Appendix to "Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities"

AÏNA LINN GEORGES, Aarhus University, Denmark
ALIX TRIEU, ANSSI, France
LARS BIRKEDAL, Aarhus University, Denmark

This is the appendix to the paper "Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities", including a proof sketch of the fundamental theorem of logical relations, formal definitions of components, linking and contexts, and the definition of a safe word.

1 PROOF SKETCH OF THE FUNDAMENTAL THEOREM

One of the key facts supporting that our logical relation models temporal stack safety is the following monotonicity lemma. It describes the monotonicity of $\mathcal{V}$ relative to a word's current read authority. It entails in particular that a stack capability that is valid in some world W is also valid in a relative-to-its-current-address-future world; and thus it does not depend on the contents of higher stack frames.

Lemma 1.1 (Address Relative Monotonicity)**.**

(1) $p \in \{\textsc{urwlx}, \textsc{urwl}, \textsc{urwx}, \textsc{urw}\} \wedge W' \sqsupseteq^a W \rightarrow \mathcal{V}(W)(p,g,b,e,a) \mathrel{-\!\!*} \mathcal{V}(W')(p,g,b,e,a)$

(2) $p \notin \{\textsc{urwlx}, \textsc{urwl}, \textsc{urwx}, \textsc{urw}\} \wedge W' \sqsupseteq^e W \rightarrow \mathcal{V}(W)(p,g,b,e,a) \mathrel{-\!\!*} \mathcal{V}(W')(p,g,b,e,a)$

Furthermore, the address relative temporal future world relation can be weakened by lowering the associated address, and the value relation is monotone wrt. private future worlds (for all non-directed capabilities).

Lemma 1.2 (Address Relative Weakening)**.** $a' \leq a \rightarrow W' \sqsupseteq^a W \rightarrow W' \sqsupseteq^{a'} W$

Lemma 1.3 (Private Monotonicity)**.** *Let w be a word that is not* DIRECTED*. Then* $W' \sqsupseteq^{priv} W \rightarrow \mathcal{V}(W)(w) \mathrel{-\!\!*} \mathcal{V}(W')(w)$.

THEOREM 1.4 (FTLR). *theorem Assume that $p = \textsc{rx}$, $p = \textsc{rwx}$ or $(p = \textsc{rwlx} \wedge g = \textsc{directed})$. Assume also that $\mathcal{V}(W)(p,g,b,e,a)$. Then we have that $\mathcal{E}(W)(p,g,b,e,a)$.*

PROOF. Upon unfolding the definition of $\mathcal{E}$, our goal is to prove that the weakest precondition holds for a program pointed to by the program counter (p,g,b,e,a), given the predicates of a safe register state. According to the underlying definition of Iris weakest preconditions, this amounts to showing that the program does not get stuck; either it executes the next instruction and continues, or the program attempts to reach outside its bounds of authority and subsequently crashes into a failed configuration.

The fundamental theorem is proved by Löb induction (the proof principle for guarded recursion), generalized for all W, p, g, b, e and a. The induction hypothesis states that the program will safely execute *later*, in other words after at least one step of execution. The fundamental theorem is thus proved by stepping through the execution of the next instruction, currently pointed to by a, (which may either fail or succeed), and then applying the induction hypothesis *one step later*.

Authors' addresses: Aïna Linn Georges, Aarhus University, Denmark, ageorges@cs.au.dk; Alix Trieu, ANSSI, France, alix.trieu@ssi.gouv.fr; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

First, we extract the resources needed for executing the next instruction. Since p has at least read authority over its range of authority, the assumption $\mathcal{V}(W)(p, g, b, e, a)$ grants access to the standard resources within the range $[b, e[$. In particular, if the program counter is valid, we can extract the standard resources for a, including the points-to predicate $a \mapsto w$ for some Word w. The proof then proceeds by case analysis on $decode(w)$.

Let us focus on the particularly interesting case `storeU` r_{dst} r_{src} 0 in which the register r_{dst} contains a stack capability, currently pointing to some address a_{stk}. In this case, we will be overriding some (uninitialized) word with a new possibly DIRECTED and valid word w_{src} from r_{src}. Since the offset is 0, the capability in r_{dst} will increment, thus initializing address a_{stk}. We will therefore have to update the standard resource for a_{stk} to an initialized standard resource, now pointing to the word w_{src}. In particular, since this is a stack address, we will have to prove that w_{src} satisfies the conditions of a stack standard resource, namely that w_{src} is monotone with regards to $\sqsupseteq^{a_{stk}}$. We first use Lemma 1.1, which asserts that w_{src} is monotone relative to the upper bound of its own read authority. Storing a DIRECTED capability on the stack will dynamically check that this upper bound is *smaller* than the current stack address, in other words below a_{stk}. We can therefore apply Lemma 1.2 to assert that w_{src} is indeed monotone with regards to $\sqsupseteq^{a_{stk}}$.

The remainder of the cases are similar. Instructions that interact with memory require invariants to be opened, whereas instructions that only change the register state will only have to establish the validity of updated registers. □

2 COMPONENTS, LINKING AND CONTEXTS

$$
\begin{aligned}
ms \in \text{MemFrag} &::= \text{Addr} \rightharpoonup \text{Word} \\
\overline{a \leftarrow\!\shortmid\ s} \in \text{Imports} &::= \overline{\text{Symbols} \times \text{Addr}} \\
\overline{s \mapsto w} \in \text{Exports} &::= \text{Symbols} \rightharpoonup \text{Word} \\
basecomp \in \text{BaseComp} &::= \text{Memfrag} \times \text{Imports} \times \text{Exports} \\
\text{Comp} &::= basecomp \mid (basecomp, c_{main})
\end{aligned}
$$

(a) Components.

$$
\frac{\begin{array}{c} comp_1 = (ms_1, \overline{a \leftarrow\!\shortmid\ s}^1, \overline{s \mapsto w}^1) \\ comp_2 = (ms_2, \overline{a \leftarrow\!\shortmid\ s}^2, \overline{s \mapsto w}^2) \qquad comp = (ms, \overline{a \leftarrow\!\shortmid\ s}, \overline{s \mapsto w}) \qquad \overline{s \mapsto w} = \overline{s \mapsto w}^1 \cup \overline{s \mapsto w}^2 \\ ms = (ms_1 \uplus ms_2)[a \mapsto w \mid a \leftarrow\!\shortmid\ s \in (\overline{a \leftarrow\!\shortmid\ s}^1 \cup \overline{a \leftarrow\!\shortmid\ s}^2), s \mapsto w \in \overline{s \mapsto w}] \\ \overline{a \leftarrow\!\shortmid\ s} = (\overline{a \leftarrow\!\shortmid\ s}^1 \cup \overline{a \leftarrow\!\shortmid\ s}^2) \setminus \{\_ \leftarrow\!\shortmid\ s \mid s \mapsto w \in \overline{s \mapsto w}\} \end{array}}{comp = comp_1 \bowtie comp_2}
$$

(b) Linking.

Fig. 1. Components and linking.

A closed program is created by linking multiple components. A component (defined in Fig. 1a) is either a base component representing a library waiting to be linked with other components. A main component is a base component with a main entrypoint. A base component is defined by a memory fragment representing the part of the memory owned by the component and its content, a list of imports indicates where a word associated with a symbol will be stored by the linker, and a list of exports associating words with symbols.

Linking components $comp_1$ and $comp_2$, noted $comp_1 \bowtie comp_2$, is defined in Fig. 1b. Two components can only be linked if they have disjoint memories, and at most one of them is a main component. The new component $comp_1 \bowtie comp_2$ is obtained by taking the union of both components' memory fragments where imports satisfied by an export have the memory updated with the exported word on the imported address. The satisfied imports are removed from the resulting linking, and the exports are obtained by combining the components' exports.

We can now define a notion of context and what is a closed program.

Definition 2.1 (Closed programs and contexts). A main component is a closed program when its imports list is empty.

A component C is a context for component $comp$, written $C[comp]$ when $comp \bowtie C$ is a closed program.

We now need to define how to run a closed program, this is done by defining its initial state. However, we are only interested in well-formed programs.

Definition 2.2 (Well-formed components). A base component $(ms, \overline{a \leftarrow\!\shortmid\ s}, \overline{s \mapsto w})$ is well-formed when:

- import and export symbols are disjoint;
- all exported words can only address memory owned by the component, i.e., ms;
- all import addresses are part of the memory owned by the component;
- ms only contains integers or capabilities that can address memory within ms, are not permit-write-local (i.e., have permission URWL, URWLX, RWL or RWLX) and are global;

- ms is disjoint from the memory reserved for the call stack.

A main component is well-formed when its base component is well-formed and its main entrypoint can only address memory it owns.

These properties are required so that initial assumptions of the capability machine are not violated. For instance, stack safety cannot be enforced if any of the components already have direct access to the call stack or can keep a copy of stack derived capabilities in their own memory by having permit-write-local capabilities.

Finally, assuming that $[b_{stk}, e_{stk}[$ is the range reserved for the stack, the initial configuration of a closed program $((ms, [], \overline{s \mapsto w}), c_{main})$ is a configuration (reg, m) in the base semantics such that:

- $reg(\text{pc}) = c_{main}$;
- $reg(r_{stk}) = (\textsc{urwlx}, \text{Directed}, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq \text{pc}$ or $r \neq r_{stk}$;
- $mem(a) = ms(a)$ if $a \in \text{dom}(ms)$, and $mem(a) = 0$ otherwise.

In the overlay semantics, the initial configuration of a closed program $((ms, [], \overline{s \mapsto w}), c_{main})$ is the configuration $(reg, ms, \emptyset, [])$ such that:

- $reg(\text{pc}) = c_{main}$;
- $reg(r_{stk}) = \text{Stk}(0, \textsc{urwlx}, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq \text{pc}$ or $r \neq r_{stk}$.

3 DEFINITION OF SAFE WORD

In the definition of ExecCall, arguments must be words that are *safe* to pass. This notion is defined as follows.

Definition 3.1 (Safe word). Given an overlay configuration $\varphi = (reg, h, stk, cs)$, a word w is *safe* to pass as argument in that configuration, written $safe(\varphi, w)$ if one of the following holds.

- w is an integer ($w \in \mathbb{Z}$);
- w is a heap capability (p, ℓ, b, e, a);
- w is a stack derived capability $\mathrm{Stk}(d, p, b, e, a)$ such that
  - $d < |cs|$ and cs is of the form $(reg_n, stk_n) :: \cdots :: (reg_0, stk_0)$ and $reg_d(r_{\mathrm{stk}}) = \mathrm{Stk}(d, p', b', e', a')$ and $e < a'$ and for all $a'' \in [b, \mathrm{canReadUpTo}(w)[$, $safe(\varphi, stk_d(a''))$;
  - or $d = |cs|$ and the above conditions hold using $reg_d = reg$ and $stk_d = stk$ instead.

Recall that, intuitively, a word is safe to pass if it cannot tamper the activation code of active return capabilities. As such, integers and heap capabilities are obviously safe. Return capabilities are unsafe to pass as this would break WBCF, though allowing to pass them would actually implement a form of `longjmp`. The activation code of return capabilities is stored on the stack, just above the stackframe used by a function, hence the $e < a'$ condition above. We must also ensure that the capability can only give access to safe words itself.

We point out that while the definition may seem not well-founded as a stack capability can only be safe if its contents are also safe, which may itself contain the original capability at first glance. This is actually not the case as stack capabilities are *directed capabilities*, and this definition provides a terminating algorithm. Indeed, in the recursive check $safe(\varphi, stk_d(a''))$, it is either a word that is immediately safe (i.e., an integer or a heap capability), or immediately unsafe (a return capability), or a stack capability w' such that $\mathrm{canReadUpTo}(w') \leq a''$ (by property of directed capabilities) and $a'' < \mathrm{canReadUpTo}(w)$ (by definition of a''). There is therefore a decreasing measure ensuring termination of the check.

In the Coq formalization, we use a more *restrictive* notion of safe words that instead only allow passing integers and heap capabilities.