Fully Abstract Trace Semantics for Low-level Isolation Mechanisms

Marco Patrignani∗ Dave Clarke
iMinds-DistrINet, Dept. Computer Science, KU Leuven
{first.last}@cs.kuleuven.be

ABSTRACT

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

Categories and Subject Descriptors
D.2.4 [Software/Program Verification]: Formal models

1. INTRODUCTION

Many processors provide isolation mechanisms as software security building blocks. These are used to withstand low-level attackers who, through injected assembly code, can read the whole memory space and access secrets in memory, violate integrity constraints and so on. With isolation mechanisms, attackers cannot directly violate security properties of isolated software since the isolated memory is not accessible to them. Examples of these protection mechanisms are fine grained, program counter-based memory access control mechanisms [4, 8, 9, 12, 17, 18, 19], which enforce security properties at process or lower levels (Ring-1). With this mechanism, the software to be secured is placed in a protected memory partition that shields it from the surrounding, potentially malicious code. The malicious code cannot read nor write the protected memory; it can only jump to specific addresses in protected memory to call functions of the protected code. This protection mechanism makes software more resilient against low-level attackers. However, this does not prevent an attacker from interacting with the protected code and violating security properties.

To enable reasoning about the behaviour of isolated software, this paper firstly introduces an assembly language enhanced with a fine grained, program counter-based memory access control. This protection mechanism was chosen due to its application in secure compilation [2, 14] and to its recent industrial implementation: the Intel SGX [9]. Then, this paper explores the design space of trace semantics and presents two different fully abstract [15] trace semantics for the protected assembly code. The paper provides a general proof strategy where full abstraction of the trace semantics can be proven simply. To the best of the authors’ knowledge, this is the first such result for an assembly language extended with the said protection mechanism. Moreover, the results seem applicable to other isolation mechanisms.

The fully abstract trace semantics provides a number of benefits. Trace semantics uses simple abstractions to represent the behaviour of protected assembly code, unburdened by low-level details, at the maximum degree of precision. So, dually, it models the behaviour of an attacker to the protected code, since it captures precisely the capabilities of that attacker. Secondly, without the trace semantics, the behaviour of protected assembly code (and thus of an attacker) is given in terms of low-level contexts. While being precise, low-level contexts are notoriously difficult to reason about, since they offer no inductive nor coinductive structure. The fully abstract trace semantics allows low-level contexts to be disregarded, since low-level contexts and traces are proven to be equally precise. Finally, the field of secure compilation benefits from the trace semantics. Recently, Agten et al. [2] and Patrignani et al. [14] presented secure compilers to assembly code enhanced with the aforementioned protection mechanism. Both results assume that the assembly language has a fully abstract trace semantics, thus the results of this paper strengthen those secure compilation claims.

In summary, the contributions of this paper are:

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

Limitations of this work are twofold. Firstly, the trace semantics cannot express side-channel attacks. Secondly, the
formalisation does not consider details of the architecture such as caches; yet this is a often assumed [2, 11, 14, 16].

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

2. BACKGROUND NOTIONS

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

2.1 The Protection Mechanism, Informally

The assembly language is enhanced with an isolation mechanism: a fine-grained, program counter-based memory access control mechanism [4, 8, 12, 17, 18, 19]. We review this mechanism from the work of Strackx and Piessens [18]. However, the techniques presented in this paper can be easily adapted to reasoning about other isolation mechanisms [4, 8, 17]. The protection mechanism provides a secure environment for running code that must be protected from the environment for running code that must be protected from the code it interacts with. This mechanism assumes that the memory is logically divided into a protected and an unprotected partition. The protected partition is further divided into a code and a data section. The code section contains a variable number of entry points: the only addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected partition. For the sake of simplicity, code in protected memory cannot read the unprotected one; as discussed in Section 2.2 below. Based on the location of the program counter, instructions that violate the access control policy are ruled out. The table below summarises the access control model enforced by the protection mechanism.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protected</td>
<td>Unprotected</td>
</tr>
<tr>
<td>Entry Point</td>
<td>Code</td>
</tr>
<tr>
<td>Protected</td>
<td>r x</td>
</tr>
<tr>
<td>Unprotected</td>
<td>x</td>
</tr>
</tbody>
</table>

2.2 Devising Fully Abstract Trace Semantics

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

**Example 1.**

```
1 100 sub r0 r1
2 101 movi r3 106
3 102 j1 r3
4 103 movi r3 10
5 104 movs r5 r5
6 105 call r2
7 106 movi r1; 41
8 107 movi r0 0
9 108 ret

1 100 sub r0 r1
2 101 movi r3 106
3 102 j1 r3
4 103 movi r3 10
5 104 movs r5 r5
6 105 call r2
7 106 movi r1; 42
8 107 movi r0 0
9 108 ret
```

Both P_L and P_R assign to r3 the result of r3 − r1 (line 1). If the result of the operation is not less than 0 (line 3), they respectively write the contents of r4 and r5 at the unprotected address 10 (lines 4,5) and call a function whose address is stored in r3 (line 6). Otherwise, they assign different values to r11 (line 7) and return 0 (lines 8,9).

In order to describe the behaviour of the protected code of Example 1, this paper defines a traces semantics for it. The trace semantics gives a simple characterisation of the behaviour of that code as a set of sequences of labels. In this paper, trace semantics are devised to capture the execution of a protected program, which is a program allocated in the protected memory partition. The execution of a program is the evaluation of a sequence of instructions; in order to abstract over instructions, not all of them generate a label in a trace. Labels are generated only by call and ret instructions. If they are executed by unprotected code, they are named calls and returnbacks, if they are executed by protected code they are named callsbacks and returns [2, 14].

Following is the syntax of labels.

```
L ::= a | τ | a ::= g? | g! g ::= call p(τ) | 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, τ is a list of the contents of all registers and v indicates the contents of register r0.

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

```
\tau_1 = call 100(1,2,\ldots)\tau \cdot ret 0!
\tau_2 = call 100(2,1,40,\ldots)\tau \cdot call 40(\ldots)!
```

Traces \tau_1 and \tau_2 are generated by both P_L and P_R. Trace \tau_1 does not capture the different value stored in r1 (line 7), which, even if it is not the returned value of the function, still constitutes an observable difference between P_L and P_R. Trace \tau_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 \tau_1 and \tau_2 do not capture the observable differences between P_L and P_R, the trace semantics fails to be fully abstract.

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

3. LANGUAGE FORMALISATION

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

3.1 Syntax
The language is run on an architecture that models a Von Neumann machine consisting of a program counter \( p \), a register file \( r \), a flags register \( f \) and memory space \( m \). The program counter indicates the address of the instruction that is executed. The register file contains 12 general purpose registers \( R_k \) to \( R_{11} \) and a stack pointer register \( SP \), which contains the address of the top of the current call stack. The flags register contains a zero flag \( ZF \) and a sign flag \( SF \), which are set or cleared by arithmetic instructions and are used by branching instructions. For the sake of simplicity, assume the architecture targeted by the language works with \( \ell \) bit-long words, where \( \ell \) is a power of 2. This allows the formalisation presented to scale to architectures with words of different sizes.

\[
\begin{align*}
\text{Words} & \quad w ::= i \in \mathcal{I} \\
\text{Memories} & \quad m ::= \emptyset \\
\text{Empty word} & \quad 0 ::= 0^\ell \\
\text{Numbers} & \quad m; a \mapsto w \\
\text{Addresses} & \quad a \in 0..2^\ell - 1 \\
\text{Values} & \quad v \in \mathcal{V} \\
\text{Memory descriptors} & \quad s ::= (a_b, n_c, n_d, n)
\end{align*}
\]

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

\[
\begin{align*}
\text{movl} & \quad r_2 r_3 \\
\text{movs} & \quad r_2 r_3 \\
\text{mov} & \quad r_2 k \\
\text{add} & \quad r_4 r_3 \\
\text{sub} & \quad r_4 r_3 \\
\text{cmp} & \quad r_1 r_2 \\
\text{jmp} & \quad r_1 \\
\text{je} & \quad r_1 \\
\text{jl} & \quad r_1 \\
\text{call} & \quad r_1 \\
\text{ret} & \quad r_1 \\
\text{halt} & \quad r_1
\end{align*}
\]

Fig. 2: Instruction set \( \mathcal{I} \).

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

3.2 Operational Semantics

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

Fig. 3 defines the access control enforcement rules informally presented in Section 2.1. Read judgments \( s \vdash \text{predicate}(a, b, \cdots) \) as: “according to \( s \), predicate holds for addresses \( a, b, \cdots \).” Assume \( s \equiv (a_b, n_c, n_d, n) \).

**Formal Definition**: The semantics is a map from memory descriptors \( s \) and program points \( p \) to the result of the execution at that point. The semantics is defined by a set of rules that capture the behavior of the language.

- **Aux-internal**:\( \vdash \text{data}(p, a) \)\( \vdash \text{writeAllowed}(p, a) \)
- **Aux-entry**:\( \vdash \text{data}(p, a) \)\( \vdash \text{writeAllowed}(p, a) \)
- **Aux-external**:\( \vdash \text{data}(p, a) \)\( \vdash \text{writeAllowed}(p, a) \)
- **Aux-unprotected1**:\( \vdash \text{data}(p, a) \)\( \vdash \text{writeAllowed}(p, a) \)
- **Aux-unprotected2**:\( \vdash \text{data}(p, a) \)\( \vdash \text{writeAllowed}(p, a) \)

Figure 3: Access control enforcement rules.

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

In the semantics there are two call stacks, one for the protected code, called the secure stack, and one for the unprotected code, called the insecure stack. Each stack is preceded by a word containing the location of the current head of the stack, let \( SP_{\text{sec}} \) and \( SP_{\text{ext}} \) indicate the address of the secure and insecure stack respectively. When the current
Given a memory descriptor \( s = (a_0, n_c, n_d, n) \), the secure stack starts at the beginning of the protected data section and the insecure stack starts at the end of the unprotected memory partition. Thus \( \text{SP}_{\text{sec}} = (a_0 + n_c) \) and, initially, \( \text{SP}_{\text{sec}} \mapsto (a_0 + n_c + 1) \); analogously, \( \text{SP}_{\text{ext}} = (a_b + n_c + n_d) \) and, initially, \( \text{SP}_{\text{ext}} \mapsto (a_b + n_c + n_d + 1) \). Call and return instructions are responsible of setting the SP register to the correct address when crossing boundaries between protected and unprotected memory. The value of the program counter is pushed onto the stack by a call instruction (the stack grows down), while a \text{ret} instruction pops one address from the top of the stack and jumps to that location. Updating the stack pointer SP is performed by the auxiliary function \text{setStack} (Fig. 4). In the rules, notation \( m[a \mapsto w] \) indicates that memory \( m \) is updated to a new one that is equal to \( w \) except that the value stored at address \( a \) is \( w \). Notation \( r[R \mapsto w] \) indicates that the register file \( r \) is updated to a new one that is equal to \( w \) except that the value stored in register \( R \) is \( w \). Notation \( r(R) \) indicates the value contained in register \( R \) in register file \( r \). Given a jump between \( d \), rules for \( \xrightarrow{d} \) are obtained by replacing an \text{intJump} assumption with an \text{extJump} one. Let \( m(p) = \text{inst} \) denote that \text{inst} is the word allocated in \( m(p) \), where \( \text{inst} \in \mathcal{I} \). To capture termination, the program counter is set to 0 whenever the \text{halt} instruction is encountered. This way, no progress can be made, as \( m(-1) \) does not return a valid instruction: the program is in a stuck state.

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

The operational semantics of the language is a binary relation over states \( \Omega \xrightarrow{\cdot} \Omega \) defined by the rules of Fig. 6. Rules \text{Eval-callback} and \text{Eval-returnback} ensure that the adversary does to be followed after a callback is stored in the secure stack and that the address returnback entry point \( A_0 \) is pushed on the insecure stack. Thus the unprotected code always jumps to the returnback entry point when returning from a callback. Code located at the returnback entry point must contain a \text{ret} instruction in order to correctly resume the execution. Rule \text{Eval-moves-out} ensures that values written in unprotected memory are not instructions.

The transitive closure of relation \( \mapsto \) is indicated with \( \xrightarrow{\ast} \). A state \( \Omega \) performing \( n \) reduction steps is indicated as \( \Omega \xrightarrow{n} \Omega' \). The evaluation of program \( P \) is a sequence of steps that takes the initial state of \( P \) to another state.

**Definition 3 (Initial state).** The initial state of a program \( (m, s) \), denoted as \( \Omega_0(m, s) \), is the state \( (p_0, r_0, f_0, m, s) \), where \( s = (a_0, n_c, n_d, n) \), \( p_0 = (a_0 + n_c + n_d + 2) \), \( r_0 = \text{SP} \mapsto m(\text{SP}_{\text{sec}}); r_0 \mapsto 0 \ldots 0.11 \), and \( f_0 = \text{ZF} \mapsto 0; \text{SF} \mapsto 0 \).

The evaluation of \( P \) terminates if \( \Omega_0(P) \xrightarrow{\ast} \Omega^1 \); the result is stored in \( r_0 \). If the evaluation of program \( P \) does not
terminate, $P$ diverges. A program $P$ diverges, denoted as $P_{\text{div}}$, if it executes an unbounded number of reduction steps. Formally: $P_{\text{div}}$ if $\forall n \in \mathbb{N}, \exists \Omega' : \Omega(P) \to^{*} \Omega'$.

### 3.3 Contextual Equivalence

Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [15]. This notion relies on the concept of contexts, which is introduced before presenting the equivalence itself.

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

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 $\forall \mathcal{M} : P_1 \bowtie_{\mathcal{M}} \mathcal{M} \wedge [\mathcal{M}[P_1]] \leftrightarrow P_2 \bowtie_{\mathcal{M}} \mathcal{M} \wedge [\mathcal{M}[P_2]]$.

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

**Example 2** (Contextually equivalent programs). The following $P_L$ and $P_R$ write the values of $r_1$ and $r_2$ respectively at the protected address 150 (line 2) and then return 0 (line 3). Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.

```
101 movi r0 150
102 movs r0 r1
103 ret
104 movi r0 0
105 ret
```

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

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

### 4. Trace Semantics

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

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

**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 \((\text{unknown}, m, s)\).

### 4.1 \( \text{Tr}^\dagger \): Expressive Labels

Below are the labels exhibited by the \( \text{Tr}^\dagger \) traces semantics.

\[ \begin{align*}
\Lambda &::= a | \tau_i \\
\alpha &::= \sqrt{\gamma} | \delta
\end{align*} \]

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

Fig. 7 presents the rules defining the relation \( \Theta \xrightarrow{\pi} \Theta' \), which describe when a state \( \Theta \) generates trace \( \pi \) and results in state \( \Theta' \). The \( \text{Tr}^\dagger \) traces of a program \( P \) is defined as follows: \( \text{Tr}^\dagger(P) = \{ \pi \mid \Theta \text{ is a trace of } P \} \).

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

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

The following \( P_L \) and \( P_R \) set \( r_0 \) to 20 and 10 respectively (line 1), then write the instruction \( \text{jmp } r_0 \) at address 20 and 10 respectively (line 2). Finally, they jump to the instruction they just wrote (line 3).

\[
\begin{align*}
\text{1 & 100 movi } r_0 & 20 \\
\text{2 & 101 movi } r_0 & "\text{jmp } r_0" \\
\text{3 & 102 call } r_0
\end{align*}
\]

When \( r_0 \) is set to 20 (resp. 10), the instruction \( \text{jmp } r_0 \) written at address 20 (resp. 10) will diverge. Thus, \( P_L \) and \( P_R \) are contextually equivalent, since no context can differentiate between them. However, \( P_L \) and \( P_R \) are trace inequivalent, since the following is a trace of \( P_L \) and not of \( P_R \) (\( \cdots \)call 100? \cdot write(20, “\text{jmp } r_0”); \( \cdots \)call 20!

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

### 4.2 \( \text{Tr}^\ddagger \): Changes to the Semantics

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

Following are the labels of \( \text{Tr}^\ddagger \); we drop the greek letter notation and use latin letters in order to immediately differentiate between \( \text{Tr}^\dagger \) and \( \text{Tr}^\ddagger \). Flags do not appear in traces because they are always set to 0, as are all registers but \( R_0 \) in case of a return. Write-outs are prohibited, so there are no labels that capture them.

\[
\begin{align*}
\text{L} &::= a | \tau_i \\
\alpha &::= \sqrt{\gamma} | \delta
\end{align*} \]

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

---

**Figure 7: Rules of the \( \text{Tr}^\dagger \) trace semantics.**
5. FULL ABSTRACTION THEOREM

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

A fully abstract trace semantics is both sound and complete with respect to the operational semantics. Soundness states that the trace semantics captures all details of the operational semantics. Thus, for all contexts, two trace equivalent programs cannot be told apart. The universal quantification over contexts makes this more difficult to prove. Completeness states that abstractions of the trace semantics are preserved in the operational semantics. This is achieved by proving the contrapositive of the completeness statement. For this, the proof consists of devising an algorithm that constructs a program, a “witness”, that tells two trace inequivalent programs apart [2, 6, 14]. Since the language is low-level and untyped, this is not particularly complicated.

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

**Theorem 1 (Soundness).** $P_1 \simeq_P P_2 \Rightarrow P_1 \simeq_P P_2$.

Call the interface of a state its registers, flags and unprotected memory. Two states $\Omega_1$ and $\Omega_2$ have the same interface, denoted as $\Omega_1 \simeq_2 \Omega_2$, if they have the same registers, flags and unprotected memory. Formally, $\Omega_1 \simeq_2 \Omega_2$ if $\Omega_1 = (p_1, r, f, m_1, s_1)$ and $\Omega_2 = (p_2, r, f, m_2, s_2)$ and $\mathbb{m}_{usr}(m_1, s_1) = \mathbb{m}_{usr}(m_2, s_2)$. Given $\Omega = (p, r, f, m, s)$, redefine $[\Omega]$ to be the state $\Theta = (p, r, f, \mathbb{m}_{usr}(m, s), s)$ if $s \not\vdash \text{protected}(p)$ and (unknown, $m$, $s$) otherwise.

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

**Lemma 1 (Interface Preservation).** If $\Theta_1 \sim \Theta_2$ and $\Theta_1 = [\Omega_1]$ and $\Omega_1 \rightarrow \Omega_1'$ and $\Theta_2 = [\Omega_2]$ then $\Omega_2 \rightarrow \Omega_2'$ and $\Theta_2 = [\Omega_2']$ and $\Omega_1 \simeq_2 \Omega_2$.

**Theorem 2 (Completeness).** $P_1 \simeq_P P_2 \Rightarrow P_1 \simeq_P P_2$.

Completeness is equivalently stated as: $P_1 \not\simeq_P P_2 \Rightarrow P_1 \not\simeq_P P_2$. This is proven by devising an algorithm that takes as input two different traces $\Sigma_1$ and $\Sigma_2$ and the two programs $P_1$ and $P_2$ generating them and outputs a program $P$ that interacts with $P_1$ and $P_2$ and is able to differentiate between them. The two different traces are generated as follows. Since $P_1 \not\simeq_P P_2$, we have that $\text{Tr}(P_1) \not= \text{Tr}(P_2)$, thus there exists a trace $\sigma$ that belongs to either only $\text{Tr}(P_1)$ or only $\text{Tr}(P_2)$. Assume wlog that $\sigma \in \text{Tr}(P_1)$. The trace $\sigma$ can be split in two parts $\sigma_1$ and $\sigma_2$ such that $\sigma_1 \sim \sigma_2$, and so that there exists a trace $\sigma' \in \text{Tr}(P_2)$ that can be split in two parts $\sigma_1'$ and $\sigma_2'$ such that $\sigma' \sim \sigma_2'$ and $\sigma_1' \not\sim \sigma_1''$. Trace $\sigma'$ always exists, it could be an empty trace, it could be composed by an empty $\sigma_1''$ and, possibly, by an empty $\sigma_2'$. The traces input for the algorithm are $\sigma_1 = \sigma_1 \sigma_2$ and $\sigma_2 = \sigma_2 \sigma_1'$. The interested reader is referred to the companion technical report for proofs [13]. This general proof strategy is presented for both Theorem 1 and Theorem 2. The generalised approach is tailored to each semantics only in the relatively simple proof of Lemma 1. Since Theorem 1 and 2 hold for both $\mathcal{T}^s$ and $\mathcal{T}^r$, both semantics are fully abstract w.r.t. the corresponding operational semantics.

5.1 Benefits of the Trace Semantics

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

Without the trace semantics, the capabilities of an attacker towards protected code are expressed as sequences of instructions, which can be difficult to reason about. With the trace semantics, the capabilities of that attacker are captured via the simple notion of traces. The full abstraction property ensures that traces express precisely all the capabilities of an attacker, without leaving any out.
The second benefit is in the field of secure compilation. Given two programs \( C_1 \) and \( C_2 \) in a language and indicate their compilation to an assembly language with \( C_1^A \) and \( C_2^A \) respectively. Proving the compilation scheme secure is formally stated as \( C_1 \simeq C_2 \iff C_1^A \simeq C_2^A \) [1]. The more complex direction of this proof is \( C_1 \simeq C_2 \Rightarrow C_1^A \simeq C_2^A \), but it can be simplified by adopting a fully abstract trace semantics for the assembly language. That proof is simpler in this way than by adopting the notion of contextual equivalence and performing induction on all possible low-level contexts. The complex direction becomes \( C_1 \simeq C_2 \Rightarrow C_1^A \simeq C_2^A \), whose contrapositive is \( C_1^A \not\simeq C_2^A \Rightarrow C_1 \not\simeq C_2 \). This can be proven similarly to Theorem 2, as Agten et al. [2] and Patrignani et al. [14] did. Both implicitly use the second trace semantics where the compiler adds code that enforces the changes to the operational semantics.

6. RELATED WORK

Full abstraction has been largely studied as a way to state the correctness of a denotational semantics with respect to an operational one [15]. This technique has been adopted for different programming languages paradigms, such as the \( \lambda \)-calculus [10] and the \( \pi \)-calculus [5].

Fine grained, program counter-based memory access control mechanisms have been implemented in several software [8, 17, 18, 19] and hardware forms [4, 12] and recently in the Intel SGX processor [9]. From the theoretical point of view, assembly languages extended with these protection mechanisms have been recently studied as target languages for secure compilation schemes [2, 14]. The language and trace semantics of this paper are inspired by those works. Besides elaborating on details of their formalisation, this work provides the first proof of full abstraction of the trace semantics.

A different research area studies logics for assembly languages: Hoare logics [16] or separation logics [7]. Jensen et al. [7] present a summary of the most recent advances in the latter. That research area focusses on providing reasoning facilities for assembly code, while this paper focusses on reasoning on the security of assembly code.

A different protection mechanism that could be employed at the assembly level is a typed assembly language [11]. To the best of the authors’ knowledge, no fully abstract trace semantics has been provided for such languages.

7. CONCLUSION AND FUTURE WORK

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

Providing a fully abstract trace semantics for a machine with multiple instances of an isolation mechanism or with multiple cores seems 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 reviewers for useful feedback on an earlier draft.

8. REFERENCES


