Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors

Bart Coppens*, Ingrid Verbauwhede‡, Koen De Bosschere*, and Bjorn De Sutter*†
*Electronics and Information Systems Departement, Ghent University, Belgium
Email: {bart.coppens, kdb, bjorn.desutter}@elis.ugent.be
‡Department of Electrical Engineering, Katholieke Universiteit Leuven, Belgium
Email: Ingrid.Verbauwhede@esat.kuleuven.be
†Electronics and Informatics Department, Vrije Universiteit Brussel, Belgium

***Abstract*—This paper studies and evaluates the extent to which automated compiler techniques can defend against timing-based side-channel attacks on modern x86 processors. We study how modern x86 processors can leak timing information through side-channels that relate to control flow and data flow. To eliminate key-dependent control flow and key-dependent timing behavior related to control flow, we propose the use of if-conversion in a compiler backend, and evaluate a proof-of-concept prototype implementation. Furthermore, we demonstrate two ways in which programs that lack key-dependent control flow and key-dependent cache behavior can still leak timing information on modern x86 implementations such as the Intel Core 2 Duo, and propose defense mechanisms against them.**

I. Introduction

These days many cryptographic systems are implemented on top of programmable instruction set processors. Many components of such processors feature observable data-dependent behavior. When the observable behavior of such components depends on the value of a cryptographic key, attackers can study that behavior to derive information about the key being used.

This information obtained through the observation of an implementation, rather than from the input-output behavior of a cryptosystem, is said to leak from the implementation. The components through which this information leaks are called side channels, and attacks collecting and exploiting this information are called side-channel attacks.

Known side channels are execution time [1], [2], power consumption behavior [3], instruction or data cache behavior [4]–[10], branch predictor behavior [11], pipeline instruction and execution behavior [6], and pipeline speculation behavior [6].

For most of these side channels, countermeasures have been proposed that rely on hardware modifications [6], [7], or on software modifications [1], [2], [8], [9], [12], or on both [5], [10]. An interesting approach was presented by Molnar et al. [13], [14], in which hardware support was combined with the removal of control flow to support the so-called *program counter security model*. Fundamentally, Molnar et al. propose to rely on (special) hardware to guarantee a one-to-one mapping between the flow of control in a program's execution and all observable behavior, and to rely on source-to-source software transformations to remove any control-flow dependency on cryptographic keys. If control flow is made independent of a key, and if the observable behavior only depends on control flow (i.e. on the trace of program counter values, but not on the values being computed during the program execution), no information about the key can be derived through side channels. For example, consider the potential side-channel consisting of the execution time of an application. If control flow is independent of a secret key, and execution time only depends on control flow, then an observed execution time cannot reveal any information about the secret key.

This approach raises some interesting questions. First of all, to what extent can one rely on source-to-source transformations to ensure that no key-dependent control flow occurs in a program? For what types of program constructs can this be guaranteed, for which target architectures can this be guaranteed, and for which compilers can this be guaranteed? Secondly, to what extent do existing architectures, including ones that implement recently proposed countermeasures against side-channel attacks, support the program counter security model? In particular, does the widely used x86 architecture support this, and if so, does it do so without compromising performance too much?

This papers responds to these important questions. The paper's major contributions are:

- We analyze to which extent the most recent Intel implementations of the x86 architecture, which is the most widely used desktop architecture, support the program counter security model, revealing two potential side channels not previously reported.
- We expose flaws in the source-to-source transformation approach proposed by Molnar et al., which make that approach impractical.
- Instead we propose to perform the necessary transformations in a compiler backend. The transformations we propose are similar to those proposed by Molnar et al., but we apply them quite differently.

This work has been supported by the Foundation for Scientific Research - Flanders under project G.0300.07, and by the ECRYPT and HiPEAC European Networks of Excellence.