10.4 Instrumentation

Goal: Consider program P with attack $\pi = (t_\pi, \text{shist}, \text{lhist})$.
Characterise TSO witnesses of π in P
by SC computations in a program P_π
that is instrumented for attack π.

By instrumentation we mean we replace
every thread in P by a modified version.

Tricky: • Unbounded store buffers
• Unbounded happens-before dependencies.

Instrumenting the attacker:

Idea: • Emulate store buffering under SC
using auxiliary addresses
↳ When the attacker executes the delayed store st_π,
under SC it is done right behind the issue action.
↳ To mimic store buffering, st_π now accesses
an auxiliary address that helpers do not load.
↳ Indeed, in $\tilde{C}_4$ the helpers are no longer active
and hence do not access the delayed stores.
• How many auxiliary addresses?
↳ One per address in the program (last store).

Technically: • Starting from st_π to address a,
stores are replaced by st_π^{aux} to addresses (a,d). // d = delay.
• As long as address a has not been written,
(a,d) holds the initial value 0.
When the attacker stores v to address a,
we set $mem[(a,d)] = (v,d)$.
Hence, (a,d) always holds the most recent store
to address a.

- A load $r \leftarrow mem[a]$ of the attacker reads value v from the buffer whenever $mem[(a,d)] = (v,d)$. Otherwise $mem[(a,d)] = 0$, and the load obtains $v = mem[a]$ from memory.

Definition (instrumentation of the attacker):
Consider thread t_A regs r^* init lo begin $\langle inst \rangle^*$ end.
Let $A = (t_A, stinst, ldinst)$ be the attack.
The instrumentation of t_A for attack A is the thread:

$[\![t_A]\!] :=$ thread t_A regs r^* init lo

begin
$\langle linst \rangle^*$ // Source code
$[\![stinst]\!]_{A_1}$ // Move to copy of source code
$[\![ldinst]\!]_{A_1}$
$[\![\langle linst \rangle]\!]^*_{A_2}$ // Copy of source code.
end.

with
$[\![\underbrace{l_1 : mem[e_1] \leftarrow e_2 \text{ goto } l_2}_{stinst}]\!]_{A_1} :=$ $l_1 : mem[(e_1,d)] \leftarrow (e_2,d)$ goto $\tilde{l}_x$;
$\tilde{l}_x : mem[as_A] \leftarrow e_1$ goto l_2;

$[\![\underbrace{l_1 : r \leftarrow mem[e] \text{ goto } l_2}_{ldinst}]\!]_{A_2} :=$ $\tilde{l}_1 :$ assert $mem[(e,d)] = 0$ goto $\tilde{l}_x$
$\tilde{l}_{x1} :$ $mem[bb] \leftarrow$ true goto $\tilde{l}_{x2}$;
$\tilde{l}_{x2} :$ $mem[(e,bb)] \leftarrow$ lda goto $\tilde{l}_{x3}$;

$[\![l_1 : mem[e_1] \leftarrow e_2 \text{ goto } l_2]\!]_{A_2} :=$ $\tilde{l}_1 :$ $mem[(e_1,d)] \leftarrow (e_2,d)$ goto $\tilde{l}_2$;

$[\![l_1 : r \leftarrow mem[e] \text{ goto } l_2]\!]_{A_2} :=$ $\tilde{l}_1 :$ assert $mem[(e,d)] = 0$ goto $\tilde{l}_{x1}$;
$\tilde{l}_{x1} :$ $r \leftarrow mem[e]$ goto $\tilde{l}_2$;
$\tilde{l}_1 :$ assert $mem[(e,d)] \neq 0$ goto $\tilde{l}_{x2}$;
$\tilde{l}_{x2} :$ $(r,d) \leftarrow mem[(e,d)]$; goto $\tilde{l}_2$;

$[\![l_1 : \text{local goto } l_2]\!]_{\pi_2} := \widehat{l_1} : \text{local goto } \widehat{l_2};$

$[\![l_1 : \text{nfence goto } l_2]\!]_{\pi_2} := /$

Comment:

- Note that the instrumentation $[\![\text{st/ist}]\!]_{\pi_1}$ keeps the address used in the store in a fresh address a_{st_π}.
- The instrumentation deletes fences as they forbid to delay st_π over ld_π.
- The instrumentation $[\![\text{ld/ist}]\!]_{\pi_1}$ checks the value is not read early. Moreover, if such a happens-before address (a, hb) to access level load, lda. It also sets a flag hb to forbid helper actions that do not contribute to happens-before path τ_3.

Instrumenting helpers:

Idea: · How to decide whether a new action act is in happens-before relation with an earlier action act' so that $ld_\pi \to^*_{hb} act' \to^*_{hb} act$?

Need to know two facts:

↳ Has the thread of act already contributed an action act' to τ_3? In this case, $act' \to^*_{po} act$. The information about such a contribution can be kept in the control-flow of the helper.

↳ Does τ_3 contain a load or store access to addr(act)?
- If there was a load act' = ld, we can add a store act = st and get $ld \to^*_{hb} st$
- If there was a store, we are free to add a load or a store.

↳ Need one auxiliary address (a, hb)
per address a in the program.
The addresses (a, hb) range over the domain

$$\{0, lda, sta\}$$

of access types.
It is sufficient to store the maximal access type
wrt. the ordering:

$$0 \text{ (no access)} < lda \text{ (load access)} < sta \text{ (store access)}.$$

Technically: The augmentation on a thread's contribution to $\tilde{\tau}_3$
+ access types is based on the following lemma.

Lemma:
Consider $\tilde{\tau} = \tilde{\tau}_1.\, act_1.\, \tilde{\tau}_2 \in C_{sc}(P)$
where for all $act_2 \in \tilde{\tau}_2$ we have $act_1 \rightarrow_{hb}^* act_2$.
Then $\tilde{\tau}.act$ satisfies $act_1 \rightarrow_{hb}^* act$

if
(1) $\exists\, act_2 \in \tilde{\tau}_1.\, \tilde{\tau}_2 :\ thread(act_2) = thread(act)$,
(2) act is a load whose address is stored in $act_2.\, \tilde{\tau}_2$, or
(3) act is a store (with issue) whose address
is loaded or stored in $act_2.\, \tilde{\tau}_2$.

W.R.t. this, the instrumentation of helpers is as follows.

Definition (Instrumentation of helpers):
Consider thread t regs r^* init to begin $\langle instr \rangle^*$ end.
The instrumentation of t is
$[\![t]\!]$:= thread $\tilde{t}$ regs $\tilde{r}, r^*$ init to

begin
$[\![\langle instr \rangle]\!]^*_{H_0}$ $[\![\langle lds/sts\, instr \rangle]\!]^*_{H_1}$ $[\![\langle instr \rangle]\!]^*_{H_2}$ $[\![\langle \rangle]\!]^*_{H_3}$
end

- Here, $\langle ldst\,inst\rangle^*$ is the subsequence of all load and store instructions. Their instrumentation $[\![\langle ldst\,inst\rangle]\!]^*_{H_2}$ is used to move to the code copy $[\![\langle linst\rangle]\!]^*_{H_2}$
- Let $\langle l\rangle^*$ be all labels used by the thread. The instructions $[\![\langle l\rangle]\!]^*_{H_3}$ raise a success flag when a TSO witness has been found.
- The instrumentation $[\![\langle linst\rangle]\!]^*_{H_0}$ of the original source code forces the helper to either enter the code copy $[\![\langle linst\rangle]\!]^*_{H_2}$ or stop when the hb-flag is raised.

The functions are as follows:

$[\![l_1 : instr \ \underline{goto}\ l_2]\!]_{H_0} :=$
$l_1 :$ assert $mem[hb] = 0$ goto l_x;
$l_x :$ instr goto l_2;

$[\![l_1 : r \leftarrow mem[e] \ \underline{goto}\ l_2]\!]_{H_1} :=$
$l_1 :$ assert $mem[(e, hb)] =$ sta goto $\widetilde{l_x}$;
$\widetilde{l_x} : r \leftarrow mem[e]$ goto $\widetilde{l_2}$;

$[\![l_1 : mem[e_1] \leftarrow e_2 \ \underline{goto}\ l_2]\!]_{H_1} :=$
$l_1 :$ assert $mem[(e_1, hb)] \geq$ lda goto $\widetilde{l_{x_1}}$
$\widetilde{l_{x_1}} : mem[e_1] \leftarrow e_2$ goto $\widetilde{l_{x_2}}$;
$\widetilde{l_{x_2}} : mem[(e_1, hb)] \leftarrow$ sta goto $\widetilde{l_2}$;

$[\![l_1 : local/\underline{mfence}\ goto\ l_2]\!]_{H_2} :=$ $\widetilde{l_1} :$ local/mfence goto $\widetilde{l_2}$;
$[\![l_1 : mem[e_1] \leftarrow e_2 \ goto\ l_2]\!]_{H_2} :=$ $\widetilde{l_1} : mem[e_1] \leftarrow e_2$ goto $\widetilde{l_x}$;
$\widetilde{l_x} : mem[(e_1, hb)] \leftarrow$ sta goto $\widetilde{l_2}$;

$[\![l_1 : r \leftarrow mem[e] \ goto\ l_2]\!]_{H_2} :=$ $\widetilde{l_1} : \widetilde{r} \leftarrow e$ goto $\widetilde{l_{x_1}}$;
$\widetilde{l_{x_1}} : r \leftarrow mem[\widetilde{r}]$ goto $\widetilde{l_{x_2}}$
$\widetilde{l_{x_2}} : mem[(\widetilde{r}, hb)] \leftarrow$
$\max\{lda, mem[(\widetilde{r}, hb)]\}$
goto $\widetilde{l_2}$;

$[\![l]\!]_{H_3} :=$ $\tilde{l}$: $\tilde{r} \leftarrow mem[a_{sl_H}]$ goto $\tilde{l}_{x_1}$;

$\tilde{l}_{x_1}$: $\tilde{r} \leftarrow mem[(\tilde{r}, hb)]$ goto $\tilde{l}_{x_2}$;

$\tilde{l}_{x_2}$: assert $\tilde{r} \neq 0$ goto $\tilde{l}_{x_3}$;

$\tilde{l}_{x_3}$: $mem[suc] \leftarrow$ true goto $\tilde{l}_{x_4}$;

Note:
In the instrumentation of loads, $[\![l_1 : r \leftarrow mem[e]$ goto $l_2]\!]_{H_2}$, auxiliary register $\tilde{r}$ ensures that we do not overwrite the address given by e when modifying r (may be used within e).

Theorem (Soundness and completeness of instrumentation):
$\forall$ Hack H is feasible in program P
iff P_H reaches a goal configuration under SC.

$\forall$ goal configuration is a pair (pc, val)
with $val(suc) = true$.

Theorem:
- Program P is robust iff no instrumentation P_H reaches a goal configuration under SC
- If the data domain is finite and given as input, robustness is PSPACE-complete.

Proof:
Upper bound: We show that the complement of robustness, the non-robustness problem (given a program P, check that P is not robust) can be solved in non-deterministic polynomial space (NPSPACE).
By Savitch's Theorem NPSPACE = PSPACE,
and hence non-robustness $\in$ PSPACE
$\Rightarrow$ We negate the answer and get robustness $\in$ PSPACE.

Essentially, we use that

$$\text{co-NPSPACE} = \text{co-PSPACE} = \text{PSPACE} \; (= \text{NPSPACE}).$$

To solve non-robustness in NPSPACE,
we guess a suitable attack A
and compute the linear-size instrumentation P_A.
Then we guess a reaching path in P_A.
For the path, we only need to store

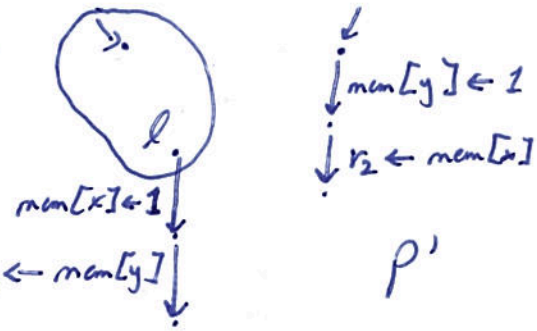
- the current configuration (works in linear space)
- the number of steps taken (works in linear space as well).

Return yes, if a goal configuration is reached.
Return no, if the search for a path deadlocks
or the number of steps exceeds the number of configurations.
For the latter, note that
there are 2^n configurations with n bits.
We need another n-bit to count to 2^n.

Lower bound: We first give a reduction of SC-reachability
in Boolean programs to non-robustness.

- Consider a single-threaded Boolean program P
with control-location ℓ

- Using a second thread and fresh variables,
we append a Dekker cycle to ℓ:



- Then ℓ is SC-reachable in P
iff P' is not robust.
- Since control-state reachability
is PSPACE-hard, so is non-robustness.

To see that also robustness is PSPACE-hard,
consider a problem Prob ∈ PSPACE
that we want to reduce to robustness.
Since PSPACE is closed under complement,
we have

$$\text{co-Prob} \in \text{PSPACE}.$$

We just showed that non-robustness is PSPACE-hard.
Hence there is a reduction

$$f: \text{co-Prob} \to \text{non-Rob}$$

so that

instance $i \in$ co-Prob iff $f(i)$ is not robust.

Since $i \in$ co-Prob iff $i \notin$ Prob,
we have

$i \in$ Prob iff $f(i)$ is robust.

So function f is also a reduction of Prob to robustness. □