# Fully Abstract Trace Semantics of Low-level Protection Mechanisms – Extended Abstract –

Marco Patrignani and Dave Clarke

iMinds-DistriNet, Dept. Computer Science, Katholieke Universiteit Leuven

**Abstract**

Fine-grained program counter-based memory access control mechanisms can be used to enhance low-level machine models to become the target of secure (fully abstract) compilation schemes. A secure compilation scheme reduces the power of a low-level attacker with code injection privileges to that of a high-level attacker which generally does not have such privileges. The existing trace semantics for a fine-grained program counter-based memory access control mechanism is not fully abstract, thus the protection mechanism it models cannot be used as the target of a *provably* secure compilation scheme. This paper shows why is such a fully abstract trace semantics needed, and proposes a correction to the existing trace semantics that makes it fully abstract and thus capable of supporting a secure compilation scheme.

Low-level machine code offers virtually no protection mechanism from an attacker that has code injection privileges, who is free to read sensible data and disrupt the execution flow with malicious code. A way to defend against these kind of attacks is by employing a fine-grained program counter-based memory access control mechanisms (FPMAC). The idea behind recent FPMACs implementations [3, 5, 8, 9], is to run sensitive code in isolation, so that malicious low-level code cannot tamper with it. Although details of these works differ, the FPMAC protection mechanism can be summarized as follows. The memory is logically divided into a protected and an unprotected section. Protected memory is further divided into a code and a data section. The code section contains a number of entry points: addresses which unprotected memory instructions can jump to and execute. The data section is accessible only from the protected section. The following table provides a representation of the access control model enforced by the protection mechanism.

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

This protection mechanism can be the target for a secure (fully abstract) compilation scheme, as the works of Agten *et al.* [2] and Patrignani *et al.* [6] have recently shown. A fully abstract compilation scheme preserves and reflects contextual equivalence at both high- and low-level code, thus it is well suited to expressing the preservation of security policies through compilation. Contextual equivalence is a relation between two programs $C_1$ and $C_2$ that cannot be distinguished by a third program called the tester. Formally, for all contexts $\mathbb{C}$ with a hole, if that hole is filled by either $C_1$ or $C_2$, the behavior of the context does not vary: $C_1 \simeq C_2 = \forall \mathbb{C}.\mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$, where $\Uparrow$ denotes divergence. Contextual equivalence can be used to model security policies as follows: saying that variable $f$ of program $C$ is confidential is equivalent to saying that $C$ is contextually equivalent to a program $C'$ that only differs from $C$ in its value for $f$. Denote a high-level program $C$ and its compilation $C^{\downarrow}$, a fully abstract compilation scheme is indicated as follows: $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$ [1].

In the proof of full abstraction for a compilation scheme, the most important result is proving the following direction of the coimplication: $C_1^{\downarrow} \not\simeq C_2^{\downarrow} \Rightarrow C_1 \not\simeq C_2$. This result can be proven by devising an algorithm that is capable of creating a high-level context that differentiates high-level component $C_1$ from $C_2$, given the low-level context that differentiates their compiled counterparts. Since low-level contexts are simply memory regions filled with instructions and a hole, they do not provide an inductive structure and thus offer little help in a proof. In order to circumvent this problem, the low level language can be equipped with a fully abstract trace semantics, denoted $\mathsf{Trace}_{\mathsf{L}}(M)$. Full abstraction of the trace semantics means that the notions of trace semantics and of contextual equivalence coincide [7]: $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \iff M_1 \simeq M_2$. This result grants an assumption of the form: $M_1 \not\simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2)$, which can be used in the aforementioned direction of a proof of full abstraction of a compilation scheme as follows: $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. At this point, the proof is concluded by devising an algorithm that takes two different low-level traces as input and produces a high-level context that differentiates between two programs, as Agten *et al.* [2] and Patrignani *et al.* [6] have shown.

## Flawed Trace Semantics for FPMAC-enhanced Low-level Languages

The first trace semantics for an untyped assembly language that adopts a FPMAC protection mechanism was presented by Agten *et al.* [2], based on labels defined by the grammar:

$$\Lambda ::= \alpha \mid \tau \qquad \alpha ::= \gamma? \mid \gamma! \qquad \gamma ::= \texttt{call}\ a(\overline{v}) \mid \texttt{ret}\ v$$

Those labels capture information flowing from protected to unprotected memory via decoration ! and the other direction via decoration ?. What is not captured by that trace semantics is other information flow, for example when protected code writes in unprotected memory. Consider two low-level programs $M_1$ and $M_2$ that exhibit the same trace semantics. If $M_1$ performs a write in unprotected memory but $M_2$ does not, an external program is able to tell whether it is interacting with $M_1$ or $M_2$ by simply monitoring the changes happening to unprotected memory. This result clearly contradicts assumption $M_1 \not\simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2)$, making the trace semantics not fully abstract.

To make the semantics fully abstract, the information exchanged between protected and unprotected memory must be restricted to what appears on the labels of the trace. Information can be exchanged between protected and unprotected memory in three ways: via reads and writes in memory, via values in registers and via flags. The FPMAC protection mechanism prevents external memory to write inside the protected memory, so the only dangerous reads and writes are from the protected memory towards unprotected one. Two possible solutions arise, namely, change the language: either those reads and writes are prohibited, or change the semantics: those reads and writes are captured by the labels in the trace semantics [4]. Values in registers are captured by the actions of the traces, however during a return, only one value is passed, thus all registers besides that one containing the communicated value must be reset to a default value. The same reasoning applies to flags, which must also be reset to a default value when the program counter jumps from protected to unprotected memory. An alternative to registers and values resetting would be to include them in the labels.

## Proof Sketch of Full Abstraction of FPMACs Trace Semantics

The proof of full abstraction of the trace semantics, $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \iff M_1 \simeq M_2$, is split in two cases, one for each direction of the co-implication.

The completeness case: $M_1 \simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2)$ is equivalently stated as: $\mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2) \Rightarrow M_1 \not\simeq M_2$. This can be proven in a known fashion: devise an algorithm that takes in input two different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ and the two low-level programs $M_1$ and $M_2$ generating them and outputs a low-level program $M$ that can interact with $M_1$ and $M_2$ such that $M$ diverges only with one of the two low-level programs. The input traces are sequences of actions such that even-numbered actions are messages from $M$ to either $M_1$ or $M_2$ and odd-numbered actions are messages from either $M_1$ or $M_2$ to $M$. Since the traces are different by hypothesis, there exists an odd-numbered index $j$ for which $\alpha_1^j \neq \alpha_2^j$. Based on the difference that is found in those odd-numbered actions, the algorithm outputs $M$ so that $M$ is able to terminate in a case and diverge in the other. Even though details of the algorithm are missing, the algorithm sketched here seems feasible to prove the completeness.

Let us now give an intuition behind the proof strategy devised for the soundness case: $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \Rightarrow \forall \mathbb{C}.\mathbb{C}[M_1]\Uparrow \iff \mathbb{C}[M_2]\Uparrow$. Coinductively define an equivalence relation between the states of the execution of $\mathbb{C}[M_1]$ and $\mathbb{C}[M_2]$. Break the operational semantics in two sets of transitions, based on whether the program counter points to protected or unprotected memory. Transitions where the program counter moves within unprotected or within protected memory are $\tau$ transitions in the trace semantics, thus they preserve state equivalence. Transitions where the program counter jumps from unprotected to protected memory preserve state equivalence as they are generated by $\mathbb{C}$, they are ?-decorated transitions in the trace semantics. Transitions where the program counter jumps from protected to unprotected memory must be proven to preserve state equivalence, they are !-decorated transitions in the trace semantics. This last point can be proven with any of the suggested solutions since they force all communicated information to be on the labels of the traces. Although the proof still needs to be formally carried out, we believe this proof strategy to be sound.

# References


[1] Martín Abadi. Protection in programming-language translations. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.

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

[3] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 375–388, New York, USA, 2011. ACM.

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

[5] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010.

[6] Marco Patrignani, Dave Clarke, Pieter Agten, and Frank Piessens. Secure Compilation of Object-Oriented Components to Untyped Assembly Code, October 2012. Submission in preparation.

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

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

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