Micro-Policies: Formally Verified Low-Level Tagging Schemes for Safety and Security

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@gmail.com⟩

INRIA Team: Prosecco

Location: 23 Avenue d'Italie, Paris, France

Language: English (no French)

Existing skills or strong desire to learn:

- formal verification in the Coq proof assistant,
- programming languages theory,
- functional programming.

Context

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 programs. This project[1] 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 working on extending a traditional hardware architecture so that every word of data in the machine (whether in memory or in registers) is associated 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 (in parallel with the logic implementing the usual behavior of the instruction—e.g.,

[1] This *micro-policies* project is a followup of CRASH/SAFE (`http://www.crash-safe.org/`) and a collaboration [4] with University of Pennsylvania, Portland State University, Harvard University, and the Celtique joint team in Rennes (INRIA, CNRS, University of Rennes 1, and ENS).

addition), 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:

- stack and kernel protection (see section 1);
- basic type and memory safety (see section 1);
- fine-grained dynamic information flow control (IFC) [4, 8, 9];
- control-flow integrity (CFI) [2];
- data race detection [11];
- linear pointers, guaranteeing absence of aliasing;
- marking pointers with permissions like "readable", "writeable" "jumpable", or "callable";
- closures (i.e., first-class functions, or code pointers together with protected local environments);
- dynamic sealing [10, 13] and trademarks [10] (the latter can for instance be used to cache the result of dynamic contracts);
- user-defined metadata (i.e., providing operations for setting and getting metadata to values);
- dynamic type tags (e.g., for C or Scheme);
- higher-order contracts [7] (recent proposals [6] use sophisticated mechanisms for tracking components and assigning blame that could probably be encoded as tags);
- taint tracking;
- isolation (e.g., replacement for virtual memory).

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 operations. Tags are often drawn from a simple datatype, for instance a list of principals for an IFC micro-policy [4]. 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 *operations* are pieces of privileged code that can be invoked by user programs, while internally performing dangerous operations 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 an operation in the IFC micro-policy. While rules are functional and are cached by the TMU, operations can be stateful. For instance, memory allocation and freeing could be the operations of a memory safety micro-policy (see section 1).

The goal of this proposal is to come up with a clean formal framework in Coq [1] for expressing (section 2), composing (section 3), and verifying (sections 1, 2, and 3) arbitrary micro-policies, and to instantiate this framework on a diverse set of interesting examples (initially on the low-level micro-policies in section 1, and later on any of the ones above). This ambitious goal can naturally be split into several internship or thesis topics focusing on specific aspects.

1 Low-level Safety Micro-Policies

We are currently devising micro-policies for *stack and kernel protection*, as well as *(dynamic) basic type and memory safety*, for a simplified model of a conventional processor extended with tags and a TMU.

Example: memory safety micro-policy For instance, the memory safety micro-policy uses fine-grained tags to detect all spatial (e.g., accessing an array out of its bounds) and temporal (e.g., referencing through a pointer after the region has been freed) memory safety violations involving heap-allocated data. Such violations are a common source of serious security vulnerabilities [3, 14] (e.g., stack- and heap-based overflows, respectively exploitable use-after-free and double-free bugs). Intuitively, for each new allocation we make up a fresh *block id* `n` and use `n` in the tag of each memory location in the newly created memory block (*à la* `memset`). The pointer to the new block also has `n` in its tag. Later, when we dereference through a pointer, we check that its block id is the same as the block id on the memory cell to which it points. When a block is freed, the tags on all its cells are changed to a constant tag representing non-reference-able memory. If we additionally use tags to distinguish between pointers and integers, we can basically turn pointers into "unforgeable capabilities" for the memory regions that they point to, in the sense that only one who was explicitly passed a pointer to a region can read or write its contents.

More precisely, the memory safety micro-policy uses three kinds of tags: one for values (`V`), another one for currently allocated memory locations (`M`), and another one for free memory locations (`F`; we assume that the entire memory starts out tagged `F`). The structure of tags is captured by the following datatype:

```
data Tag = V(Type) | M(n,Type) | F
```

Both values and memory locations have an associated `Type`. There are only two `Type`s: integers (`I`) and pointers (`P`).

```
data Type = I | P(n)
```

As explained above, the tag of memory locations and the type of pointers includes a block identifier `n`, which in our case is simply the address where the corresponding memory block starts.

The allocation operation first does the actual allocation as usual (*à la* `malloc`) obtaining an address `n` and then fills the new block with zeroes tagged `M(n,I)` (*à la* `memset`) and returns the value `n` tagged with `V(P(n))`. The deallocation operation first updates its internal data structures as usual and then tags the entire deallocated block with `F`.

In order to define the rules for this policy we first define a simple domain specific language (DSL) of tag expressions `TE`:

```
data TE = TAG1 | TAG2 | TAG3 | TAGPC | V(TYPE) | M(NE,TYPE)
data TYPE = I | P(NE) | GET_MT(TE)
data NE = GET_MN(TE) | GET_VPN(TE)
```

The expressions `TAG1`, `TAG2`, `TAG3`, and `TAGPC` are variables referring to the corresponding part of the input tag tuple. Tag expression `V(TYPE)` constructs a value tag given a type expression `TYPE`. Similarly, expression `M(NE,TYPE)` constructs a memory tag given a numeric expression `NE` and a type expression `TYPE`. The definitions of `TYPE` and `NE` are straightforward: `GET_MN(TE)` and `GET_MT(TE)` simply deconstruct a tag of the form `M(n,Type)` into `n` and `Type`, while `GET_VPN(TE)` returns `n` given a pointer value tag of the form `V(P(n))`.

We also define a set of boolean expressions (`BE`) that are used to express conditions that are tested by the rules in order to decide whether the policy was violated or not:

```
data BE = TRUE | IS_V(TE) | IS_VI(TE) | IS_VP(TE) | IS_M(TE) |
          NE1 = NE2 | BE1 AND BE2 | BE1 OR BE2
```

The symbols starting with `IS` test whether the tag has a certain shape; the other constructions are straightforward.

We are now ready to define the memory safety micro-policy's rule table in terms of this DSL. This table corresponds only to 3 instructions (the simplest nontrivial machine might have a dozen), whose opcodes are listed in the first column. The second column contains a `BE` that decides whether the instruction should be allowed to continue or not (in case it violates the policy). The third column contains a `TE` that calculates the tag of the result.

Opcode	Allow	Result
`ADD`	`IS_VI(TAG1) AND IS_VI(TAG2)`	`V(I)`
`ADD`	`IS_VP(TAG1) AND IS_VI(TAG2)`	`V(P(GET_VPN(TAG1)))`
`ADD`	`IS_VI(TAG1) AND IS_VP(TAG2)`	`V(P(GET_VPN(TAG2)))`
`LOAD`	`IS_VP(TAG1) AND IS_M(TAG2)`	
	`AND GET_VPN(TAG1)=GET_MN(TAG2)`	`V(GET_MT(TAG2))`
`STORE`	`IS_VP(TAG1) AND IS_V(TAG2)`	
	`AND IS_M(TAG3)`	
	`AND GET_VPN(TAG1)=GET_MN(TAG3)`	`M(GET_VPN(TAG1),GET_MT(TAG2))`

Addition (`ADD`) is only allowed on two integers, in which case the result is also an integer, or between an integer and a pointer, in which case the result is a pointer with the same block id. This allows pointers to be offset of outside their blocks, but the rules for `LOAD` and `STORE` prevent out-of-bounds pointers from being accessed. The rule for `LOAD` requires that the first argument is a pointer tagged `V(P(n))` (`TAG1`) and the memory location referenced by this pointer is tagged `M(n,Type)` (`TAG2`), for the same block id `n`. If these conditions are satisfied then the `LOAD` is allowed and the resulting value is tagged `V(Type)`. Similarly the rule for `STORE` requires that the first argument to the store is tagged `V(P(n))` (`TAG1`), that the second argument is tagged `V(Type)` (`TAG2`), and that the memory location referenced by the first argument is tagged `M(n,Type)` (`TAG3`), again for the same block id `n`. If these conditions are satisfied then the `STORE` is allowed and the written memory location is tagged `M(n,Type)` afterwards.

Goal The main goal here is to formally verify, in Coq [1], the correctness of such low-level safety micro-policies with respect to a higher-level abstract machine. Verification can make use of our previous ideas on refinement and verified structured code generators [4]. The proof will be structured as a series of stepwise refinements targeting increasingly higher-level abstract machines, where in each step the kernel virtualizes the services of the TMU, so that the higher-level micro-policies can be stacked on top. The topmost abstract machine will thus not only have a protected stack and kernel, a mechanism for invoking operations, a structured memory model, and a separation between pointers, integers, and instructions, but it will also provide all the necessary mechanisms for supporting even higher-level micro-policies. This vertical proof organization will illustrate one interesting way to do micro-policy composition (this subject is further discussed in section 3). Finally, the low-level micro-policies developed here will also inform the design of the general metalanguage for defining micro-policies discussed in section 2.

2 Metalanguage for Micro-Policies

In order to come up with micro-policies, we need a way to specify the rules that the TMU must execute while enforcing them. One of our goals is to come up with a declarative language for micro-policies that can be compiled into an executable form. The advantage of using a higher-level language is that it reduces the need for programming against the TMU's interface directly; most of the low-level interaction can then be handled automatically. In prior work [4], we have experimented with such a language for IFC micro-policies. In this project we want to generalize this to a language that can describe arbitrary micro-policies, not just IFC ones.

At a high level, each micro-policy is defined by a table containing symbolic rules (e.g., one for each instruction). Each rule contains several symbolic expressions drawn from a free algebra over a set of function symbols. The algebra is sorted (i.e., typed) and the sort signature of each function symbol is also part of the policy. The variables in expressions refer to the inputs to form symbolic expressions and they are also sorted. We associate a Coq function of the right type to each function constant to obtain an abstract semantics for the rule table. We obtain a concrete implementation by devising a compiler that is parameterized by a mapping from function symbols to machine code. This compiler as well as the machine code associated to function symbols can be defined using verified structured code generators [4, 5]. The final goal is to prove a generic correctness theorem stating that any table is correctly compiled to machine code (i.e., the behavior of the produced machine code is the same as specified by the abstract semantics specified in Coq) provided that each function symbol is correctly implemented. Additionally, the correct implementation of the function symbols needs to be proved for each individual micro-policy.

More ambitiously, we would not only define a language for rules, but for complete micro-policies. As explained in section and exemplified in section 1 a micro-policy contains 3 things: tags, rules, and operations. Ideally we should be able to define all three in a high-level metalanguage, and compile it down to machine code. For high-level enough micro-policies we shouldn't need to write any machine code, everything should be generated from high-level descriptions. Moreover, it should be possible to reason about these descriptions directly in the logic of Coq, without the need of redefining the operations in the logic. The challenge is to devise a language that is expressive enough to implement such operations yet still simple enough so that writing a verified compiler that is fully integrated with Coq's formalism is doable (e.g., a simple lambda calculus with datatypes).

3 Micro-Policy Composition

Since the TMU architecture is very general, we would like to enforce a variety of micro-policies simultaneously, and our decision to represent tags as word-sized values supports this: We can consider tags that are pointers to a *tuple* of more primitive tags. So, naively, we can imagine enforcing several micro-policies at once simply by having the rule cache miss handler evaluate each of their rules separately and combine the results. Unfortunately, in general, micro-policies are not orthogonal. For example, if we wish to enforce IFC together with other policies, observing the tags of these other policies via operations can reveal sensitive information and thus break the noninterference property established for the IFC micro-policy in isolation. The goal here is to design a composition mechanism that allows such interactions to be handled correctly.

There are several ways one could approach micro-policy composition, two of which are explained below. The first approach builds on the *vertical sequential composition* idea introduced in section 1. In this case a linear order is carefully chosen in advance between the micro-policies to be composed. One starts with the lowest-level micro-policy and the bare hardware and proves that the policy is correct with respect to a higher-level abstract machine. This higher-level abstract machine virtualizes the tagging mechanisms in the hardware so that the second micro-policy can be implemented on top of this first abstract machine, instead of the bare hardware. We then prove the correctness of this second micro-policy with respect to a second abstract machine, and so on. While this technique is likely to work well for one-off proofs, it is very much dependent on the set of composed policies and the initially chosen order between them. If we want to add or remove policies we can only do that at the top of the stack; otherwise we have to redo a large number of proofs.

The second approach could be called *parallel composition* or *cross-product composition*. In the absence of operations all micro-policies are orthogonal (they are basically just special reference monitors [12]), so we should be able to devise a way to prove trace properties separately for each micro-policy and then compose these proofs to obtain a proof that all these trace properties still hold for the composed micro-policy. The big advantage is that micro-policies proved in this setting can be freely composed. The big limitation is that many micro-policies do have operations, and as discussed above this can make them non-orthogonal. As mentioned above, IFC interacts with any other micro-policy having operations that expose tags, since this makes tags themselves information-flow channels through which secrets can be leaked. One idea here is to require each micro-policy to specify what should happen with its own tags on all other micro-policies' operations. Basically, all other micro-policies' operations become "virtual instructions" for each micro-policy machine (thus the name cross-product composition). For instance, the IFC micro-policy could specify that the result of a operation that returns a value's tag in another micro-policy is as classified as the original value. For this kind of cross-product composition there is additional proof effort when composing policies, and this effort scales quadratically with the number of composed policies. Moreover, because micro-policies can influence execution in more subtle ways than just stopping it, we are outside the well-studied realm of trace properties [12], and need to find another composable formalism for defining the enforced properties.

References

[1] *The Coq Proof Assistant*, 2012. Version 8.4.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 13(1), 2009.

[3] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook, 2nd Edition: Discovering and Exploiting Security Holes*. Wiley, 2007.

[4] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *POPL*, 2014. To appear.

[5] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. *ICFP*. 2013.

[6] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2011.

[7] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th International Conference on Functional Programming*. 2002.

[8] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. *IEEE S&P*. 2013.

[9] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*, 2013.

[10] J. H. Morris, Jr. Protection in programming languages. *CACM*, 16(1):15–21, 1973.

[11] S. Savage, M. Burrows, G. Nelson, , P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[12] F. B. Schneider. Enforceable security policies. *ACM Transactions of Information Systems Security*, 3(1):30–50, 2000.

[13] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. 2004. Full version in *Theoretical Computer Science* 375 (2007), 169–192.

[14] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE S&P*. 2013.