Computer Languages, Systems & Structures 42 (2015) 22–45

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Fully abstract trace semantics for protected module architectures

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

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

ARTICLE INFO

Article history:
Received 11 July 2014
Received in revised form
20 March 2015
Accepted 21 March 2015
Available online 2 April 2015

Keywords:
Fully abstract semantics
Trace semantics
Untyped assembly language
Protected modules architectures
Formal languages

ABSTRACT

Protected module architectures (PMAs) are isolation mechanisms of emerging processors that provide 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.

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 (PMAs) [1–7], 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 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.

* Corresponding author.
E-mail addresses: marco.patrignani@cs.kuleuven.be (M. Patrignani), dave.clarke@it.uu.se (D. Clarke).

http://dx.doi.org/10.1016/j.cl.2015.03.002
1477-8424/© 2015 Elsevier Ltd. All rights reserved.

The fully abstract trace semantics has the following benefits. Firstly, it allows contextual equivalence to be disregarded, since contextual and trace equivalences 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 $\mathcal{A}+\mathsf{I}$: an assembly language enhanced 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 $\mathcal{A}+\mathsf{I}$ 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 $\mathcal{A}+\mathsf{I}$ code (Section 2.1). Then it discusses the pitfalls to avoid in order to obtain a fully abstract trace semantics for $\mathcal{A}+\mathsf{I}$ 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 [2–7]. 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 where 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 $\geq$ 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 $\mathcal{A}+\mathsf{I}$ along the way. All $\mathcal{A}+\mathsf{I}$ 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