Efficient Secure Compilation Using Micro-Policies

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

Institition: INRIA Paris-Rocquencourt, Prosecco Team

Location: 23 Avenue d'Italie, Paris, France

Language: English

Existing skills or strong desire to learn:

- formal verification in the Coq proof assistant,
- functional programming (e.g. OCaml or Haskell)
- optional: security, compiler construction, programming languages theory

Research Context

Today's computer systems are distressingly insecure, but many of their vulnerabilities can be avoided if low-level code is constrained to obey sensible safety and security properties. Ideally, such properties might be enforced statically, but for obtaining pervasive guarantees all the way to the level of running machine code it is often more practical to detect and prevent violations dynamically using a *reference monitor* (Anderson, 1972; Erlingsson and Schneider, 2000; Schneider, 2000). Decades of work have produced a large body of knowledge about monitoring techniques that can reduce the vulnerability of computers and software to malicious subversion by providing sanity checks that current hardware lacks (e.g., "array accesses should always be in-bounds," "this instruction should never be the target of a *Jump*,"). Many simply enforce language abstractions that current software and hardware omit, e.g., memory safety or control-flow integrity (CFI); these policies can be automatically applied to all code with no additional programmer burden. Others are policies a system architect might like to apply to security-critical components without auditing or rewriting every line of code in a larger system (e.g., "un-sanitized data from the network should not be treated as a pointer," "private data should be encrypted before flowing to I/O devices,"). Still others offer primitives that system designers can use to protect critical data, track confidentiality and integrity, or isolate untrusted components.

Reference monitors are sometimes implemented in software, but this can significantly degrade performance and/or cause designers to settle for rough approximations of the intended policy that are potentially vulnerable to attack (Davi et al., 2014; Göktaş et al., 2014

When viewed abstractly, each micro-policy consists of (i) sets of *metadata tags* that are used to label every piece of data in the machine's memory and registers (including the *pc*); (ii) a *transfer function* that, given the current opcode and the tags on the current instruction, the *pc*, and the instruction operands, specifies how the *pc* and the instruction's result should be tagged in the next machine state; and (iii) a set of *monitor services* that can be invoked by user code. For example, in a micro-policy for "dynamic sealing" (a language-based protection mechanism in the style of perfect symmetric encryption (Morris, 1973)) the set of tags used for registers and memory might be $\{Data, Key(k), Sealed(k)\}$, where *Data* is used to tag ordinary data values, *Sealed(k)* is used to tag values sealed with the key k, and a value tagged *Key(k)* denotes a key that can be used for sealing and unsealing values. The transfer function for this micro-policy would allow, for example, arithmetic operations on values tagged *Data* but deny them on data tagged *Sealed* or *Key*. Monitor services are provided to allow user programs to create new keys and to seal and unseal data values with given keys.

Our proposed hardware support for micro-policies, called the *Programmable Unit for Metadata Processing (PUMP)* (Dhawan et al., 2015), is designed as an add-on to a conventional RISC processor. Conceptually, every word of data on the machine is associated with a *tag*, a full machine word that can, in particular, hold a pointer to an arbitrary data structure in memory. The interpretation of tags is left entirely to software; the hardware simply propagates tags from operands to results according to software-defined *rules*. To propagate tags efficiently, the processor is augmented with a *rule cache* that operates in parallel with instruction execution. On a rule cache miss, control is transferred to a trusted *miss handler* which, given the tags of the instruction's arguments, decides whether the current operation should be allowed and, if so, computes appropriate tags for its results. It adds this set of argument and result tags to the rule cache so that, when the same situation is encountered in the future, the rule can be applied without slowing down the processor.

This flexible hardware/software mechanism can efficiently implement a wide range of different micro-policies, singly or in combination. It achieves the flexibility and adaptability of software with performance comparable to dedicated hardware. Hardware simulations (Dhawan et al., 2015

overhead. One possible compiler target is a coarse-grained trusted-untrusted protection micro-policy, similar to the monitor self-protection mechanism from Azevedo de Amorim et al. (2014b), leading to a compiler similar to the one of Agten et al. (2012). A more interesting target is a micro-policy directly tailored towards the abstractions of the high-level language: the composition of control-flow integrity, stack protection, fine-grained memory protection, etc. Devising and proving the correctness of this composite policy would be interesting on its own. We will prove full abstraction in Coq for both these compilers using the proof technique of Jeffrey and Rathke (2005). We will also compare the two compilers to each other and document the conceptual differences.

Once this fist goal is achieved there are several possible directions for extension. We will try to achieve protection between an unbounded number of mutually distrustful components, even dynamically created ones. We should be able to use the compartmentalization micro-policy from Azevedo de Amorim et al. (2014b) to achieve this property; stronger than the trusted-untrusted separation of Agten et al. (2012). We will try to extend the source language with dynamic allocation, following Jeffrey and Rathke (2005) and Patrignani et al. (2015). We will try to obtain similar results for the untyped λ-calculus, in which dynamic allocation is implicit at the source level. We will also try to evaluate the runtime overhead of the various compilers on micro-benchmarks showing the best case, the worst case, and some simple common cases. Finally, to obtain efficient compiled code, it may be interesting to consider hybrid static / dynamic enforcement strategies. For this we could start by studying a compiler from the simply typed λ-calculus.

Even for such simple languages there are several expected challenges, for instance: (i) We need to formally define the class of contexts we want to protect against. While this class will be very broad, it will still need to distinguish the untrusted context from the compiled program. (ii) Obtaining end-to-end full-abstraction proofs can be challenging, and will require changes to our proof methodology for micro-policies (Azevedo de Amorim et al., 2014a), which is currently based on the weaker notion of refinement. The proof technique by Jeffrey and Rathke (2005) could be useful at bridging this gap. (iii) More realistically, we would need to secure the runtime system used by the code we produce. High-level languages (λ-calculus included) normally rely on a garbage collector, programs in all languages rely on library code, concurrent programs rely on a scheduler, and all these components also have to be protected from the untrusted context. To keep the effort level reasonable, we will carefully avoid this last set of issues for this internship: assume a system with allocation but no deallocation or garbage collection, assume only statically linked libraries, consider a single-threaded system, etc.

References

M. Abadi. Protection inprogramsming-3502 0 0cO52(the5silatioes)]TJ0 g 0 G (n.)-468In.

Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society, 2000.

C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384. ACM, 2013.

E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In S. Sagiv, editor, *14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2005.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. To appear in ACM TOPLAS, 2015.

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

G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.