Securing Interruptible Enclaves

Matteo Busi
Comp. Sci. Dept., Univ. Pisa
matteo.busi@di.unipi.it

Job Noorman
Comp. Sci. Dept., KU Leuven
job.noorman@cs.kuleuven.be

Jo Van Bulck
Comp. Sci. Dept., KU Leuven
jo.vanbulck@cs.kuleuven.be

Letterio Galletta
IMT, Lucca
letterio.galletta@imtlucca.it

Pierpaolo Degano
Comp. Sci. Dept., Univ. Pisa
degano@di.unipi.it

Jan Tobias Mühlberg
Comp. Sci. Dept., KU Leuven
jantobias.muehlberg@cs.kuleuven.be

Frank Piessens
Comp. Sci. Dept., KU Leuven
frank.piessens@cs.kuleuven.be

1 Introduction

Computer systems often provide hardware support for isolation mechanisms like privilege levels, virtual memory, or enclaved execution. Over the past years, several successful software-based side-channel attacks have been developed that break, or at least significantly weaken the isolation that these mechanisms offer. Extending a processor with new (micro-)architectural features brings a risk of enabling new such side-channel attacks. In particular, interrupt-based attacks are very effective in breaking confidentiality and integrity of code and data in *enclaves*, i.e. protected memory regions [1–3, 6]. An attacker capable of raising interrupts at will, and of managing them can observe different timings of enclaved executions, so possibly breaking isolation.

Here, we outline our work in extending a micro-processor with carefully designed interruptible enclaves *without* weakening security. We started from the existing Sancus platform [4, 5] that supports *non-interruptible* enclaved execution and we extend it with interrupts. To obtain strong security guarantees, we formalize both versions of Sancus and we prove them fully abstract. Roughly, our full abstraction theorem guarantees that the attacker gains *exactly* the same information without and with interrupts: adding interruptibility opens no new avenues of attack. Our result is reflected in the implementation of the secure interrupt handling mechanism in Sancus – actually this required back-and-forth interactions for tuning the formalization and the implementation.

2 Outline

We formally define the operational semantics of the original, uninterruptible Sancus (Sancus^{H}) and of the secure interruptible version (Sancus^{L}). The two share most of their structure and just differ in the way they deal with interrupts.

The model. The *memory* $\mathcal{M}$ and the *register file* $\mathcal{R}$ are functions from 2^{16} locations to bytes, and from each of the 16 registers in a 2-bytes word. Reads and writes are formalized much in the standard way, with particular care of faithfully reflecting those of the hardware. The memory is partitioned into a protected part, or *module* $\mathcal{M}_M$, that contains code and data of the enclave, and an unprotected section. The code of a module runs in isolation: it cannot exchange any information with the outside, except when it jumps out of the protected memory or when it is invoked at a specific entry point.

The attacker C is a *context*, consisting of the (code and data in the) unprotected section $\mathcal{M}_C$ and of an *I/O device*, used for raising interrupts at specific CPU cycles. We formalize the device as a *deterministic I/O automaton* $\mathcal{D}$ that evolves synchronously with the CPU through actions for raising interrupts, and for reading from and writing to the CPU.

A program $C[\mathcal{M}_M] = \langle \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{D} \rangle$ executes either in *protected mode*, when the code of the enclave runs, or in *unprotected mode*, when the code of the attacker runs. A memory access control relation (MAC) checks whether an instruction can be executed in the current mode.

Operational semantics. The configurations of both transition systems are $\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle$, where (i) δ is the current state of $\mathcal{D}$; (ii) t is the CPU current time; (iii) t_a is the arrival time of the last pending interrupt; (iv) $\mathcal{M}$, $\mathcal{R}$ are the current memory and the current register file; (v) pc_{old} is the previous program counter, used by MAC; (vi) $\mathcal{B}$ is the *backup*, a software inaccessible storage to save the enclave state while handling an interrupt raised in protected mode.

The inference rules defining the behaviour of both Sancus^{H} and Sancus^{L} are rather detailed to accurately describe hardware operations. We only show the rule for moving the value of register $\mathtt{r}_1$ in $\mathtt{r}_2$, through the instruction $i = \mathtt{MOV\ r_1\ r_2}$

$$\frac{i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \qquad \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}$$

The first premise checks via MAC if the instruction i can be executed. The second updates the program counter and copies the value of $\mathtt{r}_1$ in $\mathtt{r}_2$. The third premise records the duration of the instruction, syncronizing $\mathcal{D}$ and the CPU; in addition it checks for interrupts to handle (via t_a'). The fourth uses the interrupt logic represented by $\hookrightarrow_I$ that differs for Sancus^{H} and Sancus^{L}, so determining $\rightarrow$ in the conclusion.

For Sancus^{H} the last precondition is trivially the identity relation, while for Sancus^{L} the arrow $\hookrightarrow_I$ models the mechanism (*mitigation*) that makes secure the interruptible enclaved execution. This is an example of inference rule:

$$\frac{k = \mathtt{MAX\_TIME} - (t - t_a) \qquad pc_{old} \vdash_{mode} \mathtt{PM} \qquad \mathcal{R}[\mathtt{sr}].\mathtt{GIE} = 1 \qquad t_a \neq \bot \qquad \mathcal{R}' = \mathcal{R}_0[\mathtt{pc} \mapsto isr] \qquad \mathcal{D} \vdash \delta, t, \bot \curvearrowright_D^{6+k} \delta', t', t_a' \qquad \mathcal{B}' = \langle \mathcal{R}, pc_{old}, t - t_a \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_I \langle \delta', t', \bot, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

The key of mitigation is the value k that is used in the sixth premise to keep constant the delay between the detection of an interrupt and its handling (`MAX_TIME` is the longest instruction duration). The second premise states that the CPU is running in protected mode. The third and the fourth state that interrupts are enabled and that there is one pending. The fifth premise resets the register file to $\mathcal{R}_0$, their initial value, and sets the program counter to the address of the entry point of the interrupt handler. The sixth premise (and an appropriate rule for when interrupts return) implement the mitigation. The seventh saves in the backup the state of the enclave, to be restored upon completion of the interrupt.

Full abstraction. We prove that what an attacker can learn from an enclave is exactly the same before and after adding the support for interrupts. We show that the semantics of Sancus$^{\mathsf{L}}$ is *fully abstract* w.r.t. that of Sancus$^{\mathsf{H}}$, i.e. all the attacks that can be carried out in Sancus$^{\mathsf{L}}$ can also be carried out in Sancus$^{\mathsf{H}}$, and viceversa.

Let $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$ denote a *halting computation in* Sancus$^{\mathsf{H}}$, and let two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ be *contextually equivalent in* Sancus$^{\mathsf{H}}$, written $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$, iff for all contexts C, $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}}$. Similarly for Sancus$^{\mathsf{L}}$. We prove the absence of interrupt-based attacks:

Theorem 2.1 (Full abstraction).
$\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ (\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'})$.

Proof sketch

– $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ Since programs in Sancus$^{\mathsf{H}}$ behave like those in Sancus$^{\mathsf{L}}$ with no interrupts, it suffices to introduce the subset of *interrupt-less* contexts for Sancus$^{\mathsf{L}}$ that never raise interrupts. The thesis follows because an enclave hosted in a interrupt-less context terminates in Sancus$^{\mathsf{L}}$ whenever it does in Sancus$^{\mathsf{H}}$.

– $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ We first introduce the notion of observable behavior, i.e. the traces of $C[\mathcal{M}_M]$ defined by the Sancus$^{\mathsf{L}}$ semantics. Traces are built using three observables: (i) $\bullet$ denotes that the computation halts; (ii) `jmpIn?`$(\mathcal{R})$ denotes that the CPU enters the protected mode, where $\mathcal{R}$ are the observed registers and (iii) `jmpOut!`$(\Delta t; \mathcal{R})$ denotes the exit from protected mode with observed registers $\mathcal{R}$ and with Δt representing the end-to-end time measured by an attacker for code running in protected mode.

The proof follows the steps in Figure 1, where $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ means that $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ have the same traces. Implication (i) shows that the attacker in Sancus$^{\mathsf{L}}$ at most observes *as much as* traces say; implication (ii) shows that the attacker in Sancus$^{\mathsf{H}}$ is *at least as powerful as* described by traces; finally implication (iii) is our thesis that follows by transitivity.

To prove (i) first we show that the mitigation guarantees that the behavior of the context (in unprotected mode) does not depend on that of the enclave (in protected mode) and viceversa. The thesis follows since if $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ and $\overline{\beta}$ is a trace of $C[\mathcal{M}_M]$, then $\overline{\beta}$ is also a trace of $C[\mathcal{M}'_M]$.

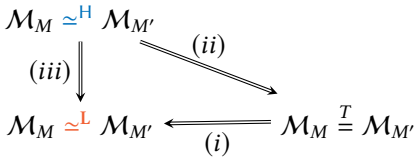

Figure 1. The steps for proving preservation of behavior.

The proof of (ii) is by contraposition: if two modules have different traces, there exists a context that distinguishes them. We build such a context through a *backtranslation* algorithm. Because of the strong limitations – e.g. only 64KB of memory is available – constructing such a context only in unprotected memory is infeasible. The backtranslation defines and uses both the unprotected memory, and the I/O device, which has unrestricted memory. Roughly the idea is to take a trace of $\mathcal{M}_M$ and one of $\mathcal{M}_{M'}$ that differ for one observable, and build a context C such that $\mathcal{M}_M$ converges and $\mathcal{M}_{M'}$ does not, so contradicting the hypothesis $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$.

3 Conclusions

We used full abstraction to formally assure that extending a microprocessor with a new feature does not weaken the isolation mechanisms that the processor offers. An important challenge for future would be to introduce some *quantification* of the weakening of security, so to allow the introduction of some bounded amount of leakage.

A technical report with all the missing details of our formalization is available on request.

References

[1] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security 2018*, William Enck and Adrienne Porter Felt (Eds.).

[2] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. [n. d.]. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. ([n. d.]).

[3] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security 2017*.

[4] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Security 2013*, Samuel T. King (Ed.).

[5] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* 20, 3 (2017).

[6] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS 2018*.