Zeebra: a compiler for a low-level language with IFC

Christoffer Müller Madsen
Department of Computer Science
Aarhus University
christoffer@guava.space

Jon Michael Aanes
Department of Computer Science
Aarhus University
jonjmaa@gmail.com

Aslan Askarov
Department of Computer Science
Aarhus University
aslan@cs.au.dk

Zee [1] is a low-level, imperative, information-flow–aware programming language, with runtime type and stack introspection via existential types and well-typed stack and heap pointers. The language is intended for writing secure programming language runtime environments, such as garbage collectors and schedulers. An important constraint in Zee is that the execution time of operations on high values in low contexts must be independent of the values; whether the operation is allocating on the heap, or just multiplying integers. Any operation that cannot guarantee this is not allowed in a low context, and must use a form of predictive mitigation [2].

The original Zee paper provides an interpreter for the language in Haskell. In this work, we present Zeebra – a Zee compiler for an LLVM backend. We discuss some of the design decisions and the challenges we have encountered.

I. Call-stack introspection

Zee allows for typed call-stack introspection through the special expression **FP** that returns the current frame pointer as an existentially-typed value. The example below uses **FP** to indirectly update argument x of function f to 5.

```
proc f(x : int_H) =
  let (α_args   : type_L, e_1      : _) := unroll (FP) in
  let (α_locals : type_L, e_locals : _) := e_1 in
  match α_args with
    | int_H → *(e_locals + sizeof α_locals) := 5
    | _     → skip
```

Here, types α_{locals} and α_{args} describe the function's stack frame. Value e_{locals} points to the stack frame's locals. We use pointer arithmetic on e_{locals} to access the arguments of the current stack frame; it can also be used to access previous stack frames. Zee's type system ensures this kind of call-stack introspection satisfies noninterference. It uses type witnesses to store information required to pattern match on the types.

Implementation. Zeebra stores type witnesses as tagged 64-bit pointers, storing the instance size in the upper 15 bits and the type category (heap, stack pointer, base type, etc.) in the lower 3 bits; the pointer points to auxiliary information, such as the pointed-to types for stack pointers. Type witnesses for variables are maintained by functions.

II. Existential types

Zee allows existentially typed values, even with types like $\exists a : \text{type}.\ a$ where the size of an instance of the type is unknown. Values of this type can escape from the stack frame they are allocated in [3]. This poses two challenges. First, we require the size of stack allocated existential values to be statically known. This restriction is not apparent in the Zee interpreter, because it relies on the host language's heap allocation. This restriction does not, however, reduce the expressive power, as the programmer can wrap the value in a heap allocation, and pass that around.

Second, we must be able to inspect unpacked type witnesses of existentially typed values outside the scope of their creation. At that point, storing type witness data on the stack is no longer an option, as it will be cleared on function return, resulting in dangling type witnesses. To mitigate this, we clone type witnesses to the heap when packing an existential type. We use reference counting to decide when it is safe to reclaim the type witnesses from the heap. Reference counting is sufficient because allocations are self-contained.

In cases where the label of the packed type is high, packing might serve as a timing side channel, as the entire type is copied. To mitigate this, we require that the label of the packed type flows to the label of the current PC.

III. Safe memory reuse

Zee uses identifier-based temporal checking [4] for safe memory reuse. There are two version counters: one for stack, and one for heap. The stack version counter is incremented with every new frame; the heap one is incremented with every heap allocation. Stack pointers inherit the version of the current frame when created. Dereferencing a stack pointer checks its version against the version of the frame it points into. Similar checks take place for heap pointers.

Implementation. We use a shadow stack for stack pointer versioning, and to keep the frames' version numbers out-of-band. We found it necessary to increment the stack version between loop iterations, because escaping pointers can become ill-typed due to memory reuse across loop iterations.

We also determined that the original Zee approach of keeping heap pointers in-band to be problematic, because an attacker can exploit the malloc behaviour to overwrite version numbers. Instead, we keep allocations and version numbers separated in different locations on the heap.

IV. Lattice size

We use a power-set lattice for security levels, encoding individual levels as bitfields. Because Zee heaps are segregated

based on the security lattice, we restrict the number of set elements to at most 10, to avoid a combinatorial explosion.

V. Evaluation

We have ported the timing-secure mark-and-sweep GC from the original Zee paper to Zeebra to test correct behaviour and security against timing attacks. For performance benchmarking, we have ported a C implementation [5] of the AES128 block cipher to Zee, where we observe an 8x median slowdown, over 101 runs of encrypting *War and Peace*. Compiler correctness is assured by an additional 650 unit tests.

References

[1] M. V. Pedersen and A. Askarov, "Static enforcement of security in runtime systems," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 335–350.

[2] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 99–110. [Online]. Available: https://doi.org/10.1145/2254064.2254078

[3] D. Grossman, J. G. Morrisett, T. Jim, M. W. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 282–293. [Online]. Available: https://doi.org/10.1145/512529.512563

[4] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, J. Vitek and D. Lea, Eds. ACM, 2010, pp. 31–40. [Online]. Available: https://doi.org/10.1145/1806651.1806657

[5] "Tiny AES in C." [Online]. Available: https://github.com/kokke/tiny-AES-c