Fully Abstract Trace Semantics for Low-level Isolation Mechanisms

Marco Patrignani $^1$  Dave Clarke $^{1,2}$

$^1$iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium

$^2$now at Dept. of Information Technology, Uppsala, Sweden

25 March 2014
Outline

1. Low-level Isolation Mechanisms: PMA
2. Reasoning about PMA
3. A Fully Abstract Trace Semantics
Why is PMA interesting?

- provides strong encapsulation at the lowest level of abstraction
Why is PMA interesting?

- provides strong encapsulation at the lowest level of abstraction
- can be exploited to develop a secure compiler [PCP13]
Why is PMA interesting?

- provides strong encapsulation *at the lowest level of abstraction*
- can be exploited to develop a secure compiler [PCP13]
- can be exploited to enforce a control-flow safe execution of C code [APJ14]
What is a Protected Modules Architecture (PMA)
What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
- implemented via Hypervisor, Hardware, Software
assembly-level isolation mechanism
implemented via Hypervisor, Hardware, Software
several research prototypes: Fides [SP12], Sancus [NAD+13], Flicker [MPP+08], TrustVisor [MLQ+10], Smart [EFPT12]
assembly-level isolation mechanism
implemented via Hypervisor, Hardware, Software
several research prototypes: Fides [SP12], Sancus [NAD₁³], Flicker [MPP⁺₀₈], TrustVisor [MLQ⁺₁₀], Smart [EFPT₁₂]
industrial prototype too: Intel SGX [MAB⁺₁₃]
assembly-level isolation mechanism
implemented via Hypervisor, Hardware, Software
several research prototypes: Fides [SP12], Sancus [NAD+13], Flicker [MPP+08], TrustVisor [MLQ+10], Smart [EFPT12]
industrial prototype too: Intel SGX [MAB+13]

Let’s see an example of PMA in action
PMA in action

```
0x0001  call 0xb53
0x0002  movs r0 0x0b55

0x0b52  movs r0 0x0b55
0x0b53  call 0x0002
0x0b54  movs r0 0x0001
0x0b55  ...

0xab00  jmp 0xb53
0xab01  ...
```

- **memory space**

- Protected module = protected memory
  - split in code and data
  - protected code is unrestricted
  - unprotected code is restricted
  - entry points for communication (■)
Low-level Isolation Mechanisms: PMA
Reasoning about PMA
A Fully Abstract Trace Semantics

PMA in action

0x0001 call 0xb53
0x0002 movs r0 0xb55

... 0xb53 call 0x0002 0xb54 movs r0 0x0001
0xb55 ...

0xab00 jmp 0xb53
0xab01 ...

- memory space
- protected module = protected memory

Marco Patrignani, Dave Clarke
Fully Abstract Trace Semantics for PMA
PMA in action

- memory space
- protected module $\Rightarrow$ protected memory
- split in code and data

```
0x0001   call 0xb53
0x0002   movs r0 0x0b55

0x0b52   movs r0 0x0b55
0x0b53   call 0x0002
0x0b54   movs r0 0x0001
0x0b55   ...

0xab00   jmp 0xb53
0xab01   ...
```
PMA in action

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

```assembly
0x0001   call 0xb53
0x0002   movs r0 0x0b55

0x0b52   movs r0 0x0b55
0x0b53   call 0x0002
0x0b54   movs r0 0x0001
0x0b55   ...

0xab00   jmp 0xb53
0xab01   ...
```
Low-level Isolation Mechanisms: PMA
Reasoning about PMA
A Fully Abstract Trace Semantics

PMA in action

0x0001 call 0xb53
0x0002 movs r0 0xb55
.

0x0b52 movs r0 0xb55
0x0b53 call 0x0002
0x0b54 movs r0 0x0001
0x0b55 ...

0xab00 jmp 0xb53
0xab01 ...

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

Marco Patrignani, Dave Clarke
Fully Abstract Trace Semantics for PMA
PMA in action

```
0x0001  call 0xb53  
0x0002  movs r0 0x0b55  
      
0xb52  movs r0 0xb55  
0xb53  call 0x0002  
0xb54  movs r0 0x0001  
0xb55  ...  
```

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

Marco Patrignani, Dave Clarke
PMA in action

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

```
0x0001  call 0xb53
0x0002  movs r0 0x0b55
...
0x0b52  movs r0 0x0b55
0x0b53  call 0x0002
0x0b54  movs r0 0x0001
0x0b55  ...
0xab00  jmp 0xb53
0xab01  ...
```
PMA in action

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

```
0x0001  call 0xb53
0x0002  movs r0 0xb55

0x0b52  movs r0 0xb55
0x0b53  call 0x0002
0x0b54  movs r0 0x0001
0x0b55  ...

0xab00  jmp 0xb53
0xab01  ...
```
PMA in action

0x0001 call 0xb53
0x0002 movs r0 0x0b55

0x0b52 movs r0 0x0b55
0x0b53 call 0x0002
0x0b54 movs r0 0x0001
0x0b55 ...

0xab00 jmp 0xb53
0xab01 ...

- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted
memory space

- protected module = protected memory

- split in code and data

- protected code is unrestricted

- unprotected code is restricted

- entry points for communication (■)
PMA in action

- memory space
- protected module $= \text{protected memory}$
- split in code and data
- protected code is unrestricted
- unprotected code is restricted
- entry points for communication (■)

```
0x0001 call 0xb53
0x0002 movs r0 0x0b55

0x0b52 movs r0 0x0b55
0x0b53 call 0x0002
0x0b54 movs r0 0x0001
0x0b55 ...

0xab00 jmp 0xb53
0xab01 ...
```
Challenge #1: Reasoning about PMA code

Formally, we can state:

$$P_1 \equiv P_2 \iff \forall C. C[P_1] \Rightarrow C[P_2]$$

where contexts are complex to reason about (but very precise).
Challenge #1: Reasoning about PMA code

0x0001 call 0xb52
0x0002 ...

0x0b52 movi r0 1
0x0b53 movi r1 0x0b56
0x0b54 jl r1
0x0b55 call 0xab01
0x0b56 ret

0xab01 ...

To reason about this code, we use contextual equivalence.

Formally, \( P_1 \approx P_2 \equiv \forall C. C[P_1] \uparrow \iff C[P_2] \uparrow \)

Contexts are complex to reason about (but very precise).

Marco Patrignani, Dave Clarke
Challenge #1: Reasoning about PMA code

- to reason about this code we use contextual equivalence
- Formally $P_1 \simeq P_2 \iff \forall C. C[P_1] \uparrow \iff C[P_2] \uparrow$

```assembly
0x0001     call 0xb52
0x0002     ...

0x0b52     movi r0 1
0x0b53     movi r1 0x0b56
0x0b54     jl r1
0x0b55     call 0xab01
0x0b56     ret

0xab01     ...
```
Challenge #1: Reasoning about PMA code

- to reason about this code we use contextual equivalence
- Formally $P_1 \simeq P_2 \triangleq \forall C. C[P_1] \uparrow \iff C[P_2] \uparrow$
- contexts are complex to reason about (but very precise)
Challenge #1: Reasoning about PMA code

traces are simpler [JR05]

0x0001  call 0xb52
0x0002  ...
...
0xb52  movi r0 1
0xb53  movi r1 0xb56
0xb54  jl r1
0xb55  call 0xab01
0xb56  ret
...
0xab01  ...

behaviour in this case is:
call in, ret 1 or call in, call out
traces rely only on the PMA code they describe what can be observed from the outside of protected PMA code
Challenge #1: Reasoning about PMA code

```
0x0001   call 0xb52
0x0002   ...
...
0xb52    movi r0 1
0xb53    movi r1 0xb56
0xb54    jl r1
0xb55    call 0xab01
0xb56    ret
```

- traces are simpler [JR05]
- behaviour in this case is:
Challenge #1: Reasoning about PMA code

- traces are simpler [JR05]
- behaviour in this case is: call in

```
0x0001  call 0xb52
0x0002  ...
...

0xb52  movi r0 1
0xb53  movi r1 0xb56
0xb54  jl r1
0xb55  call 0xab01
0xb56  ret
...

0xab01  ...
```
Challenge #1: Reasoning about PMA code

- traces are simpler [JR05]
- behaviour in this case is: call in, ret 1
Challenge #1: Reasoning about PMA code

- traces are simpler [JR05]
- behaviour in this case is: call in, ret 1 or call in,

```
0x0001  call 0xb52
0x0002  ...

0xb52  movi r0 1
0xb53  movi r1 0xb56
0xb54  jl r1
0xb55  call 0xab01
0xb56  ret

0xab01  ...
```
Challenge #1: Reasoning about PMA code

- traces are simpler [JR05]
- behaviour in this case is: call in, ret 1
  or call in, call out
Challenge #1: Reasoning about PMA code

0x0001  call 0xb52
0x0002  ...

0x0b52  movi r0 1
0x0b53  movi r1 0xb56
0x0b54  jl r1
0x0b55  call 0xab01
0x0b56  ret

0xab01  ...

- traces are simpler [JR05]
- behaviour in this case is: call in, ret 1 or call in, call out
- traces rely only on the PMA code
Challenge #1: Reasoning about PMA code

- traces are simpler [JR05]
- behaviour in this case is: call in, ret 1 or call in, call out
- traces rely only on the PMA code
- they describe what can be observed from the outside of protected PMA code

```assembly
0x00001 call 0xb52
0x00002 ...
...
0x0b52 movi r0 1
0x0b53 movi r1 0x0b56
0x0b54 jl r1
0x0b55 call 0xab01
0x0b56 ret
...
0xab01 ...
```
A trace semantics for PMA

- define states $S$ for programs
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\rightarrow$ : $S \times S$
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\rightarrow: S \times S$
- define labels (observables) $\alpha$
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\rightarrow: S \times S$
- define labels (observables) $\alpha: \text{call } p \bar{r}$

 Marco Patrignani, Dave Clarke
 Fully Abstract Trace Semantics for PMA
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\delta: S \times S$
- define labels (observables) $\alpha: \text{call } p \bar{r} | \text{ret } r_0$
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\rightarrow^i: S \times S$
- define labels (observables) $\alpha: \text{call } p \bar{r} \mid \text{ret } r_0$
- define a semantics with labels $\xrightarrow{\alpha} : S \times \alpha \times S$
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $\rightarrow: S \times S$
- define labels (observables) $\alpha: \text{call } p \bar{r} | \text{ret } r_0$
- define a semantics with labels $\xrightarrow{\alpha}: S \times \alpha \times S$
- $\text{TR}(P) = \{\bar{\alpha} \mid \exists S'. S(P) \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} S'\}$
A trace semantics for PMA

- define states $S$ for programs
- define a semantics for PMA only: $i: S \times S$
- define labels (observables) $\alpha : \text{call } p \bar{r} | \text{ret } r_0$
- define a semantics with labels $\alpha : S \times \alpha \times S$
- $\text{TR}(P) = \{ \overline{\alpha} | \exists S'. S(P) \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} S' \}$

\[
\text{TR} = \left\{ \alpha = \left\{ \begin{array}{l}
\xrightarrow{i} ; \\
\text{call } p \bar{r} \\
\text{ret } r_0 \\
\xrightarrow{\alpha}
\end{array} \right\} ; \right\}
\]
Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✅
Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✅
- precise formalism? ❌
Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ❌

- PMA code can write in unprotected memory
Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ✗
  1. PMA code can write in unprotected memory
  2. Flags convey information across function calls
Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✓
- precise formalism? ✗
  1. PMA code can write in unprotected memory
  2. flags convey information across function calls
  3. registers besides $r_0$ in ret as well
To ensure maximal precision, prove the trace semantics to be fully abstract
To ensure maximal precision, prove the trace semantics to be fully abstract

i.e. there are no other things that we missed
The spectrum of full abstraction [Cur07]

\[ TR = \begin{cases} \alpha = \{ \text{call } p \bar{r} \} ; \{ \text{ret } r_{\theta} \} ; \alpha \Rightarrow \alpha \} \end{cases} \]
The spectrum of full abstraction [Cur07]

\[ TR = \begin{cases} \alpha = \{ \text{\texttt{call p r}} \}; & \text{X} \\ \text{\texttt{ret r}}_\theta \} ; \end{cases} \]

\[ TR_L = \begin{cases} \alpha = \{ \text{\texttt{call r f}} \} \\ \text{\texttt{ret r f}} \} & \checkmark \\ \text{\texttt{movs r v}} \} ; \end{cases} \]
The spectrum of full abstraction [Cur07]

\[ TR = \begin{cases} \alpha = \{ i; \text{call p } \overline{r} \text{ret } r_\theta \}; \end{cases} \]

\[ TR_L = \begin{cases} \alpha = \{ i; \text{call } \overline{r} \overline{f} \text{ret } \overline{r} \overline{f} \text{movs } r \text{v } \}; \end{cases} \]

\[ TR_S = \begin{cases} \alpha = \{ i; \text{call p } \overline{r} \text{ret } r_\theta \}; \end{cases} \]

\[ TR = \begin{cases} \alpha = \{ i; \text{call p } \overline{r} \text{ret } r_\theta \}; \end{cases} \]
The spectrum of full abstraction [Cur07]

\[ TR = \left\{ \alpha = \{ \text{call } p \bar{r} \}, \text{ret } r_0 \}; \alpha \xlongequal{\alpha} \right\} \]

\[ TR_L = \left\{ \alpha = \{ \text{call } \bar{r} \bar{f} \}, \text{ret } r_v \}; \alpha \xlongequal{\alpha} \right\} \]

\[ TR_S = \left\{ \alpha = \{ \text{call } p \bar{r} \}, \text{ret } r_0 \}; \alpha \xlongequal{\alpha} \right\} \]
The spectrum of full abstraction [Cur07]

\[ TR = \{ \alpha = \{ \begin{array}{c} i; \\
\text{call } p \overline{r} \\
\text{ret } r_0 \end{array} \}; \} \]

\[ TR_L = \{ \alpha = \{ \begin{array}{c} i; \\
\text{call } \overline{r} f \\
\text{ret } \overline{r} f \\
\text{movs } r v \end{array} \}; \} \]

\[ TR_S = \{ \alpha = \{ \begin{array}{c} i; \\
\text{call } p \overline{r} \\
\text{ret } r_0 \end{array} \}; \} \]

\[ TR_X(P_1) = TR_X(P_2) \iff P_1 \simeq P_2 \]
Conclusion
Conclusion

- PMA is a very powerful, emerging security construct
Conclusion

- PMA is a very powerful, emerging security construct
- trace semantics for PMA enables easier reasoning about PMA code
Conclusion

- PMA is a very powerful, emerging security construct
- Trace semantics for PMA enables easier reasoning about PMA code
- If the trace semantics is fully abstract, trace equivalence can replace contextual equivalence altogether
PMA is a very powerful, emerging security construct
trace semantics for PMA enables easier reasoning about PMA code
if the trace semantics is fully abstract, trace equivalence can replace contextual equivalence altogether
the paper provides two, examples of fully abstract trace semantics for PMA
Thank you!

Qs ?
Bibliographical References I

Bibliographical References II

