Toward Complete Stack Safety for Capability Machines

Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Lars Birkedal
Aarhus University
Denmark
{ageorges,armael,alix.trieu,birkedal}@cs.au.dk

Abstract

Capability machines are computers that provide support for fine grained control over memory accesses. Pointers are replaced by capabilities, unforgeable tokens of authority that represent the ability to access a memory location. As such, capability machines are an attractive target for secure compilation, and this interest is further compounded by the recent commitment from Arm to develop an industrial prototype of CHERI-based capability machines[1]. It is thus no surprise that numerous recent works have proposed techniques for enforcing well-bracketed control-flow (WBCF) and local state encapsulation (LSE) [4, 6, 7], temporal stack safety [8], or temporal heap safety [3]. However, these solutions still fall short of ensuring what we believe to be *complete* stack safety. In this paper, we review recent propositions from the literature and identify limitations. We further propose a potential solution using a new form of capabilities.

1 Background

Capability machines are computers that provide support for fine grained control over memory accesses. Pointers are replaced by capabilities, unforgeable tokens of authority that represent the ability to access a memory location. Capabilities are represented by a range of authority (e.g., $[b, e)$), a permission (e.g., read, read-write, read-write-execute, etc), and the current address it points to, thus the capability (p, b, e, a) provides permission p over range $[b, e)$ and currently points to address a.

Capability machines are an attractive target for secure compilation as illustrated by numerous recent works showing how to enforce WBCF and LSE [4, 6, 7], temporal stack safety [8], or temporal heap safety [3].

We start by reviewing what guarantees on stack memory can be enforced by the different secure calling conventions proposed in the literature, but omit how these calling conventions operate due to space constraint.

Consider the code in Listing 1. This is a variant of the classical "awkward example" [2], which works as follows. Function `f` possesses some local state modeled by the variable `x` (line 3), which is set to `0` before calling some arbitrary adversarial code `adv()`. After the call returns, `x` is set to `1` before calling the adversary again. Finally, `x` is checked to be still equal to `1` at the end. This example relies heavily on WBCF and LSE for the assertion to succeed. If LSE is not ensured, then the second call to `adv` may modify `x`. Similarly, if WBCF is not enforced, then during its second call on line 7, `adv` could keep the return pointer to line 8, call `f`, which would then set `x` to `0` before calling `adv` again who can finally use the return pointer to line 8 and fail the assertion.

Using a secure calling convention as proposed by Georges et al. [4], Skorstengaard et al. [6, 7], it can be verified that the assertion does not fail in presence of arbitrary code. However, the calling convention proposed by Skorstengaard et al. [7] requires a prohibitive amount of memory clearing which makes it hardly usable in practice. Georges et al. [4] improves on this by introducing *uninitialized* capabilities, a new kind of capabilities that represent the permission to read, only after having written first. Leveraging uninitialized capabilities, the amount of required memory clearing is reduced from a quadratic amount to a linear one. Finally, StkTokens [6] is a calling convention proposing to use *linear* capabilities, which cannot be duplicated. The calling convention requires no memory clearing, but it may be difficult to efficiently implement linear capabilities in hardware, and some popular programming idioms (e.g., passing stack references in calls) may not be possible to implement [5, §3.6.2].

In another direction, Tsampas et al. [8] study the issue of temporal stack safety. Consider the code in Listing 2, `&x` on line 12 is a pointer to a location containing another pointer. After the call to `f`, there is now a pointer at `&x` to the location `l` previously occupied by `z` on line 11. The value of `l` depends on a global variable `N`. It should be noted that `l` is *stale* after the return and should not be allowed to be passed down. Nevertheless, `l` is passed to `h` through `g`. For well chosen values of `K` and `N`, it is possible that `l` coincides with where the return pointer of `h` is stored and thus the store at line 2 can lead to the control-flow being hijacked.

To solve this issue, Tsampas et al. propose that capabilities are extended with "lifetime" information, basically the call depth of the function's stackframe, and that capabilities with longer lifetime may not be used to store a capability with shorter lifetime. This would disallow the store on line 9 in the example. The main difficulty in implementing this proposal is that in order to allow for a call depth of size 2^n, n bits are required in the encoding of a capability. This is possibly expensive as it would already require 10 bits just to allow to use `List.map` on a list with 1000 elements.

[1] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html

PrisC'21, January 17, 2021, Online
2021.

```
void adv(void);
void f(void) {
  static int x = 0;
  x = 0;
  adv();
  x = 1;
  adv();
  assert (x == 1);
}
```

Listing (1) Awkward example

The examples are given in a C-like syntax, but we actually consider the underlying assembly code.

```
int N, K;
void h(int* x) { *x = 0; }
void g(int* x) {
  char* t[K];
  h(x); }
void f(int** x) {
  char* t[N];
  int z;
  *x = &z; }
int main(void) {
  int* x;
  f(&x);
  g(x);
  return 0; }
```

Listing (2) Example violating temporal stack safety

```
void adv(int* x)
void f(void) {
  int x, y = 0;
  adv(&x);
  x = 0;
  adv(&y);
  assert (x == 0); }
int compare(char* x, char* y)
int compare_secret(char* in) {
  char* secret = ...;
  int x = compare(secret, in);
  return x; }
```

Listing (3) Issues with passing stack references

2 Complete stack safety

While combining WBCF, LSE and temporal stack safety is desirable, we believe this is not sufficient to achieve *complete* stack safety. Indeed, when considering a high level call stack with push and pop operations, we would like that when an item is pushed, the items below become inaccessible: this corresponds roughly to LSE. Only the item at the top can be popped, this corresponds to WBCF. Finally, when an item is popped, it becomes inaccessible from the stack. This last point is only partially solved by temporal stack safety.

Consider for instance that you are implementing a function that checks whether an input is equal to some secret string, and you found a formally verified piece of code that compares two strings in constant-time: it would make sense to use it as a library routine. The bottom half of Listing 3 showcases such a scenario, where `compare_secret` checks whether an input is equal to the secret by invoking the formally verified `compare`. Assuming that `compare_secret` is invoked by some adversarial code with a random input, the comparison will most likely fail. However, it turns out that the verified comparison function `compare` copies both inputs on its stack, but it does not scrub its stack before returning! On a regular machine, the adversary could recover the secret by abusing pointer arithmetic, even though intuitively, stack safety should forbid it. This is an instance of use after free.

Finally, LSE is too restrictive as it is a common programming idiom in C to pass stack references as arguments. For instance, consider `f` in Listing 3. It passes references to local variables to an adversary: `x` in a first call, then `y` in a second call. Variable `x` is set to `0` before the second call, and checked to not have been modified afterwards. The intuition[2] is that the adversary should not be allowed to keep references to `x` in between calls as it is a stack variable whose lifetime may be short; it would thus be unsafe to keep a reference to it in a global variable for instance. This can be thought of as some sort of "fine grained" local state encapsulation.

[2] As also argued by rule 17.6 of the MISRA-C guideline [1].

While these issues were not considered by previous works, they are partly solved by some of them. Thanks to its excessive memory clearing, fine grained local state encapsulation is possible and use after free is prevented by the calling convention of Skorstengaard et al. [7], but it does not support temporal stack safety. Though a vast improvement on performance, the new calling convention of Georges et al. [4] does not prevent use after free anymore. Similarly, StkTokens [6] does not prevent use after free either.

3 Monotone capabilities

The calling convention of Georges et al. [4] seems the most practical to use, but does not prevent use after free nor provide temporal stack safety. Their calling convention uses a combination of *local* capabilities and *uninitialized* capabilities. Local capabilities are capabilities that can only be stored on the stack (to simplify). An uninitialized capability with authority over range $[b, e)$ and currently pointing to address a gives read authority over $[b, a)$. When used to write to a, the boundary is automatically increased so that read authority is now given over $[b, a + 1)$. Intuitively, by making the stack capability local and uninitialized, a caller knows that any reference to its stack variables are necessarily stored on the stack, and a callee cannot read what is left on the stack by uninitializedness of the capability. However, this does not protect a callee from its caller as shown by the previous example of Listing 3. We introduce *monotone* capabilities which can be used to prevent use after free and provide temporal stack safety. Assuming that the stack is growing upward, a monotone capability is a local capability that can only be stored *above* where it has read authority. This ensures temporal stack safety as in Listing 2, by preventing passing down a reference to a stack variable to the caller. By combinining uninitialized and monotone capabilities, a callee is ensured that its caller cannot have kept a capability with read authority over its stackframe. Additionally, it is no longer necessary to scrub its stackframe, thus rendering dead store elimination less harmful [9].

References

[1] Motor Industry Software Reliability Association et al. 2008. *MISRA-C: 2004: Guidelines for the Use of the C Language in Critical Systems*. MIRA.

[2] Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 143–156. https://doi.org/10.1145/1863543.1863566

[3] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy, S&P 2020, San Francisco, CA, USA, May 18-21, 2020*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

[4] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and Provable Local Capability Revocation using Uninitialized Capabilities. *Proc. ACM Program. Lang.* POPL (2021). (Conditionally accepted).

[5] Lau Skorstengaard. 2019. *Formal Reasoning about Capability Machines*. Ph.D. Dissertation. Aarhus University.

[6] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* 3, POPL (2019), 19:1–19:28. https://doi.org/10.1145/3290332

[7] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2020. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 5:1–5:53. https://doi.org/10.1145/3363519

[8] Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 243–255. https://doi.org/10.1109/CSF.2019.00024

[9] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 1025–1040. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/yang