}{s \vdash \texttt{unprotected}(p)}\ \text{(Aux-unprotected1)} \qquad \frac{(a_b + n_c + n_d) \le p}{s \vdash \texttt{unprotected}(p)}\ \text{(Aux-unprotected2)}$$

$$\frac{a_{uc} \le a < a_{ud}}{s \vdash \texttt{unprotectedCode}(a)}\ \text{(Aux-unprotected-code)} \qquad \frac{a_{ud} \le a \quad s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{unprotectedData}(a)}\ \text{(Aux-unprotected-data)}$$

$$\frac{p = a_b + (n-1) \cdot \mathcal{N}_e}{s \vdash \texttt{returnEntryPoint}(p)}\ \text{(Aux-returnEntry)} \qquad \frac{p = a_b + m \cdot \mathcal{N}_e \quad m \in \mathbb{N} \quad m < n}{s \vdash \texttt{entryPoint}(p)}\ \text{(Aux-entryPoint)} \qquad \frac{(a_b + n_c) \le p \quad p < (a_b + n_c + n_d)}{s \vdash \texttt{data}(p)}\ \text{(Aux-data)}$$

$$\frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{protected}(a)}{s \vdash \texttt{readAllowed}(p, a)}\ \text{(Aux-read-1)} \qquad \frac{s \vdash \texttt{unprotectedData}(a)}{s \vdash \texttt{readAllowed}(p, a)}\ \text{(Aux-read-2)} \qquad \frac{s \vdash \texttt{unprotectedData}(a)}{s \vdash \texttt{writeAllowed}(p, a)}\ \text{(Aux-write-1)}$$

$$\frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{data}(a)}{s \vdash \texttt{writeAllowed}(p, a)}\ \text{(Aux-write-2)} \qquad \frac{s \vdash \texttt{unprotectedCode}(p) \quad s \vdash \texttt{entryPoint}(p')}{s \vdash \texttt{entryJump}(p, p')}\ \text{(Aux-entry)}$$

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

Figure 4: Access control enforcement rules. Assume $s = (a_b, n_c, n_d, n, a_{uc}, a_{ud})$.

unprotected parts of a memory m according to descriptor s, respectively as:

$$\texttt{m}_{\mathsf{sec}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \texttt{protected}(a)\}$$
$$\texttt{m}_{\mathsf{ext}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \texttt{unprotected}(a)\}$$

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 top of the stack: $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$ are memory locations that indicate the top of the secure and insecure stack respectively. Given a memory descriptor $s = (a_b, n_c, n_d, n, a_{uc}, a_{ud})$, the secure stack starts at the beginning of the protected data section and the insecure stack starts at beginning of the unprotected data section, the stack grows up. Thus $\mathsf{SP}_{\mathsf{sec}} = (a_b + n_c)$ and, initially, $\mathsf{SP}_{\mathsf{sec}} \mapsto (a_b + n_c + 1)$; analogously, $\mathsf{SP}_{\mathsf{ext}} = (a_{ud})$ and, initially, $\mathsf{SP}_{\mathsf{ext}} \mapsto (a_{ud} + 1)$. Call and return instructions see the SP register being set to the correct address when crossing boundaries between protected and unprotected memory by using $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$. The value of the program counter is pushed onto the stack by a `call` instruction, 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 using the auxiliary function $\searrow^{\mathsf{SS}}$ (Figure 5).

In the rules, notation $m[a \mapsto w]$ indicates that memory m is updated to

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

$$\frac{\begin{array}{c} s \vdash \texttt{intJump}(p,p') \\ s \vdash \texttt{protected}\ r(\mathsf{SP}) \end{array}}{p,r,m,s \searrow^{\mathsf{SS}} p',r,m} \ (\text{Stack-no-change-i}) \qquad \frac{\begin{array}{c} s \vdash \texttt{extJump}(p,p') \\ s \vdash \texttt{unprotected}\ r(\mathsf{SP}) \end{array}}{p,r,m,s \searrow^{\mathsf{SS}} p',r,m} \ (\text{Stack-no-change-e})$$

Figure 5: Stack switch enforcement rules.

a new one that is equal to m except that the value stored at address a is w. Notation $r[\mathtt{r} \mapsto w]$ indicates that the register file r is updated to a new one that is equal to r except that the value stored in register $\mathtt{r}$ is w. Notation $r(\mathtt{r})$ indicates the value contained in register $\mathtt{r}$ 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. When the stack is changed, the register file is updated to initialise SP to the top of the right stack: the address stored at $\mathsf{SP_{sec}}$ or $\mathsf{SP_{ext}}$.

The operational semantics is a small step semantics that describes how each instruction of the language transforms an execution state into a new one. The operational semantics describes 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 execution state $\Omega = (p, r, f, m, s)$, let $\lfloor\Omega\rfloor$ be the state $(p, r, f, \mathtt{m_{sec}}(m, s), s)$ and $\lceil\Omega\rceil$ be the state $(p, r, f, \mathtt{m_{ext}}(m, s), s)$. Relations $\xrightarrow{i}$ and $\xrightarrow{e}$ describe the evaluation of instructions that only affect the protected and unprotected parts of memory respectively. These relations build up to the complete operational semantics. Rules for $\xrightarrow{e}$ can be obtained from the rules for $\xrightarrow{i}$ (Figures 6 to 7) by replacing all `intJump` assumptions with an `extJump` one and are thus omitted. Let $m(p) = \mathtt{inst}$ denote that `inst` is the word allocated in $m(p)$, where $\mathtt{inst} \in \mathcal{I}$. When an access control violation is detected, or when the secure stack is overflowed, all registers and flags are reset and the execution is halted. 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*

$$\frac{m(p) = (\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{readAllowed}(p, r(\texttt{r}_\texttt{s})) \quad r' = r[\texttt{r}_\texttt{d} \mapsto m(r(\texttt{r}_\texttt{s}))]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)} \text{(Eval-movl)}$$

$$\frac{m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{writeAllowed}(p, r(\texttt{r}_\texttt{d})) \quad m' = m[r(\texttt{r}_\texttt{d}) \mapsto r(\texttt{r}_\texttt{s})]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m', s)} \text{(Eval-movs)}$$

$$\frac{m(p) = (\texttt{movi}\ \texttt{r}_\texttt{d}\ \texttt{i}) \quad s \vdash \texttt{intJump}(p, p+1) \quad r' = r[\texttt{r}_\texttt{d} \mapsto i]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)} \text{(Eval-movi)}$$

$$\frac{m(p) = (\texttt{cmp}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{d}) \quad s \vdash \texttt{intJump}(p, p+1) \quad f' = f[\mathsf{ZF} \mapsto (\texttt{r}_\texttt{s} == \texttt{r}_\texttt{d}); \mathsf{SF} \mapsto (\texttt{r}_\texttt{s} < \texttt{r}_\texttt{d})]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f', m, s)} \text{(Eval-compare)}$$

$$\frac{m(p) = (\texttt{add}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad v = (r(\texttt{r}_\texttt{d}) + r(\texttt{r}_\texttt{s})) \quad r' = r[\texttt{r}_\texttt{d} \mapsto v] \quad f' = f[\mathsf{ZF} \mapsto (v == 0)]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)} \text{(Eval-add)}$$

$$\frac{m(p) = (\texttt{sub}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad v = (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s})) \quad r' = r[\texttt{r}_\texttt{d} \mapsto v] \quad f' = f[\mathsf{ZF} \mapsto (v == 0); \mathsf{SF} \mapsto (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s}) < 0)]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)} \text{(Eval-sub)}$$

Figure 6: Operational semantics of instructions in the protected memory partition (part I).

$$\frac{m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = r(\texttt{r}_\texttt{d}) \quad s \vdash \texttt{intJump}(p, p') \quad p, r, m, s \searrow^{\mathsf{SS}} p', r', m' \quad r'' = r'[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \quad m'' = m'[r''(\mathsf{SP}) \mapsto p + 1]}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m'', s)} \text{(Eval-function-call)}$$

$$\frac{m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) \quad s \vdash \texttt{intJump}(p, p') \quad r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \quad p, r', m, s \searrow^{\mathsf{SS}} p', r'', m'}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m', s)} \text{(Eval-function-ret)}$$

$$\frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{ZF}) == 1 \quad p' = r(\texttt{r}_\texttt{i}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)} \text{(Eval-je-true)}$$

$$\frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{SF}) == 1 \quad p' = r(\texttt{r}_\texttt{i}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)} \text{(Eval-jl-true)}$$

$$\frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{ZF}) == 0 \quad s \vdash \texttt{intJump}(p, p+1)}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)} \text{(Eval-je-false)}$$

$$\frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{SF}) == 0 \quad s \vdash \texttt{intJump}(p, p+1)}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)} \text{(Eval-jl-false)}$$

$$\frac{m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}) \quad p' = r(\texttt{r}_\texttt{d}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)} \text{(Eval-jump)}$$

$$\frac{m(p) = (\texttt{halt})}{(p, r, f, m, s) \xrightarrow{i} (-1, r, f, m, s)} \text{(Eval-halt)}$$

Figure 7: Operational semantics of instructions in the protected memory partition (part II).

$\Omega^\perp$, *when the program counter does not point to a valid instruction:* $m(p) \notin \mathcal{I}$.

The operational semantics of $\mathcal{A}$+I is a binary relation over states $\rightarrow \subseteq \Omega \times \Omega$ (Figures 8 to 9). Rule Eval-callback and Eval-returnback ensure that the

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

$$\frac{\begin{array}{c} m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{writeAllowed}(p, r(\texttt{r}_\texttt{d})) \\ s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{d})) \quad m' = m[r(\texttt{r}_\texttt{d}) \mapsto r(\texttt{r}_\texttt{s})] \end{array}}{(p, r, f, m, s) \to (p+1, r, f, m', s)}\ (\textsf{Eval-movs-out})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{readAllowed}(p, r(\texttt{r}_\texttt{s})) \\ s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{s})) \quad r' = r[\texttt{r}_\texttt{d} \mapsto m(r(\texttt{r}_\texttt{s}))] \end{array}}{(p, r, f, m, s) \to (p+1, r', f, m, s)}\ (\textsf{Eval-movl-out})$$

Figure 8: Operational semantics of whole $\mathcal{A}$+I programs (part 1).

$$\frac{\begin{array}{c} m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = r(\texttt{r}_\texttt{d}) \quad s \vdash \texttt{exitJump}(p, p') \quad r' = r[\textsf{SP} \mapsto r(\textsf{SP}) + 1] \\ m' = m[r(\textsf{SP}) \mapsto p+1] \quad p, r', m', s \searrow^{\textsf{SS}} p', r'', m'' \quad r''' = r''[\textsf{SP} \mapsto r''(\textsf{SP}) + 1] \\ m''' = m''[r'''(\textsf{SP}) \mapsto a] \quad s \vdash \texttt{returnEntryPoint}(a) \end{array}}{(p, r, f, m, s) \to (p', r''', f, m''', s)}\ (\textsf{Eval-callback})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = r(\texttt{r}_\texttt{d}) \quad s \vdash \texttt{entryJump}(p, p') \\ p, r, m, s \searrow^{\textsf{SS}} p', r', m' \quad r'' = r'[\textsf{SP} \mapsto r'(\textsf{SP}) + 1] \quad m'' = m'[r''(\textsf{SP}) \mapsto p+1] \end{array}}{(p, r, f, m, s) \to (p', r'', f, m'', s)}\ (\textsf{Eval-call})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{ret}) \quad p' = m(r(\textsf{SP})) \quad s \vdash \texttt{entryJump}(p, p') \\ r' = r[\textsf{SP} \mapsto r(\textsf{SP}) - 1] \quad p, r', m, s \searrow^{\textsf{SS}} p', r'', m' \quad s \vdash \texttt{returnEntryPoint}(p') \end{array}}{(p, r, f, m, s) \to (p', r'', f, m', s)}\ (\textsf{Eval-returnback})$$

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

Figure 9: Operational semantics of whole $\mathcal{A}$+I programs (part 2).

address to be followed after a callback is stored in the secure stack and that the address of the returnback entry point is pushed onto 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.

The reflexive, transitive closure of relation $\to$ is indicated with $\to^*$. A state Ω performing n reduction steps is indicated as $\Omega \to^n \Omega'$. 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 a state* (p_0, r_0, f_0, m, s)*, where* $s = (a_b, n_c, n_d, n)$*,* $p_0 = (a_b + n_c + n_d + 2)$*,* $r_0 = [\textsf{SP} \mapsto m(\textsf{SP}_{\textsf{ext}}); \texttt{r}_\texttt{i} \mapsto 0\ _{\texttt{i}=0..11}]$*, and* $f_0 = [\textsf{ZF} \mapsto 0; \textsf{SF} \mapsto 0]$*.*

The evaluation of P terminates if $\exists \Omega'.\Omega_0(P) \to^* \Omega'$ and $\Omega'^{\perp}$; the result of the

computation is stored in $\mathtt{r}_0$. If the evaluation of program P does not terminate, P diverges, i.e. it executes an unbounded number of reduction steps, this is denoted as $P\Uparrow$. Formally: $P\Uparrow$ if $\forall n \in \mathbb{N}, \exists \Omega'.\ \Omega_0(P) \rightarrow^n \Omega'$.

3.3. Contextual Equivalence for $\mathcal{A}$+I

Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [20]. This notion relies on the concept of contexts, which is introduced before presenting the equivalence itself.

Since our focus is on $\mathcal{A}$+I 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, $\forall a \in \mathtt{dom}(\mathbb{M}), s \vdash \mathtt{unprotected}(a)$. 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 $\mathtt{dom}(\mathbb{M}) = \mathtt{dom}(m)$ if $\mathbb{M} = m[\cdot]$; formally, $P \frown \mathbb{M}$ if $P = (m', s)$ and $\mathtt{dom}(m') \cap \mathtt{dom}(\mathbb{M}) = \emptyset$. 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, always assume the compatibility of a protected 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 17 (Contextually equivalent programs). *The following programs* P_L *and* P_R *write the values of* $\mathtt{r}_1$ *and* $\mathtt{r}_2$ *respectively to 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, thereby establishing full abstraction of the trace semantics.

4. Trace Semantics for $\mathcal{A}$+I

This section gives two different trace semantics for protected $\mathcal{A}$+I programs. The differences between these semantics stem out of the different ways to achieve full abstraction pointed out by Curien [19]. The first trace semantics, Tr^{S}, relies on changes to the semantics of protected programs (Section 4.1), while the second one, Tr^{L}, possesses more expressive labels (Section 4.2). 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 (Section 4.3).

4.1. Tr^{S}*: Changes to the Semantics*

As for the operational semantics, a notion of execution states is required for the trace semantics as well. Execution states for Tr^{S}, denoted as Θ, are the same as Ω except that Θ does not deal with the whole memory, just with its protected partition. So, the memory m of (p, r, f, m, s) spans only the protected memory partition indicated by memory descriptor s. Additionally, Θ can be (unk, m, s), an unknown state that models when code is executing in unprotected memory [16].

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* (unk, m, s)*.*

The semantics of protected programs is changed as follows (Figure 10):

- when the program counter jumps between the protected and the unprotected memory partitions, or vice-versa, flags are set to 0 (Rule Stack-out-to-in' and Stack-in-to-out');
- in case of a `return`, all registers but $\mathtt{r}_0$ are also set to 0 (Rule Eval-return');
- readouts and writeouts are prohibited (Rule Aux-write-1' and Aux-read-2' replace the access control rules with the homonimous name).

These changes do not limit the expressivity of the language, they ensure communication between protected and unprotected code happens in a specific fashion.

Following are the labels of Tr^{S}, they include those presented in Section 2.2. Observable actions include a tick $\surd$ indicating that the evaluation has terminated. Flags do not appear in traces because they are always set to 0, as are all registers but $\mathtt{r}_0$ in case of a return. Readouts and writeouts are prohibited, so there are no labels that capture them.

$$
\begin{aligned}
\textit{Labels} \quad & L ::= a \mid \tau_i \\
\textit{Observable actions} \quad & a ::= \surd \mid g? \mid g! \\
\textit{Actions} \quad & g ::= \mathtt{call}\ p\ (r) \mid \mathtt{ret}\ p\ r(\mathtt{r}_0)
\end{aligned}
$$

(Aux-write-1')

$$\frac{s \vdash \texttt{unprotectedCode}(p) \qquad s \vdash \texttt{unprotectedData}(a)}{s \vdash \texttt{writeAllowed}(p, a)}$$

(Aux-read-2')

$$\frac{s \vdash \texttt{unprotectedCode}(p) \qquad s \vdash \texttt{unprotectedData}(a)}{s \vdash \texttt{readAllowed}(p, a)}$$

(Stack-out-to-in')

$$\frac{\begin{array}{c} s \vdash \texttt{entryJump}(p, p') \\ m' = m[\mathsf{SP_{ext}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP_{sec}})] \\ f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \texttt{unprotected } r(\mathsf{SP}) \\ s \vdash \texttt{protected } r'(\mathsf{SP}) \end{array}}{p, r, f, m, s \searrow^{\mathsf{SS}} p', r', f', m'}$$

(Stack-in-to-out')

$$\frac{\begin{array}{c} s \vdash \texttt{exitJump}(p, p') \\ m' = m[\mathsf{SP_{sec}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP_{ext}})] \\ f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \texttt{protected } r(\mathsf{SP}) \\ s \vdash \texttt{unprotected } r'(\mathsf{SP}) \end{array}}{p, r, f, m, s \searrow^{\mathsf{SS}} p', r', f', m'}$$

(Eval-return')

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

Figure 10: Changes to auxiliary functions and to the operational semantics for Tr^{S}.

(Trace-s-internal)

$$\frac{(p, r, f, m, s) \xrightarrow{i} (p', r', f', m', s) \qquad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{\tau_i}\!\!\!\!\!\!\rightarrow (p', r', f', m', s)}$$

(Trace-s-termination)

$$\frac{(p, r, f, m, s) \xrightarrow{i} (p', r', f', m', s) \qquad (p', r', f', m', s)^{\perp}}{(p, r, f, m, s) \xrightarrow{\checkmark}\!\!\!\!\!\!\rightarrow (p', r', f', m', s)}$$

(Trace-s-call)

$$\frac{s \vdash \texttt{entryPoint}(p) \qquad f = [\mathsf{SF} \mapsto 0; \mathsf{ZF} \mapsto 0]}{(\mathsf{unk}, m, s) \xrightarrow{\texttt{call}\ p\ (r)?}\!\!\!\!\!\!\rightarrow (p, r, f, m, s)}$$

(Trace-s-returnback)

$$\frac{s \vdash \texttt{returnEntryPoint}(p) \qquad f = [\mathsf{SF} \mapsto 0; \mathsf{ZF} \mapsto 0]}{(\mathsf{unk}, m, s) \xrightarrow{\texttt{ret}\ p\ r(\texttt{r}_0)?}\!\!\!\!\!\!\rightarrow (p, r, f, m, s)}$$

(Trace-s-callback)

$$\frac{s \vdash \texttt{exitJump}(p, p') \qquad m(p) = (\texttt{call } p') \qquad m' = m[r(\mathsf{SP}) \mapsto p + 1]}{(p, r, f, m, s) \xrightarrow{\texttt{call}\ p'\ (r)!}\!\!\!\!\!\!\rightarrow (\mathsf{unk}, m', s)}$$

(Trace-s-return)

$$\frac{p' \in 0...2^{\ell} \qquad s \vdash \texttt{exitJump}(p, p') \qquad m(p) = (\texttt{ret})}{(p, r, f, m, s) \xrightarrow{\texttt{ret}\ p'\ r(\texttt{r}_0)!}\!\!\!\!\!\!\rightarrow (\mathsf{unk}, m, s)}$$

Figure 11: Rules of the Tr^{S} trace semantics.

The relation $\twoheadrightarrow$ defines how labels are generated (Figure 11). Internal instructions, generated by a $\xrightarrow{i}$ transition, do not produce a visible label (Rule Trace-s-internal). If a state is stuck, then the label for termination is produced (Rule Trace-s-termination). A call to an entry point generates a call label (Rule Trace-s-call) while a return to the returnback entry point generates a returnback label (Rule Trace-s-returnback). Calling to an unprotected address generates a callback label (Rule Trace-s-callback), while returning to any unprotected address generates a return label (Rule Trace-s-return).

The reflexive and transitive closure of $\twoheadrightarrow$, denoted with $\Rightarrow\!\!\!\!\!\Rightarrow$, is responsible for the accumulation of labels into traces (Figure 12).

$$\frac{}{\Theta \overset{\epsilon}{\Longrightarrow\!\!\!\!\!\!\to} \Theta}\ \text{(Trace-s-refl)} \qquad \frac{\Theta \xrightarrow{\tau_i}\!\!\!\!\to \Theta'}{\Theta \overset{\epsilon}{\Longrightarrow\!\!\!\!\!\!\to} \Theta'}\ \text{(Trace-s-tau-i)}$$

$$\frac{\Theta \overset{\overline{a}}{\Longrightarrow\!\!\!\!\!\!\to} \Theta'' \qquad \Theta'' \overset{\overline{a}'}{\Longrightarrow\!\!\!\!\!\!\to} \Theta'}{\Theta \overset{\overline{a}\cdot\overline{a}'}{\Longrightarrow\!\!\!\!\!\!\to} \Theta'}\ \text{(Trace-s-trans)} \qquad \frac{\Theta \xrightarrow{a}\!\!\!\!\to \Theta'}{\Theta \overset{a}{\Longrightarrow\!\!\!\!\!\!\to} \Theta'}\ \text{(Trace-s-action)}$$

Figure 12: Reflexive and transitive closure of the Tr^{S} trace semantics rules.

The Tr^{S} traces of a program P are defined as follows:

$$\mathsf{Tr}^{\mathsf{S}}(P) = \{\overline{a} \mid \exists\Theta.\ \Theta_0(P) \overset{\overline{a}}{\Longrightarrow\!\!\!\!\!\!\to} \Theta\}$$

4.2. Tr^{L}: Expressive Labels

This section presents Tr^{L}, a trace semantics that changes the labels of Section 2.2 to include all possible observable behaviour, including readouts and writeouts. The semantics used here is the one presented in Section 3.

The states of the Tr^{L} semantics are indicated with Θ, they do not change from the definition given for the Tr^{S} semantics. The syntax of labels, however, changes as indicated below, including a readout and a writeout label.

$$\begin{aligned}
\textit{Labels} &\quad \lambda ::= \tau \mid \alpha \\
\textit{Observable actions} &\quad \alpha ::= \gamma? \mid \delta! \mid \surd \\
\textit{Actions} &\quad \gamma ::= \texttt{call}\ p(r;f) \mid \texttt{ret}\ p(r;f) \\
\textit{Prefixable actions} &\quad \delta ::= \gamma \mid \omega(a,v).\delta \\
\textit{Prefixes} &\quad \omega ::= \texttt{read} \mid \texttt{write}
\end{aligned}$$

To ensure that the issues of Examples 6 to 7 and Examples 13 to 15 (Section 2.2.1 and 2.2.2) do not arise, $\delta!$ labels are converted to a normal form.

The normal form of $\delta!$ labels is achieved by applying the rewrite rules presented in Figure 13. Rules (Constraint-write) to (Constraint-read) ensure that labels created by the semantics are consistent [21]. Rule (Write-order), (Read-order), (WR-order) and (RW-order) ensure the prefix of reads and writes are sorted based on the address field. If two actions are performed at the same address, their order is the same as the order in which the program performed those actions. Rules (Write-no-read) to (Read-no-write) ensure that reading the same writeout value (resp. writing the same readout value) does not appear in labels. Rules (Write-drop) to (Read-drop) eliminate multiple writeouts and readouts to the same address.

The rewrite rules of Figure 13 are convergent so their application always returns a unique result (Theorem 3 in Appendix A). We can thus define the normal-form function $\mathsf{norm}(\cdot)$ as the application of those rewrite rules. This function inputs a δ label and returns it in normal form, i.e. a sequence of $\texttt{write}(a,v)$ and $\texttt{read}(a,v)$ label sorted on the address parameter a.

The rules that define the single label relation $\longrightarrow\!\!\!\!\to \subseteq \Theta \times \lambda \times \Theta$ (Figure 14) rely on the semantics presented in Section 3.2. Rules for generating $\texttt{call}$, $\texttt{return}$ and τ labels resemble the rules for the Tr^{S} semantics. Rule Trace-tau-compression

$$\texttt{write}(a,v)\texttt{read}(a,v') \Rightarrow v = v' \quad \text{(Constraint-write)}$$
$$\texttt{read}(a,v)\texttt{read}(a,v') \Rightarrow v = v' \quad \text{(Constraint-read)}$$
$$\texttt{write}(a,v)\texttt{write}(a',v') \rightsquigarrow \texttt{write}(a',v')\texttt{write}(a,v) \quad \text{if } a' < a \quad \text{(Write-order)}$$
$$\texttt{read}(a,v)\texttt{read}(a',v') \rightsquigarrow \texttt{read}(a',v')\texttt{read}(a,v) \quad \text{if } a' < a \quad \text{(Read-order)}$$
$$\texttt{read}(a,v)\texttt{write}(a',v') \rightsquigarrow \texttt{write}(a',v')\texttt{read}(a,v) \quad \text{if } a' < a \quad \text{(WR-order)}$$
$$\texttt{write}(a,v)\texttt{read}(a',v') \rightsquigarrow \texttt{read}(a',v')\texttt{write}(a,v) \quad \text{if } a' < a \quad \text{(RW-order)}$$
$$\texttt{write}(a,v)\texttt{read}(a,v) \rightsquigarrow \texttt{write}(a,v) \quad \text{(Write-no-read)}$$
$$\texttt{read}(a,v)\texttt{write}(a,v) \rightsquigarrow \texttt{read}(a,v) \quad \text{(Read-no-write)}$$
$$\texttt{write}(a,v)\texttt{write}(a,v') \rightsquigarrow \texttt{write}(a,v') \quad \text{(Write-drop)}$$
$$\texttt{read}(a,v)\texttt{read}(a,v) \rightsquigarrow \texttt{read}(a,v) \quad \text{(Read-drop)}$$

Figure 13: Rewrite rules to reduce a $\delta!$ label to its normal form.

ensures that τ labels are not accumulated, so readout and writeout labels are not spaced out with τs. For writeouts, Rule Trace-writeout ensures writeout labels are always created, dually, for readouts, Rule Trace-readout ensures readout labels are always created when reading unprotected data. Rule Trace-writeout-termination addresses Example 8, so no writeout label is created when a program terminates.

The reflexive transitive closure of the $\longrightarrow\!\!\!\!\!\rightarrow$ relation is captured by relation $\Longrightarrow\!\!\!\!\!\!\Rightarrow \subseteq \Theta \times \overline{\alpha} \times \Theta$ (Figure 15). The only difference with the way this is performed with regards to Tr^{S} (Figure 12) is that when a label is produced in Tr^{L}, it is converted to a normal form via the $\mathsf{norm}(\cdot)$ function and stripped of its non-interfering reads via the $\mathsf{StripNI}(\cdot)$ function (Figure 16 defined below).

The trace semantics of a state is defined as follows:

$$\mathsf{Tr\text{-}state}(\Theta) = \{\overline{\alpha} \mid \exists \Theta'.\Theta \xRightarrow{\overline{\alpha}} \Theta'\}$$

Thus, the Tr^{L} traces of a program P are defined as the traces of its initial state:

$$\mathsf{Tr}^{\mathsf{L}}(P) = \mathsf{Tr\text{-}state}(\Theta_0(P))$$

The greatest concern when adding readouts is detecting whether a readout is non-interfering, as explained in Examples 10 to 12. In fact, non-interfering readouts must not have a corresponding label in traces. To understand whether a readout to a certain address is non-interfering, we rely on judgment $\mathsf{NI}(\Theta, a)$. That judgment tells whether an address a is non-interfering for a state Θ if Θ performs a readout to a that does not affect future traces but for the read value. Formally:

$$\begin{aligned}\mathsf{NI}(\Theta, a) \triangleq \forall v, w. \quad & \Theta \xRightarrow{\alpha_1} \Theta' \text{ and } \Theta \xRightarrow{\alpha_2} \Theta'' \\ & \text{and } \alpha_1 = \overline{\omega(a',v')}\texttt{read}(a,v)\delta! \text{ and } \alpha_2 = \overline{\omega(a',v')}\texttt{read}(a,w)\delta! \\ & \text{and } \mathsf{Tr\text{-}state}(\Theta') = \mathsf{Tr\text{-}state}(\Theta'')\end{aligned}$$

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

$$\frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \quad (p',r',f',m',s)^{\perp}}{(p,r,f,m,s) \xrightarrow{\checkmark}\!\!\!\!\!\rightarrow (p',r',f',m',s)}\ \text{(Trace-termination)}$$

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

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

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

$$\frac{m(p) = (\texttt{ret}) \quad p' \in 0...2^{\ell} \quad s \vdash \texttt{exitJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\texttt{ret } p'(r;f)!}\!\!\!\!\!\rightarrow (\mathsf{unk},m,s)}\ \text{(Trace-return)}$$

$$\frac{\Theta \xrightarrow{\tau}\!\!\!\!\!\rightarrow \Theta' \quad \Theta' \xrightarrow{\alpha}\!\!\!\!\!\rightarrow \Theta''}{\Theta \xrightarrow{\alpha}\!\!\!\!\!\rightarrow \Theta''}\ \text{(Trace-tau-compression)}$$

$$\frac{m(p) = \texttt{movs r}_\texttt{d}\ \texttt{r}_\texttt{s} \quad s \vdash \texttt{intJump}(p,p+1) \quad s \vdash \texttt{unprotectedData}(a) \quad a = r(\texttt{r}_\texttt{d}) \quad v = r(\texttt{r}_\texttt{s}) \quad (p+1,r,f,m,s) \xrightarrow{\delta!}\!\!\!\!\!\rightarrow (\mathsf{unk},m,s) \quad v' \in \mathcal{W}}{(p,r,f,m,s) \xrightarrow{\texttt{write}(a,v)\delta!}\!\!\!\!\!\rightarrow (\mathsf{unk},m,s)}\ \text{(Trace-writeout)}$$

$$\frac{m(p) = \texttt{movs r}_\texttt{d}\ \texttt{r}_\texttt{s} \quad s \vdash \texttt{intJump}(p,p+1) \quad s \vdash \texttt{unprotectedData}(a) \quad (p+1,r,f,m,s) \xrightarrow{\overline{\omega(a,v)}\checkmark}\!\!\!\!\!\rightarrow (p',r',f',m',s')}{(p,r,f,m,s) \xrightarrow{\overline{\omega(a,v)}\checkmark}\!\!\!\!\!\rightarrow (p',r',f',m',s')}\ \text{(Trace-writeout-termination)}$$

$$\frac{m(p) = \texttt{movl r}_\texttt{d}\ \texttt{r}_\texttt{s} \quad s \vdash \texttt{intJump}(p,p+1) \quad s \vdash \texttt{unprotectedData}(a) \quad a = r(\texttt{r}_\texttt{s}) \quad v \in \mathcal{W} \quad r'' = r[\texttt{r}_\texttt{d} \mapsto v] \quad (p+1,r'',f,m,s) \xrightarrow{\delta!}\!\!\!\!\!\rightarrow (\mathsf{unk},m,s)}{(p,r,f,m,s) \xrightarrow{\texttt{read}(a,v)\delta!}\!\!\!\!\!\rightarrow (\mathsf{unk},m,s)}\ \text{(Trace-readout)}$$

Figure 14: Rules for the Tr^{L} trace semantics for writeouts and readouts labels.

$$\frac{}{\Theta \xRightarrow{\epsilon} \Theta}\ \text{(Trace-l-refl)} \qquad \frac{\Theta \xrightarrow{\tau_i}\!\!\!\!\!\rightarrow \Theta'}{\Theta \xRightarrow{\epsilon} \Theta'}\ \text{(Trace-l-tau-i)} \qquad \frac{\Theta \xRightarrow{\overline{\alpha}} \Theta'' \quad \Theta'' \xRightarrow{\overline{\alpha}'} \Theta'}{\Theta \xRightarrow{\overline{\alpha}\cdot\overline{\alpha}'} \Theta'}\ \text{(Trace-l-trans)} \qquad \frac{\Theta \xrightarrow{\alpha}\!\!\!\!\!\rightarrow \Theta' \quad \mathsf{StripNI}(\Theta, \mathsf{norm}(\alpha)) = \alpha'}{\Theta \xRightarrow{\alpha'} \Theta'}\ \text{(Trace-l-action)}$$

Figure 15: Reflexive and transitive closure of relation $\longrightarrow\!\!\!\!\!\rightarrow$ for Tr^{L}.

The $\mathsf{NI}(\cdot)$ definition relies on the formalisation of $\mathsf{Tr\text{-}state}(\Theta)$ which returns the set of traces that can be generated from Θ; it is formalised below. $\mathsf{Tr\text{-}state}(\cdot)$ is used to access the behaviour of the program after either value is read from address a, no difference can be found there for the readout to be non-interfering. It is not sufficient to check the single immediate action δ following the readout, as the readout value could be stored in memory and be used only for successive computations. The prefix $\overline{\omega(a',v')}$ makes it possible to identify a readout that

happens at any point during the first action.

Note that the definition of $\mathsf{Tr\text{-}state}(\cdot)$ and that of $\mathsf{NI}(\cdot)$ are mutually recursive. However, they are still well-founded since $\mathsf{Tr\text{-}state}(\cdot)$ uses $\mathsf{NI}(\cdot)$ when filtering a label $\delta!$ generated as $\Theta \xrightarrow{\delta!}\!\!\!\!\!\rightarrow \Theta'$ and then $\mathsf{NI}(\cdot)$ relies on $\mathsf{Tr\text{-}state}(\cdot)$ on the traces generated from Θ' onwards.

With this information, define a function $\mathsf{StripNI}(\Theta, \alpha)$ that returns α' which is α stripped of its non-interfering reads, provided that α is generated from Θ (Figure 16). Since this function preserves the ordering of the labels in α, when

$$
\begin{aligned}
\mathsf{StripNI}(\Theta, \gamma) &= \gamma \\
\mathsf{StripNI}(\Theta, \mathtt{write}(a, v)\delta) &= \mathtt{write}(a, v)\delta' && \text{if } \mathsf{StripNI}(\Theta, \delta) = \delta' \\
\mathsf{StripNI}(\Theta, \mathtt{read}(a, v)\delta) &= \delta' && \text{if } \mathsf{StripNI}(\Theta, \delta) = \delta' \text{ and } \mathsf{NI}(\Theta, a) \\
\mathsf{StripNI}(\Theta, \mathtt{read}(a, v)\delta) &= \mathtt{read}(a, v)\delta' && \text{if } \mathsf{StripNI}(\Theta, \delta) = \delta' \text{ and } \neg\mathsf{NI}(\Theta, a)
\end{aligned}
$$

Figure 16: Function to strip a label of its non-interfering reads.

applied to labels in normal form it still produces labels in normal form.

4.3. Trace Equivalence for $\mathcal{A}$+I Programs

The notion of trace equivalence is presented generically for both trace semantics under consideration. Use $\mathsf{Tr}(P)$ to indicate the traces of an $\mathcal{A}$+I program P, be it expressed through Tr^{S} or Tr^{L}. 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 indicate arbitrary values for registers and flags with notation (r, f) and an unprotected address with p.

Example 18 (Traces of previous examples). *The code of Example 5 is* not *trace equivalent; the following trace is generated by P_L but not by P_R:*

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

The code of Example 12 is not trace equivalent; the following trace is generated by P_L but not by P_R:

$$\mathtt{call}\ 100\ (r; f)? \cdot \mathtt{read}(10, v)\mathtt{call}\ 30(30, \ldots; f)!$$

The code of Example 17 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 having possibly different values of r and f:

$$\mathtt{call}\ 100\ (r; f)? \cdot \mathtt{ret}\ p\ (r[\mathtt{r_0} \mapsto 0]; f)!$$

5. Full Abstraction of the Trace Semantics

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

A fully abstract trace semantics is both sound and complete with respect to the operational semantics. Soundness means that the trace semantics captures all behaviours expressible with the operational semantics. Thus, for all contexts, two trace equivalent programs cannot be told apart. Completeness means that the trace semantics does not capture additional behaviours that are not expressible with the operational semantics. Thus, for all trace-inequivalent programs, there exists a context that can differentiate them.

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.

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 $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m_2, s_2)$. Given $\Omega = (p, r, f, m, s)$, define $\|\Omega\|$ to be the state $\Theta = (p, r, f, \mathtt{m_{sec}}(m, s), s)$ if $s \vdash \mathtt{protected}(p)$ and (unk, m, s) otherwise.

The proof of soundness (Theorem 1 below) states that an unprotected program interacting with P_1 cannot distinguish it from P_2. The proof strategy relies on both programs offering the same interface to the unprotected program. 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 says 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.

The proofs of Lemma 1 and of Theorems 1 to 2 can be found in Appendix B, C and D.

Lemma 1 (Interface preservation after same observable action). *If* $\Theta_1 \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \overset{\alpha}{\longrightarrow\!\!\!\!\!\!\rightarrow} \Theta_1'$ *and* $\Theta_1 = \|\Omega_1\|$ *and* $\Omega_1 \to^* \Omega_1'$ *and* $\Theta_1' = \|\Omega_1'\|$ *and* $\Theta_2 \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \overset{\alpha}{\longrightarrow\!\!\!\!\!\!\rightarrow} \Theta_2'$ *and* $\Theta_2 = \|\Omega_2\|$ *and* $\Omega_2 \to^* \Omega_2'$ *and* $\Theta_2' = \|\Omega_2'\|$ *and* $\Omega_1 \mathrel{\mathring{=}} \Omega_2$ *then* $\Omega_1' \mathrel{\mathring{=}} \Omega_2'$ *(assuming there is no overflow of the secure stack).*

Theorem 1 (Soundness). $P_1 \simeq_{\mathrm{T}} P_2 \Rightarrow P_1 \simeq P_2$ *(assuming there is no overflow of the secure stack).*

Theorem 2 (Completeness). $P_1 \simeq P_2 \Rightarrow P_1 \simeq_{\mathrm{T}} P_2$ *(assuming there is no overflow of the secure stack).*

This general proof strategy is presented for both Theorem 1 and 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.

6. Related Work

Full abstraction has been largely studied as a way to formalise the correctness of a denotational semantics with respect to an operational one [20]. It has been studied for different programming languages paradigms, such as the λ-calculus [22] and the π-calculus [23].

Trace semantics was developed to study the behaviour of concurrent CSP [24] and it has been adopted for describing concurrent and distributed language behaviour [25]. Several works have devised fully abstract trace semantics for functional [26, 27, 28, 29] and object-oriented [16, 30] languages. Abadi and Plotkin [26] developed a fully abstract trace semantics for a λ-calculus with references in order to prove a secure compilation using Address Space Layout Randomisation secure. Jagadeesan *et al.* [27] extended the results of Abadi and Plotkin to a λ-calculus with more advanced language features and equipped that language with a fully abstract trace semantics for secure compilation purposes. While the languages are different, the goal of the trace semantics of these works and of the presented work are analogous, as the trace semantics is used to prove secure compilation results related to the language. Laird [28] presented a fully abstract trace semantics for a functional language with locally declared general references that does not focus on the security aspects of that language. Ghica and Tzevelekos [29] provided a fully abstract trace semantics, with regards to a game operational semantics, of a C-like language that, unlike this work, does not present a protection mechanism. Jeffrey and Rathke [16] provided a fully abstract trace semantics for a core Java-like language that enforces strong encapsulation of objects in packages and of fields in classes. Welsch and Poetzsch-Heffter [30] devised a fully abstract trace-based semantics for class libraries in Java-like languages, focussing on backward compatibility for class libraries instead of security.

Different techniques can be used to capture the behaviour of untyped assembly code, for example denotational semantics and logical relations. If the goal is reasoning about a specific aspect of assembly code, it can be equipped with a denotational semantics capturing precisely that aspect [31]. As PMA operates at the untyped assembly language level, most of the instructions of the language can be seen as modifying a global state (i.e., the memory). The rules concerning global state update [21, 32] could be used to define a denotational semantics for PMA-enhanced assembly. We expect that these results can affect the definition of a denotational semantics for PMA-enhanced assembly language as they have affected the definition of the Tr^{L} semantics. If the goal is reasoning about compiled assembly code, biorthogonality [33] and Kripke logical relations [34] have been used for proving compiler correctness. These powerful techniques, and their evolution in relation transition systems [35] have unfortunately not been used for PMA-enhanced languages. Reasoning about the behaviour of PMA-enhanced untyped assembly code with logical relations remains an open research area.

A different research area studies logics for assembly languages: Hoare logics [15] or separation logics [36]. Jensen *et al.* [36] 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.

PMA, in the form of fine grained, program counter-based memory access control mechanisms, have been implemented in several software [3, 5, 6, 7] and hardware forms [2, 4] and recently by Intel in the SGX processor [1]. From the theoretical point of view, assembly languages extended with these protection mechanisms have been recently studied as target languages for secure compilation schemes [10, 11, 12]. The language and trace semantics of this paper are inspired by those works.

A different protection mechanism that could be employed at the assembly level is a typed assembly language [14]. 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 programs enhanced with PMA. To this extent, it formalised $\mathcal{A}$+I, an assembly language extended with that isolation mechanism. Then, it provided two different trace semantics for $\mathcal{A}$+I: Tr^{S} and Tr^{L}. Tr^{S} can be used to model the behaviour of components that are securely-compiled to $\mathcal{A}$+I and it can be used to simplify proofs of secure compilation to $\mathcal{A}$+I. Tr^{L} investigates the challenges of including readout and writeout operations in the trace characterisation. Moreover, it provided a general proof strategy where both trace 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.

The Tr^{L} semantics relies on a partitioning of unprotected code into a code and a data section to achieve full abstraction. This partitioning is not enforceable in some PMA architectures (e.g., the Intel SGX or Sancus), while in software or hypervisor-based implementation of PMA it can be. As previously mentioned, however, the Tr^{L} semantics captures the behaviour of protected code that reads or writes in a certain area of unprotected memory and jumps to a different area of unprotected memory when returning control to unprotected code. Thus, Tr^{L} can be used to describe the behaviour of protected PMA code that uses readouts and writeouts to exchange parameters with unprotected code. To eliminate the need for the partitioning of unprotected code, we envision that the semantics must accumulate the knowledge of its readouts and writeouts when producing the traces. Addressing this challenge, therefore providing fully abstract trace semantics for protected, arbitrary PMA code, is left for future work.

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.

Marco Patrignani holds a Ph.D. fellowship from the Research Foundation Flanders (FWO). This work has been supported in part by the Intel Labs University Research Office. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

References

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, U. R. Savagaonkar, Innovative instructions and software model for isolated execution, in: HASP '13, ACM, New York, NY, USA, 2013, pp. 10:1–10:1. doi:10.1145/2487726.2488368.
URL http://doi.acm.org/10.1145/2487726.2488368

[2] K. Eldefrawy, A. Francillon, D. Perito, G. Tsudik, SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust, in: NDSS'12, 2012, pp. 1–15.
URL http://www.eurecom.fr/publication/3536

[3] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, H. Isozaki, Flicker: an execution infrastructure for TCB minimization, SIGOPS Oper. Syst. Rev. 42 (4) (2008) 315–328. doi:10.1145/1357010.1352625.
URL http://doi.acm.org/10.1145/1357010.1352625

[4] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens, Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base, in: Proceedings of the 22Nd USENIX Conference on Security, SEC'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 479–494.
URL http://dl.acm.org/citation.cfm?id=2534766.2534808

[5] L. Singaravelu, C. Pu, H. Härtig, C. Helmuth, Reducing TCB complexity for security-sensitive applications: three case studies, SIGOPS Oper. Syst. Rev. 40 (4) (2006) 161–174. doi:10.1145/1218063.1217951.
URL http://doi.acm.org/10.1145/1218063.1217951

[6] R. Strackx, F. Piessens, Fides: Selectively hardening software application components against kernel-level or process-level malware, in: CCS 2012, ACM Press, 2012, pp. 2–13. doi:10.1145/2382196.2382200.
URL http://doi.acm.org/10.1145/2382196.2382200

[7] R. Strackx, F. Piessens, B. Preneel, Efficient isolation of trusted subsystems in embedded systems, in: SecureComm, 2010, pp. 344–361. doi:10.1007/978-3-642-16161-2_20.
URL http://dx.doi.org/10.1007/978-3-642-16161-2_20

[8] A. Ahmed, M. Blume, An equivalence-preserving CPS translation via multi-language semantics, SIGPLAN Not. 46 (9) (2011) 431–444. doi:10.1145/2034574.2034830.
URL http://doi.acm.org/10.1145/2034574.2034830

[9] M. Abadi, Protection in programming-language translations, in: Secure Internet programming, Springer-Verlag, 1999, pp. 19–34.
URL http://dl.acm.org/citation.cfm?id=380171.380174

[10] P. Agten, R. Strackx, B. Jacobs, F. Piessens, Secure compilation to modern processors, in: CSF '12, IEEE, 2012, pp. 171 – 185. doi:10.1109/CSF.2012.12.
URL http://dx.doi.org/10.1109/CSF.2012.12

[11] M. Patrignani, D. Clarke, F. Piessens, Secure Compilation of Object-Oriented Components to Protected Module Architectures, in: (APLAS'13), Vol. 8301 of LNCS, 2013, pp. 176–191. doi:10.1007/978-3-319-03542-0_13.
URL http://dx.doi.org/10.1007/978-3-319-03542-0_13

[12] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, F. Piessens, Secure compilation to protected module architectures, ACM Transactions on Programming Languages and Systems (TOPLAS).

[13] M. Patrignani, D. Clarke, Fully Abstract Trace Semantics of Low-level Isolation Mechanisms, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, 2014, pp. 1562–1569.
URL https://lirias.kuleuven.be/handle/123456789/361235

[14] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, ACM Trans. Program. Lang. Syst. 21 (3) (1999) 527–568. doi:10.1145/319301.319345.
URL http://doi.acm.org/10.1145/319301.319345

[15] A. Saabas, T. Uustalu, A compositional natural semantics and Hoare logic for low-level languages, Electr. Notes Theor. Comput. Sci. 156 (2006) 151–168. doi:10.1016/j.entcs.2005.09.031.
URL http://dx.doi.org/10.1016/j.entcs.2005.09.031

[16] A. Jeffrey, J. Rathke, Java Jr.: fully abstract trace semantics for a core java language, in: ESOP'05, Springer-Verlag, 2005, pp. 423–438. doi:10.1007/978-3-540-31987-0_29.
URL http://dx.doi.org/10.1007/978-3-540-31987-0_29

[17] J. A. Goguen, J. Meseguer, Security policies and security models, in: IEEE Symposium on Security and Privacy, 1982, pp. 11–20.

[18] S. A. Zdancewic, Programming languages for information security, Ph.D. thesis, Cornell University (2002).

[19] P.-L. Curien, Definability and full abstraction, Electron. Notes Theor. Comput. Sci. 172 (2007) 301–310. doi:10.1016/j.entcs.2007.02.011.
URL http://dx.doi.org/10.1016/j.entcs.2007.02.011

[20] G. D. Plotkin, LCF considered as a programming language, Theoretical Computer Science 5 (1977) 223–255. doi:http://dx.doi.org/10.1016/0304-3975(77)90044-5.
URL http://www.sciencedirect.com/science/article/pii/0304397577900445

[21] J. Power, O. Shkaravska, From comodels to coalgebras: State and arrays, Electron. Notes Theor. Comput. Sci. 106 (2004) 297–314. doi:10.1016/j.entcs.2004.02.041.
URL http://dx.doi.org/10.1016/j.entcs.2004.02.041

[22] R. Milner, Fully abstract models of typed *lambda*-calculi, Theor. Comput. Sci. 4 (1) (1977) 1–22. doi:http://dx.doi.org/10.1016/0304-3975(77)90053-6.
URL http://www.sciencedirect.com/science/article/pii/0304397577900536

[23] A. Jeffrey, J. Rathke, Full abstraction for polymorphic pi-calculus, in: FOSSACS'05, Springer-Verlag, 2005, pp. 266–281. doi:10.1007/978-3-540-31982-5_17.
URL http://dx.doi.org/10.1007/978-3-540-31982-5_17

[24] N. Francez, C. Hoare, W. Roever, Semantics of nondeterminism, concurrency and communication, in: J. Winkowski (Ed.), Mathematical Foundations of Computer Science 1978, Vol. 64 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1978, pp. 191–200. doi:10.1007/3-540-08921-7_67.
URL http://dx.doi.org/10.1007/3-540-08921-7_67

[25] D. Sangiorgi, Expressing mobility in process algebras: First-order and higher-order paradigms, PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh (1992).

[26] M. Abadi, G. Plotkin, On protection by layout randomization, in: CSF '10, IEEE, 2010, pp. 337–351. doi:10.1109/CSF.2010.30.
URL http://dx.doi.org/10.1109/CSF.2010.30

[27] R. Jagadeesan, C. Pitcher, J. Rathke, J. Riely, Local memory via layout randomization, in: CSF '11, IEEE, 2011, pp. 161–174. doi:10.1109/CSF.2011.18.
URL http://dx.doi.org/10.1109/CSF.2011.18

[28] J. Laird, A fully abstract trace semantics for general references, in: Automata, Languages and Programming, Vol. 4596 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 667–679. doi:10.1007/978-3-540-73420-8_58.
URL http://dx.doi.org/10.1007/978-3-540-73420-8_58

[29] D. R. Ghica, N. Tzevelekos, A system-level game semantics, Electr. Notes Theor. Comput. Sci. 286 (2012) 191–211. doi:http://dx.doi.org/10.1016/j.entcs.2012.08.013.
URL http://www.sciencedirect.com/science/article/pii/S1571066112000436

[30] Y. Welsch, A. Poetzsch-Heffter, A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries, Science of Computer Programming - (0) (2013) –. doi:http://dx.doi.org/10.1016/j.scico.2013.10.002.
URL http://www.sciencedirect.com/science/article/pii/S0167642313002529

[31] D. A. Schmidt, Denotational Semantics: A Methodology for Language Development, William C. Brown Publishers, Dubuque, IA, USA, 1986.

[32] G. D. Plotkin, J. Power, Notions of computation determine monads, in: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS '02, Springer-Verlag, London, UK, UK, 2002, pp. 342–356.
URL http://dl.acm.org/citation.cfm?id=646794.704856

[33] N. Benton, C.-K. Hur, Biorthogonality, step-indexing and compiler correctness, SIGPLAN Not. 44 (9) (2009) 97–108. doi:10.1145/1631687.1596567.
URL http://doi.acm.org/10.1145/1631687.1596567

[34] C.-K. Hur, D. Dreyer, A kripke logical relation between ml and assembly, SIGPLAN Not. 46 (1) (2011) 133–146. doi:10.1145/1925844.1926402.
URL http://doi.acm.org/10.1145/1925844.1926402

[35] C.-K. Hur, D. Dreyer, G. Neis, V. Vafeiadis, The marriage of bisimulations and kripke logical relations, SIGPLAN Not. 47 (1) (2012) 59–72. doi:10.1145/2103621.2103666.
URL http://doi.acm.org/10.1145/2103621.2103666

[36] J. B. Jensen, N. Benton, A. Kennedy, High-level separation logic for low-level code, SIGPLAN Not. 48 (1) (2013) 301–314. doi:10.1145/2480359.2429105.
URL http://doi.acm.org/10.1145/2480359.2429105

[37] A. Jeffrey, J. Rathke, A fully abstract may testing semantics for concurrent objects, Theor. Comput. Sci. 338 (1-3) (2005) 17–63. doi:10.1016/j.tcs.2004.10.012.
URL http://dx.doi.org/10.1016/j.tcs.2004.10.012

Appendices

A. Properties of the Rewrite Rules of Figure 13

The rewrite rules of Figure 13 are confluent (Lemma 2) and terminating (Lemma 3), thus they are convergent (Theorem 3). This implies that when applied to a prefix $\overline{\omega(a,v)}$, they will always return its unique normal form.

In the following, denote a prefix sequence $\overline{\omega(a,v)}$ with π.

Lemma 2 (Confluence). *The rewrite rules are confluent. For any π, for all π' and π'' such that $\pi \rightsquigarrow \pi'$ and $\pi \rightsquigarrow \pi''$, there exists π''' such that $\pi' \rightsquigarrow^* \pi'''$ and $\pi'' \rightsquigarrow^* \pi'''$.*

Proof. This proof proceeds by induction over the length of $\overline{\omega(a,v)}$.

Base case (length 0 or 1) No reduction rules apply, so the theorem holds.

Inductive case (length > 1) For any combination of the first two actions of the prefix ($\omega_0(a_0, v_0)$ and $\omega_1(a_1, v_1)$) only one rule is applicable, as presented in the case analysis below. Thus, π' and π'' are the same and so both flow into the same π'''.

- $\texttt{read}(a,v)\texttt{read}(a,v)$ only Rule (Read-drop) applies.
- $\texttt{read}(a,v)\texttt{read}(a,w)$ **and** $v \neq w$ firstly, Rule (Constraint-read) ensures that $v = w$, then only Rule (Read-drop) applies.
- $\texttt{read}(a,v)\texttt{read}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Rule (Read-order) applies.
- $\texttt{read}(a,v)\texttt{write}(a,v)$ only Rule (Read-no-write) applies.
- $\texttt{read}(a,v)\texttt{write}(a,w)$ **and** $v \neq w$ no rule applies.
- $\texttt{read}(a,v)\texttt{write}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Rule (RW-order) applies.
- $\texttt{write}(a,v)\texttt{write}(a,v)$ only Rule (Write-drop) applies.
- $\texttt{write}(a,v)\texttt{write}(a,w)$ **and** $v \neq w$ only Rule (Write-drop) applies.
- $\texttt{write}(a,v)\texttt{write}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Rule (Write-order) applies.
- $\texttt{write}(a,v)\texttt{read}(a,v)$ only Rule (Write-no-read) applies.
- $\texttt{write}(a,v)\texttt{read}(a,w)$ **and** $v \neq w$ firstly, Rule (Constraint-read) ensures that $v = w$, then only Rule (Write-no-read) applies.
- $\texttt{write}(a,v)\texttt{read}(b,v)$ **and** $a \neq b$ and $a \neq b]$ in this case if $a < b$ no rule apply, while if $a > b$ only Rule (WR-order) applies.

□

Lemma 3 (Termination). *The rewrite rules are terminating: for any prefix, all possible sequences of application of the rewrite rules to $\overline{\omega(a,v)}$ are finite.*

Proof. The proof proceeds by structural induction on $\overline{\omega(a,v)}$.

Base case: $\overline{\omega(a,v)} = \epsilon$ As no rewrite rules apply here, this case is terminating.

Inductive case: $\overline{\omega(a,v)} = \omega_0(a_0,v_0)\cdots\omega_n(a_n,v_n)$ The inductive hypothesis IH tells us that applying the rewriting rules to a prefix $\omega_1(a_1,v_1)\cdots\omega_n(a_n,v_n)$ of length n is terminating in q steps. Apply Lemma 2 to know that the confluent form of $\omega_1(a_1,v_1)\cdots\omega_n(a_n,v_n)$ is $\omega'_1(a'_1,v'_1)\cdots\omega'_n(a'_n,v'_n)$. The following alternatives arise based on $\omega_0(a_0,v_0)$ and $\omega'_1(a'_1,v'_1)$.

$a_0 < a'_1$. In this case no rewrite rules apply for prefixes of index 0 and 1. Their application terminates in q steps for prefixes 1 onwards as stated in the IH. So this case terminates in q steps.

$a_0 > a'_1$. In this case, ω_0 and ω'_1 are swapped in place by applying one rewriting rule. We can then apply the IH to state that applying the rewrite rules to $\omega_0(a_0,v_0)\omega'_2(a'_2,v'_2)\cdots\omega'_n(a'_n,v'_n)$ is terminating. By applying Lemma 2, we can define the confluence form of that prefix with π. $\omega_1(a_1,v_1)\pi$ is thus terminating since no rewrite rules apply to prefixes of index 0 and 1 while their application terminates in q steps for prefixes 1 onwards as stated in the IH. So this case terminates in $q+1$ steps.

$a_0 = a'_1 = a$. The following cases arise:

$\omega_0(a,v_0) = \texttt{read}(a,v_0)$ and $\omega'_1(a,v'_1) = \texttt{read}(a,v'_1)$. Firstly, Rule (Constraint-read) is applied, thus $v_0 = v'_1$. Then, Rule (Read-drop) drops one of those actions, so we can apply the IH since the prefix is of length n. So this case terminates in $q+2$ steps.

$\omega_0(a,v_0) = \texttt{read}(a,v_0)$ and $\omega'_1(a,v'_1) = \texttt{write}(a,v'_1)$. The following cases arise:

$v_0 = v'_1 = v$. Rule (Read-no-write) is applied, and only the 0-indexed action is kept. We can apply the IH to the prefix $\omega_0(a_0,v_0)\omega'_2(a'_2,v'_2)\cdots\omega'_n(a'_n,v'_n)$ which is of length n. So this case terminates in $q+1$ steps.

$v_0 \neq v'_1$. In this case no rewrite rules apply for prefixes of index 0 and 1. Their application terminates for prefixes 1 onwards as stated in the IH. So this case terminates in q steps.

$\omega_0(a,v_0) = \texttt{write}(a,v_0)$ and $\omega'_1(a,v'_1) = \texttt{read}(a,v'_1)$. Firstly, Rule (Constraint-write) is applied, thus $v_0 = v'_1$. Then, Rule (Write-drop) drops the 1-indexed action, so we can apply the IH to the prefix $\omega_0(a_0,v_0)\omega'_2(a'_2,v'_2)\cdots\omega'_n(a'_n,v'_n)$ since it is of length n. So this case terminates in $q+2$ steps.

$\omega_0(a,v_0) = \texttt{write}(a,v_0)$ and $\omega'_1(a,v'_1) = \texttt{write}(a,v'_1)$. Rule (Write-drop) is applied and only the 1-indexed action is kept, so we can apply the IH and this case holds. So this case terminates in $q+1$ steps.

Since all cases terminate in a finite number of steps, applying the rewrite rules always terminates. □

Theorem 3 (Convergence). *The rewrite rules are convergent, i.e. they are confluent and terminating.*

Proof. By Lemma 2 and Lemma 3. □

B. Proof of Lemma 1

Proof. The proof proceeds by induction on $\overline{\alpha}$ that leads to a case analysis on α. We omit the inductive cases and proceed directly to the case analysis considered for the base case.

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

?-decorated action. This action can either be a call of the form $\texttt{call}\ p(r;f)$ or a return of the form $\texttt{ret}\ p(r;f)$, only the case for the call is presented since the one for the return is analogous.

So: $\Theta_1 \xrightarrow{\texttt{call}\ p(r;f)?}\!\!\!\!\rightarrow \Theta'_1$ and $\Theta_2 \xrightarrow{\texttt{call}\ p(r;f)?}\!\!\!\!\rightarrow \Theta'_2$.

By definition, $\Theta'_1 = \Omega'_1 = (p, r, f, m'_1, s_1)$ and $\Theta'_2 = \Omega'_2 = (p, r, f, m'_2, s_2)$.

The thesis is $\Omega'_1 \mathrel{\mathring{=}} \Omega'_2$, so $(p, r, f, m'_1, s_1) \mathrel{\mathring{=}} (p, r, f, m'_2, s_2)$. Both states need to have equal registers, flags and unprotected memory. The first two points are clear, as registers and flags are set to be the same by the label. What needs to be proven is that $\texttt{m}_{\texttt{ext}}(m'_1, s_1) = \texttt{m}_{\texttt{ext}}(m'_2, s_2)$.

From hypothesis $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, we have that $\texttt{m}_{\texttt{ext}}(m_1, s_1) = \texttt{m}_{\texttt{ext}}(m_2, s_2)$.

Since the action $\texttt{call}\ p(r;f)$ does not touch the unprotected memory, we have that $\texttt{m}_{\texttt{ext}}(m_1, s_1) = \texttt{m}_{\texttt{ext}}(m'_1, s_1)$ and $\texttt{m}_{\texttt{ext}}(m_2, s_2) = \texttt{m}_{\texttt{ext}}(m'_2, s_2)$.

By transitivity we obtain that $\texttt{m}_{\texttt{ext}}(m'_1, s_1) = \texttt{m}_{\texttt{ext}}(m'_2, s_2)$ holds, so this case holds as well.

!-decorated action. Here, $\delta!$ is in the form $\delta_1 \cdots \delta_n \gamma!$. The action $\gamma!$ can either be a call of the form $\texttt{call}\ p(r;f)$ or a return of the form $\texttt{ret}\ p(r;f)$; only the case for the call is presented since the one for the return is analogous.

So: $\Theta_1 \xrightarrow{\delta_1\cdots\delta_n \texttt{call}\ p(r;f)!}\!\!\!\!\rightarrow \Theta'_1$ and $\Theta_2 \xrightarrow{\delta_1\cdots\delta_n \texttt{call}\ p(r;f)!}\!\!\!\!\rightarrow \Theta'_2$.

$\Theta_1 = \Omega_1 = (p, r, f, m_1, s_1)$ and $\Theta_2 = \Omega_2 = (p, r, f, m_2, s_2)$.

By definition, $\Theta'_1 = (\mathsf{unk}, m'_1, s_1)$ and $\Theta'_2 = (\mathsf{unk}, m'_2, s_2)$.

We can reconstruct Ω'_1 by applying the following hypotheses: $\Theta_1 = \|\Omega_1\|$ and $\Omega_1 \rightarrow^* \Omega'_1$ and $\Theta'_1 = \|\Omega'_1\|$. Analogously, we can reconstruct Ω'_2.

So, $\Omega'_1 = (p, r, f, m'_1, s_1)$ and $\Omega'_2 = (p, r, f, m'_2, s_2)$. The thesis is $\Omega'_1 \mathrel{\mathring{=}} \Omega'_2$, so $(p, r, f, m'_1, s_1) \mathrel{\mathring{=}} (p, r, f, m'_2, s_2)$. Both states need to have equal registers, flags and unprotected memory. The first two points are clear, as registers and flags are set to be the same by the label. What needs to be proven is that $\texttt{m}_{\texttt{ext}}(m'_1, s_1) = \texttt{m}_{\texttt{ext}}(m'_2, s_2)$.

From hypothesis $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, we have that $\texttt{m}_{\texttt{ext}}(m_1, s_1) = \texttt{m}_{\texttt{ext}}(m_2, s_2)$.

What needs to be considered are the prefixes $\delta_1 \cdots \delta_n$, which can be either readouts or writeouts: The proof now proceeds by induction on n.

Base case, $n = 0$ Trivial, since we have that $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m'_1, s_1)$ and $\mathtt{m_{ext}}(m_2, s_2) = \mathtt{m_{ext}}(m'_2, s_2)$.

This, combined with so the hypothesis $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m_2, s_2)$, fulfils this case.

Inductive case, $n = k + 1$ Consider $\delta_1 \cdots \delta_k \delta'$, the inductive hypothesis states that up to δ_k, external memories are the same. Indicate the memory up to the kth step with m_k, the inductive hypothesis states that $\mathtt{m_{ext}}(m_1^k, s_1) = \mathtt{m_{ext}}(m_2^k, s_2)$.

Two cases arise for δ', one for the readout and one for the writeout.

$\delta' = \mathtt{read}(a, v)$ Readouts do not change the external memory, so apply the inductive hypothesis and this case holds.

$\delta' = \mathtt{write}(a, v)$ Writeouts *do* change the external memory, so $\mathtt{m_{ext}}(m'_1, s_1) = \mathtt{m_{ext}}(m_1^k, s_1)[a \mapsto v]$ and $\mathtt{m_{ext}}(m'_2, s_2) = \mathtt{m_{ext}}(m_2^k, s_2)[a \mapsto v]$.
Since the initially-equal memories $\mathtt{m_{ext}}(m_1^k, s_1)$ and $\mathtt{m_{ext}}(m_2^k, s_2)$ are changed in the same way, the thesis holds in this case as well.

□

Having covered all the cases, the theorem holds.

The proof of Lemma 1 for Tr^{S} is included in the proof for Tr^{L} with very small syntactical changes since the labels of Tr^{S} are a subset of the labels of Tr^{L}.

C. Proof of Theorem 1

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 $(m, s) \frown \mathbb{M}$. Given an instruction $i \in \mathcal{I}$, identify a transition triggered by the execution of that instruction as $\xrightarrow{i}$.

Proof. 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. Direction $\Rightarrow$, so the thesis is $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \Rightarrow \mathbb{M}[P_2]\Uparrow$.
Apply the definition of contextual equivalence (Definition 4) and the thesis becomes $\forall \mathbb{M}.\ \mathbb{M}[\Omega_0(P_1)]\Uparrow \Rightarrow \mathbb{M}[\Omega_0(P_2)]\Uparrow$.
Let $\Omega_1 = \mathbb{M}[\Omega_0(P_1)]$ and $\Omega_2 = \mathbb{M}[\Omega_0(P_2)]$.
The thesis is $\forall \mathbb{M}.\ \forall n \in \mathbb{N}. \exists \Omega'_1.\ \Omega_1 \to^n \Omega'_1 \Rightarrow \forall m \in \mathbb{N}. \exists \Omega'_2.\ \Omega_2 \to^m \Omega'_2$.
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: $\exists \Omega'_2. \Omega_2 \to^{h+1} \Omega'_2$.

   The inductive hypothesis (IH) is: $\forall \mathbb{M}.\ \forall n \in \mathbb{N}. \exists \Omega'_1.\ \mathbb{M}[\Omega_0(P_1)] \to^n \Omega'_1 \Rightarrow \mathbb{M}[\Omega_0(P_2)] \to^h \Omega_2^h$.
   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: both p_1 and p_2 are in the protected partition (1a) or in the unprotected one (1b).

(a) $s_1 \vdash \texttt{protected}(p_1)$ and $s_2 \vdash \texttt{protected}(p_2)$.
This case relies on the trace semantics rules to say that either P_1 and P_2 produce the same label, or they diverge; in both cases there is a corresponding reduction in the operational semantics. There are two cases: either both programs will perform another action $\delta!$ (1(a)i), or not (1(a)ii).

i. $\exists \delta!.\ \|\Omega_1^h\| \xRightarrow{\delta!} \|\Omega_1^{h'}\|$.
By hypothesis $P_1 \simeq_{\mathrm{T}} P_2$, $\|\Omega_2^h\| \xRightarrow{\delta!} \|\Omega_2^{h'}\|$.
This, in conjunction with IH, implies the thesis $\Omega_2 \rightarrow^{h+1} \Omega_2'$.
Note that $\delta!$ cannot be a $\surd$, as this violates the hypothesis $\forall n \in \mathbb{N}.\mathbb{M}[\Omega_0](P_1)] \rightarrow^n \Omega_1'$.

ii. $\nexists \delta!.\ \|\Omega_1^h\| \xRightarrow{\delta!} \|\Omega_1^{h'}\|$.
Let $\Omega_1^h = (p_1, r_1, f_1, m + m_1, s_1)$ and $\Omega_2^h = (p_2, r_2, f_2, m + m_2, s_2)$.
In this case, Ω_2^h does not terminate, since it does not produce a $\surd$ label, so it computes, generating τ actions.
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 Ω_2 can always make a step for instruction i_2, so $\Omega_2 \rightarrow^h \Omega_2^h \xrightarrow{i_2} \Omega_2'$.

(b) $s_1 \vdash \texttt{unprotected}(p_1)$ and $s_2 \vdash \texttt{unprotected}(p_2)$.
In this case we need to prove that, for whatever computation was done so far, P_1 and P_2 end up with a program counter in the same location in their unprotected memory. We rely on Lemma 1 to state that, if P_1 and P_2 have jumped inside the protected partition and then back outside, their unprotected memory is still the same.
By IH $\exists l \leq h.\ \mathbb{M}[\Omega_0(P_1)] \rightarrow^l \Omega_1^l$ and $\|\Omega_0(P_1)\| \xRightarrow{\overline{\alpha}\alpha} \|\Omega_1^l\|$.
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, $\|\Omega_0(P_2)\| \xRightarrow{\overline{\alpha}\alpha} \|\Omega_2^l\|$.
By Lemma 1, $\Omega_1^l = (p^l, r^l, f^l, \mathbb{M}^l + m_1, s_1)$ and $\Omega_2^l = (p^l, r^l, f^l, \mathbb{M}^l + m_2, s_2)$ (if α does not exist and $\overline{\alpha}$ is the empty list, there is no need to apply Lemma 1).
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*.

□

D. Proof of Theorem 2

Completeness is equivalently stated as: $P_1 \not\eqsim_{\mathrm{T}} P_2 \Rightarrow P_1 \not\eqsim P_2$.

Proof Sketch. 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 [10, 11, 37]. The algorithm produces unprotected code that performs all ?-decorated actions in the traces and then terminates with result 1 or diverges, based on the program it is interacting with after the different !-decorated action.

The two different traces are generated as follows. Since $P_1 \not\eqsim_{\mathrm{T}} P_2$, 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 $\overline{\alpha_s}$ (the common prefix) and $\overline{\alpha_d}$ such that $\overline{\alpha}{=}\overline{\alpha_s\alpha_d}$, 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 $\overline{\alpha_1} = \overline{\alpha_s\alpha_d}$ and $\overline{\alpha_2} = \overline{\alpha_s}\overline{\alpha'_d}$.

Algorithm description. Assume that there is always enough memory to store the algorithm; call the algorithm P. In P there must be four functions in order to set the flags to the all combinations. These function are of the form:

- store $\mathtt{r_1}$ and $\mathtt{r_2}$ in unprotected memory;
- set $\mathtt{r_1}$ and $\mathtt{r_2}$ to the right values that set the flag to the desired combination (e.g, for $\mathsf{SF}{=}0$; $\mathsf{ZF}{=}1$, set $\mathtt{r_1}{=}1$ and $\mathtt{r_2}{=}1$);
- execute $\mathtt{cmp}\ \mathtt{r_1}\ \mathtt{r_2}$;
- restore $\mathtt{r_1}$ and $\mathtt{r_2}$ 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 stack c is set to p_0 – the initial value of the program counter.

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 a P that generates those traces. Thus, at location c, the algorithm writes code depending on what action is being considered.

`call` p (r, f)? Firstly, the algorithm writes a `call` to the function that sets the flags to f. Then the top of stack c is incremented by 1. Then, all twelve registers are set to the values of r, thus given that the values of register i in r is v_i, the following instruction is written: `movi` $\mathtt{r_i}$ v_i for all $i = 0..11$. 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 (or more, if `add` instructions are used). Then based on which register contains the value p that is where the call is directed, instruction `call` $\mathtt{r_p}$ is written. Then the top of c is incremented by 1.

`ret` p (r, f)? As in the previous step, the algorithm sets flags and registers to the desired values. Then instruction `ret` is written. Then the top of c is incremented by 1.

!-decorated actions. These actions are generated by protected code.

callbacks. If both actions are of the form `call` p (r, f)?, then p is pushed on top of the stack c.

returns. If both actions are of the form `ret` p (r, f)?, then the top of the stack c is popped.

writeouts. The algorithm adds no code to P. In this case we are assured that control will jump back to the code because protected code does not write in an executable part of the unprotected memory.

readouts. If both actions are of the form $\mathtt{read}(a, v)$, then the following instructions are written before other code at address c: `movi` $\mathtt{r_0}$ a; `movi` $\mathtt{r_1}$ v; `movs` $\mathtt{r_1}$ $\mathtt{r_0}$. These instructions ensure that address a contains value v.

To avoid clashes with writeouts and readouts, assume traces are inspected beforehand and the location where P is stored does not clash with the addresses of those operations.

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

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 diverging code at the address indicated by c.

different actions. Assume that $\alpha_1 = \mathtt{ret}\, p\, (r, f)!$ and $\alpha_2 = \mathtt{call}\, 10\, (r, f)!$. Then the algorithm writes instructions `movi` $\mathtt{r_0}$ `1`; `halt` at c and diverging code at address 10.

Assume that $\alpha_1 = \mathtt{write}(a, v).\delta!$ and $\alpha_2 = \delta!$. In this case, before executing the protected code that generates that trace, the algorithm writes value u, different from v, at address a. Then, after the protected code has performed $\delta!$, the value in a is read and compared

to u. If they are the same, then instructions `movi` $\mathtt{r}_0$ `1`; `halt` are written at c, otherwise diverging code is written there.

Other cases are similar.

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

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

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

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

Concerning readouts, they are included in the traces only if they are followed by different actions. Readouts that are not followed by different actions satisfy the non-interference judgment $\mathsf{NI}(\cdot)$, they are non-interfering. On the other hand, readouts that are followed by different actions do not satisfy that judgment, they are interfering. Function $\mathsf{StripNI}(\cdot)$ (Figure 16), which is used to accumulate labels in Rule Trace-l-action, ensures that all non-interfering readout labels are eliminated from traces. So, readouts that appear in traces are interfering and thus followed by different actions. It is that action that determines what the code generated by the algorithm is, no action is undertaken for readouts. □