Micro-Policies: Formally Verified Tagging Schemes for Safety and Security (Extended Abstract)

Cătălin Hriţcu (INRIA Paris)[1]

Today's computer systems are distressingly insecure. A host of vulnerabilities arise from the violation of known, but in-practice unenforceable, safety and security policies, including high-level programming models and critical invariants of low-level code. This project is aimed at showing that a rich and valuable set of *micro-policies* can be efficiently enforced by a new generic hardware-software mechanism and result in more secure and robust computer systems. The key idea is to add metadata to the data being processed (e.g., "this is an instruction," "this word comes from the network," "this one is private"), to propagate the metadata as instructions are executed, and to check that invariants on the metadata are enforced throughout the computation. For this we are extending a traditional hardware architecture so that every word of data in the machine is associated with a word-sized tag. In particular, tags can be pointers to arbitrary data structures in memory. The interpretation of these tags is left entirely to software: the hardware just propagates tags from operands to results as each instruction is executed, following software-defined rules.

Abstractly, the tag propagation rules can be viewed as a partial function from argument tuples of the form (*opcode*, *pc tag*, $argument_1$ *tag*, $argument_2$ *tag*, ...) to result tuples of the form (*new pc tag*, *result tag*), meaning "if the next instruction to be executed is *opcode*, the current tag of the program counter (PC) is *pc tag*, and the arguments expected by this opcode are tagged $argument_1$ *tag*, etc., then executing the instruction is allowed and, in the new state of the machine, the PC should be tagged *new pc tag* and any new data created by the instruction should be tagged *result tag*." In general, the graph of this function *in extenso* will be huge; so, concretely, a hardware *tag management unit (TMU)* maintains a cache of recently-used rule instances (i.e., input tags / output tags pairs). On each instruction dispatch, the TMU forms an argument tuple as described above and looks it up in the rule cache. If the lookup is successful, the result tuple includes a new tag for the PC and a tag for the result of the instruction (if any); these are combined with the ordinary results of instruction execution to yield the next machine state. Otherwise, if the lookup is unsuccessful, the hardware invokes a *cache fault handler*—a trusted piece of system software with the job of checking whether the faulting combination of tags corresponds to a micro-policy violation or whether it should be allowed. In the latter case, an appropriate rule instance specifying tags for the instruction's results is added to the cache, and the faulting instruction is restarted. As already mentioned above, the hardware is generic and the interpretation of micro-policies is programmed in software, with the results cached in hardware for common-case efficiency.

Many useful dynamic enforcement mechanisms for safety and security fit this micro-policy model:

- fine-grained dynamic information flow control (IFC);
- kernel protection;
- hardware types;
- memory safety;
- control-flow integrity;
- call stack protection;
- software-based fault isolation;
- closures (i.e., first-class functions);
- linear pointers (guaranteeing absence of aliasing);
- pointer permissions (like "readable" or "callable");
- dynamic sealing and trademarks;
- dynamic type tags (e.g., for C or Scheme);
- higher-order contracts;
- data race detection;
- taint tracking;
- process isolation;
- user-defined metadata.

When seen from the highest level a micro-policy is the combination of a set of tags, a set of rules, and a set of monitor operations. Tags are often drawn from a simple datatype, for instance a list of principals for an IFC micro-policy. As discussed above, the rules are just a convenient description of a partial function from a tuple of tags to another tuple of tags. They are used to monitor program execution and determine how tags are propagated. The *monitor operations* are trusted routines that can be invoked by user programs and that can internally perform privileged instructions such as freely inspecting and changing tags (they are similar to system calls in an operating system). For instance, declassifying a secret piece of information is a monitor operation in the IFC micro-policy. While rules are functional and are cached by the TMU, monitor operations can be stateful. For instance, memory allocation and freeing could be the operations of a memory safety micro-policy.

The goal of this ongoing work is to come up with a clean formal framework in Coq for expressing, composing, and verifying arbitrary micro-policies, and to instantiate this framework on a diverse set of interesting examples.

[1]Joint work with Arthur Azevedo de Amorim (UPenn and INRIA), Nathan Collins (Portland State), André DeHon (UPenn), Delphine Demange (IRISA), Maxime Dénès (UPenn), Udit Dhawan (UPenn), Nick Giannarakis (NTUA and INRIA), Leonidas Lambropoulos (UPenn), David Pichardie (IRISA), Benjamin C. Pierce (UPenn), Randy Pollack (Harvard), Antal Spector-Zabusky (UPenn), and Andrew Tolmach (Portland State).