An Efficient Software Protection Scheme
(Abstract)

Rafail Ostrovsky, MIT

In 1979 Pippenger and Fischer [PF] showed how a two-tape Turing Machine whose head positions (as a function of time) are independent of the input, can simulate, on-line, a one-tape Turing Machine with a logarithmic slowdown in the running time. We show a similar result for random-access machine (RAM) model of computation. In particular, we show how to do an on-line simulation of arbitrary RAM program by probabilistic RAM whose memory access pattern is independent of the program which is being executed with a poly-logarithmic slowdown in the running time.

A main application of our result concerns *software protection*, one of the most important issues in computer practice. A theoretical formulation of the problem for a generic one-processor, random-access machine (RAM) model of computation was given by Goldreich [G]. In this paper, we present a simple and an efficient software protection scheme for this model. In particular, we show how to protect any program at the cost of a poly-logarithmic slowdown in the running time of the protected program, previously conjectured to be impossible.

Software is very expensive to create and very easy to steal. "Software piracy" is a major concern (and a major loss of revenue) to all software-related companies. Software pirates borrow/rent software they need, copy it to their computer and use it without paying anything for it. The question of how one prevents a pirate from illegal copying of software is a question of "software protection". Ad-hoc methods have been used for decades, but only recently, a precise formulation of the problem and a solution to it was given by Goldreich [G]. Current work builds on work started in [G], making the software protection scheme more efficient and using weaker assumptions.

Let us examine various options which any software company has when considering how to protect its software. On one hand, it can sell a *physically shielded* computer with all the software installed, which self-destructs if ever opened. Clearly, such a "solution" eliminates the problem of software piracy at the price of forcing customer to purchase a new computer for each new task. We consider such a "solution" infeasible and contradictory to a general-purpose machine paradigm. One would want to sell just the software, which runs on any general-purpose computer, but which is impossible to copy. This, however, is unachievable: if one sells software which runs on any general-purpose computer, the software can always be duplicated. Thus, we always need some physically-shielded hardware to prevent the duplication. What is the minimal amount of protected hardware one needs?

We require only a constant number of registers to be physically protected. That is, only a single *chip* with a fixed number of registers is protected while the entire memory is open to the pirate, who can inspect it and alter it, in order to learn something about the protected program. Thus, we assume that a physically shielded chip is connected to a random access memory (RAM) to which the pirate has a complete read/write access.

The next question we turn to is the interpretation of "software protection". Let us consider the following hypothetical situation: suppose you are a software producer selling a protected program which took you an enormous effort to write. Your competitor purchases

G. Brassard (Ed.): Advances in Cryptology - CRYPTO '89, LNCS 435, pp. 610-611, 1990.

your program, experiments with it widely and learns some partial information about your implementation. Intuitively, if the information he gained through experimentation with your protected program simplifies his task of writing a software package, then we consider the protection scheme to be insecure. Thus, the software protection must hide *all* the information about the implementation.

Software protection is secure if, intuitively, whatever any poly-time adversary can do when having access to an (encrypted) program running on a protected chip, he can also do when having access to a "specification oracle" (such an oracle on any input "magically" gives the output and the running time). Essentially, the protected program must behave like a black box which, on any input, hums for a while and gives an output and such that no information except its I/O behavior and running time can be extracted. Thus, not only the values stored in the general-purpose memory must be hidden (using encryption) but also the *sequence* in which memory locations are accessed during program execution must be hidden. Notice that if this is not the case, the program "loop structure" is revealed to the adversary, even if all the memory locations are securely encrypted. Surely, the information about the "loop structure" does provide some information about the structure of the code. To prevent this, the memory *access pattern* should be *independent* of the program which is being executed.

Nothing in this world comes for free. What is the price one has to pay for protecting the software? The answer is "speed". The protected program will run slower then the unprotected one. What is the minimal slowdown we can achieve without sacrificing the security of the protection? *Software protection overhead* is defined as the number of steps the protected program must make for each step of the source-code program.

The key problem of efficient software protection is to be able to hide the access pattern efficiently. Goldreich shows how to achieve $O((\log m)^c \cdot 2^{\sqrt{2\log m \cdot \log\log m}})$ overhead for hiding the access pattern, where m is the total (RAM) memory size and c is some small constant. He conjectures that a poly-logarithmic (in m) overhead is impossible to achieve.

The main contribution of this paper is to show how to achieve a poly-logarithmic overhead of hiding the access pattern. (Actually, our result is even stronger: we show how to achieve a poly-logarithmic overhead *as a function of the program running time* for hiding the access pattern, instead of poly-logarithmic overhead of *total RAM memory size*, which could be much bigger then the program running time.)

Moreover, in addition to considerable efficiency speedup, the scheme presented in this paper gives a simple and explicit construction of the protection scheme, while the scheme shown in [G], gives a non-trivial recursive solution. It should also be noticed that in [G] one-way permutations are used, while here, we reduce the assumption to the existence of one-way functions only.

References

[G] Goldreich, O. "Towards a Theory of Software Protection and simulation by Oblivious RAMs" *STOC 87*.

[PF] Pippenger , N., and M.J. Fischer, "Relations Among Complexity Measures" *JACM*, Vol 26, No. 2, 1979, pp. 361-381.