Compiler Mitigations for Time Attacks on Modern x86 Processors

JEROEN VAN CLEEMPUT, BART COPPENS, and BJORN DE SUTTER, Ghent University

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 data flow. We study the efficiency, effectiveness, portability, predictability and sensitivity of several mitigating code transformations that eliminate or minimize key-dependent execution time variations. Furthermore, we discuss the extent to which compiler backends are a suitable tool to provide automated support for the proposed mitigations.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*Protection mechanisms*; D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*

General Terms: Experimentation, Measurement, Security

Additional Key Words and Phrases: Time-based side channels, x86 architecture, variable latency instructions

ACM Reference Format:
Van Cleemput, J., Coppens, B., and De Sutter, B. 2012. Compiler mitigations for time attacks on modern x86 processors. ACM Trans. Architec. Code Optim. 8, 4, Article 23 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086702 http://doi.acm.org/10.1145/2086696.2086702

1. INTRODUCTION

Many processors execute cryptographic software on a regular basis. When implemented properly, the input-output behavior of that software provides strong guarantees against attacks on privacy, authentication, and other applications of cryptography.

In practice, however, the implemented input-output relation is not the only observable property. Depending on the physical or network access to devices, attackers can observe properties such as electromagnetic radiation, power consumption, resource consumption and execution times. These properties are called side channels when they feature a correlation with protected data such as secret keys. Side-channel attacks exploit this correlation to attack cryptographic software.

1.1. Side-Channel Attacks and Defenses

Depending on the granularity with which side-channel information can be observed by an attacker, different types of attacks can be mounted. Trace-driven attacks have been described in which side channels leak enough information to reconstruct the whole execution trace of the attacked algorithm [Kocher et al. 1999; Lauradoux 2005; Aciiçmez and Ç. Koç 2006; Aciiçmez et al. 2007a, 2007b; Aciiçmez 2007]. Access-driven attacks have also been discussed in which enough side channel information is available

This work was supported by the Agency for Innovation by Science and Technology (IWT), by the Fund for Scientific Research - Flanders, and by Ghent University.
Authors' address: J. Van Cleemput, B. Coppens, and B. De Sutter, Ghent University, Department of Electronics and Information Systems, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium; email: {Jeroen.VanCleemput, Bart.Coppens, Bjorn.DeSutter}@elis.UGent.be.

DOI 10.1145/2086696.2086702 http://doi.acm.org/10.1145/2086696.2086702

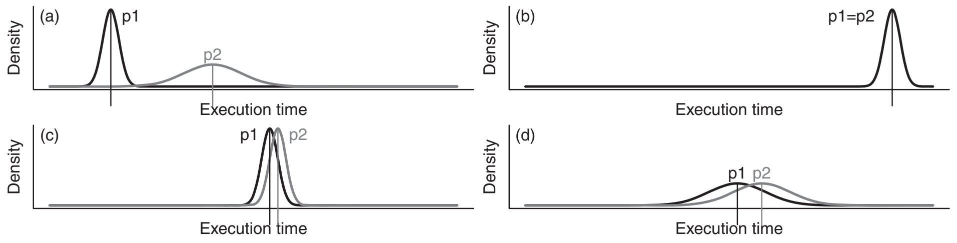


Fig. 1. Conditional execution time distributions of a cryptographic algorithm given two possible secret keys.

to reconstruct the access patterns to architectural components such as caches, branch predictions tables, and execution units [Neve and Seifert 2007; Aciiçmez et al. 2010; Gullasch et al. 2010; Brumley and Hakala 2009; Osvik et al. 2006; Ristenpart et al. 2009; Wang and Lee 2006; Uhsadel et al. 2008]. Time-based attacks have been proposed that are based solely on the observed execution time [Bernstein 2005; Bonneau and Mironov 2006; Brumley and Boneh 2005; Brumley and Tuveri 2011; Kocher 1996; Aciiçmez et al. 2007; Dhem et al. 1998]. It has been shown that for many attacks, the dependence between secret key properties and side-channel information need not be known to the attacker beforehand. Instead, it suffices to discover the existence of a dependence during the attack. One well-known method to measure this dependence is based on so-called mutual information [Gierlichs et al. 2008].

Many mitigation strategies and countermeasures against side-channel attacks have been described in literature. They range from the hardware level [Wang and Lee 2006; 2007; Gueron 2008; Bernstein 2005], over software [Molnar et al. 2005; Bernstein 2005; Hedin and Sands 2005; Brickell et al. 2006; Coppens et al. 2009; Köpf and Dürmuth 2009], to the algorithmic level of abstraction [Kocher 1996; Joye and Yen 2003; Guajardo and Mennink 2010], and combinations thereof [Bayrak et al. 2011].

This paper explores the trade-off between security, performance, and portability of compiler-based defenses against time-based side-channel attacks on x86 processors. We consider attacks based on direct measurements of the execution time of a cryptographic process, as well as indirect attacks that measure the influence of a cryptographic process on the execution time of another process controlled by the attacker.

1.2. Leaked Information and Security Guarantees

Some of the aforementioned attacks derive bits of the secret key directly from the reconstructed traces and patterns or from the observed timing [Gullasch et al. 2010; Aciiçmez et al. 2007a], while others collect statistical information with which the search space of a brute-force attack can be pruned [Köpf and Basin 2007]. In the latter case, the pruning will be more effective when more useful information is leaked. Regarding this leaked information, most countermeasures aim at limiting the amount of information leaked [Bayrak et al. 2011; Köpf and Dürmuth 2009]. Other countermeasures try to add noise to the information [Kocher 1996]. To illustrate the difference between limiting information leakage and adding noise, imagine a decryption algorithm with two possible keys. The two conditional statistical distributions of the algorithm's execution time (one for each key) are depicted in Figure 1(a). Very few measurements of the execution time suffice to extract the key reliably. When the algorithm is rewritten such that both conditional distributions become identical, as depicted in Figure 1(b), no information leaks at all. When the distributions are not identical but overlapping, as in Figures 1(c) and 1(d), less information is available to an attacker. The amount of useful information depends on the whole conditional distributions, not only on their expected values. So depending on the attack context, more or less difference in expected execution times can be tolerated. When little noise is present because an attacker has full control over all tasks running on a local machine, an acceptable level of security can only be reached by making the execution times obtained with different keys nearly identical, as in Figure 1(c). But when a lot of noise is present due to variable network delay over the Internet and due to the unknown load of a cloud processor, a larger difference in expected execution times as depicted in Figure 1(d) can still provide the same level of security. The effect of such noise has been quantified in literature [Ristenpart et al. 2009; Crosby et al. 2009]. As can be expected, the signal-to-noise ratio has a major influence on the amount of useful information an attacker can extract, and on the effort he has to invest to mount an attack. Smaller signal-to-noise ratios require the attacker to perform more measurements, up to the point where it can become infeasible to mount an attack, e.g., because encryption keys are refreshed frequently.

This paper studies compiler transformations for sensitive code fragments that reduce the signal-to-noise ratio by minimizing (potentially even eliminating) the dependence of execution time behavior on secret key values. We study the overhead of a number of code transformations in terms of average performance losses, as well as their resulting security in terms of indistinguishability between different keys' execution times. We also study the portability of those properties over different processor architecture versions and their sensitivity to features of the transformed code fragments. Furthermore, we pinpoint issues with the predictability and testability of the delivered security. Together, these aspects enable us to assess the feasibility of mitigation techniques in static compilers and in dynamic compilers. The latter are found in virtual machines (VMs) that use JIT compilers and in dynamic binary translation engines. Static and dynamic compilers differ with regard to side-channel mitigation because on the one hand static compilers are allowed to spend more compilation time for providing security guarantees, while dynamic compilers on the other hand do not need to worry about the portability of the security guarantees.

1.3. Contributions of This Paper

Key-dependent control flow transfers are the most obvious cause of correlation between secret keys and execution time. This correlation can be avoided by eliminating conditional branches and by fixing loop bounds. Our previous work discussed how a compiler backend can exploit the conditional move instruction on the x86 to eliminate branches by means of if-conversion [Coppens et al. 2009]. The elimination of conditional control flow transfers also eliminates, in a portable manner, the influence of secret keys on execution time through the microarchitectural side channels of branch prediction, instruction caching, and branch target buffers [Aciiçmez et al. 2007a, 2007b; Aciiçmez 2007]. Others have proposed satisfying solutions for closing the microarchitectural side channel of data caches [Wang and Lee 2006, 2007].

This paper therefore focuses on operations with variable latency irrespective of cache and branching behavior. The issues caused by such instructions relate to data flow and to features of modern processor pipelines. The presence of these causes was discussed in our previous work [Coppens et al. 2009], but no solutions were studied. This paper presents and evaluates such solutions. Our main contributions are the following.

—The study of several mitigation techniques against timing variations caused by data flow behavior on modern x86 processor pipelines.
—A demonstration of the fact that compilers can provide strong protection only at a high performance overhead, and without forward compatibility.
—A demonstration of the fact that weaker protection, without portable security guarantees, can be provided at lower levels of overhead.

1.4. Structure of the Paper

The remainder of this paper is structured as follows. Section 2 discusses causes of timing variations related to data flow and modern x86 pipelines. Section 3 discusses some potential mitigations, which are evaluated in Section 4. Section 5 draws conclusions.

2. EXECUTION TIME ON MODERN PROCESSORS

This section discusses the variations on execution time on modern x86 processors caused by data flow properties. As for execution time variations relating to control flow and caches, we will neglect those causes in the remainder of this paper unless explicitly stated otherwise, and we refer the reader to the existing literature as mentioned in the introduction. This section also discusses the relevant differences with regard to data flow between different recent x86 processors.

For this paper, we performed experiments on a dual core Intel Core 2 Duo E8400 that lacks hyperthreading (HT), on a dual CPU 2x4 core Intel(R) Xeon E5620 with HT, and on a single core Intel Atom N280 with HT. On any of these processors, the execution time of a program depends heavily on the data dependencies between successive instructions in a program. As the (true) data dependencies on the critical path in a data dependency graph (DDG) of the executed code put a fundamental limit on the Instruction-Level Parallelism (ILP) that can be exploited by a processor, having fewer dependencies implies that more Instructions get executed Per Cycle (IPC).

By and large, the data dependencies in a program are fixed statically, as they are determined by the registers occurring in the instruction encodings. The one exception to the observations is formed by memory operations. Whether a store and a consecutive load depend on each other depends on the addresses used in the operations. The influence of this dependence on execution time is discussed in detail in Section 2.3.

Furthermore, the length of the critical path in a DDG is determined by the individual execution latencies of the operations on it. Most non-memory operations have fixed latencies. There are some exceptions, however, for arithmetic instructions that are so complex that they are implemented by means of microcode instruction sequences. Some typical such sequences allow an early exit (a.k.a. early termination), of which the influence on execution time is discussed in Section 2.2.

First we briefly discuss how conditions occurring in conditional move instructions do not influence execution time, as we will rely on this property in the mitigating transformations presented later in the paper.

2.1. Conditional Moves

Consider the three following x86 loop bodies.

```
body 1: mov ecx, edi      body 2: mov eax, edi      body 3: test edx,edx
        add eax, ebx              add eax, ebx              cmoveq eax, edi
                                                            add eax, ebx
```

In all loop bodies, the last instruction adds the value in `ebx` to that in `eax`. When the first loop body is executed in a loop, `eax` serves as an accumulator to which the value in `ebx` is added repeatedly. So all additions in subsequent iterations depend on each other. In the second loop, each iteration starts with a fresh value being copied into `eax`. The register renaming pipeline stages in out-of-order processors detect this, and the second loop gets executed up to 40% faster as a result.

In the third loop body, `test` sets condition flags depending on the loop-invariant value in `edx`. If that value is zero, the `cmoveq` instruction is executed, copying the value of `edi` into `eax`. In that case, the third body performs the same computation as the second body. If the value of `edx` is not zero, the third body performs the same computation as the first body. Despite these two different behaviors, we observed identical execution times when executing this third loop body in an unrolled loop with `edx` either fixed to zero or to some nonzero value.

This experiment demonstrates that the values of the guard conditions of conditional moves do not influence execution timing. As discussed extensively in our previous work [Coppens et al. 2009], this property fundamentally results from the fact that register renaming is implemented in the in-order stages of pipeline frontends. As we do not expect that aspect of pipeline design to change in the near future, we can safely rely on this fixed execution time of conditional moves. We relied on it to apply if-conversion in our existing work [Coppens et al. 2009], and will rely on it again in Section 3.2.

2.2. Variable Latency Arithmetic Instructions

Variable-latency arithmetic instructions such as multiplication are known to be a side channel [Groszschaedl et al. 2009]. For that reason, some architectures offer the option to disable the variable latency through control registers [ARM Limited 2004].

Studying several x86 documents [Coke et al. 2008; Fog 2011], we found one class of commonly used arithmetic instructions with variable instruction latency: the signed and unsigned, 32-bit and 64-bit variations of the integer division. These instructions are also used to compute remainders of divisions, and hence play a role when modulo arithmetic is needed, as in many cryptographic algorithms.[1]

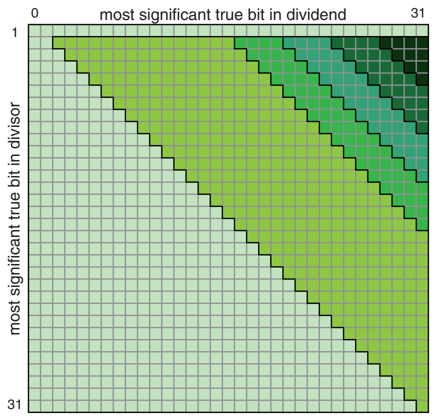
Intel documents state that the execution time of an integer division instruction on its recent out-of-order processors depends on arguments being zero and on the number of quotient bits that need to be generated. This number equals the distance in bit positions between the most significant bits of the divisor and the dividend. Public documentation also mentions the minimum and maximum latency of the division instructions on different generations of Intel core [Fog 2011; Granlund 2011], but further details on the relation between operands and latency are missing.

In order to understand that relation and the potential consequences for side-channel attacks, we measured the execution time of a program consisting of division instructions executed in a loop on loop-invariant structured and randomly selected arguments covering the whole 32-bit unsigned integer range. We measured execution times with operands which are exact powers of two and compared this to the execution times using random numbers of the same magnitude. We found the latencies indistinguishable.

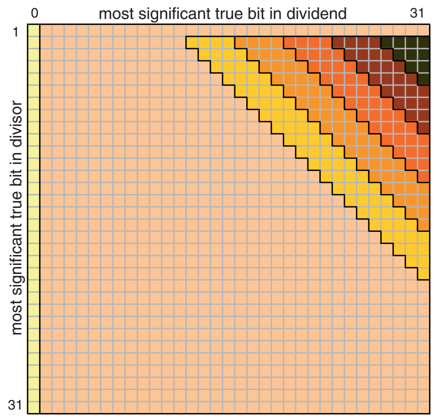
As we can only measure the execution time and cycle count of that whole program, we cannot give exact latencies for individual divisions. We can, however, distinguish different latencies by comparing the total execution times. Figure 2 shows the results of this experiment for the Core 2 and Xeon for 32-bit unsigned divisions. Each shade corresponds to a latency, with darker shades corresponding to higher latencies.

Several observations can be made about these results. First of all, the Intel documentation is correct. Secondly, there are a limited number of distinct latencies: six for the Core 2 processor, and seven for the Xeon Nehalem core. Third, due to the regularity of the results and the existence of the `bsr` bit-scan-reverse instructions that computes the location of the most-significant bit set, it is rather easy for a given processor to write code that computes the latency class given the divisor and dividend values. Fourth, the exact latencies and their patterns differ from one architecture generation to the other.

[1]While no side-channel attacks exploiting variable-latency divisions are currently known, it is better to be cautious. Moreover, variable-latency divisions are conceptually not different from variable-latency multiplications. The conclusions drawn here can hence also be used on other architectures that feature variable-latency multiplications [ARM Limited 2004], and in the event Intel would decide to implement variable-latency multiplications in the future.



(a) Core 2 latency classes



(b) Xeon latency classes

Fig. 2. Table showing the different latency classes of the 32-bit unsigned integer division instruction depending on the most significant true bits in dividend and divisor.

The results for other division instructions are similar, the main difference being that for 64-bit divisions, about twice as many different latency classes are observed. So the four above observations still hold. In Section 3, we will consider some mitigation strategies based on these observations.

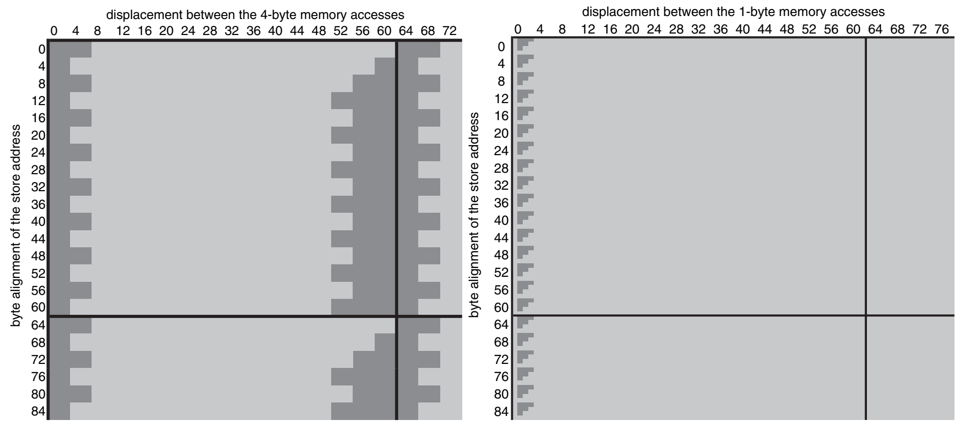
2.3. Interaction between Memory Operations

Consider the following program fragment.

```
loop body:   mov dword    [ebx], 2    // store
             add          eax, [ecx]  // load
```

When this fragment is executed in a loop, it will repeatedly store the value 2 at the memory location to which register `ebx` points, and it will repeatedly add to `eax` the value at the memory location to which `ecx` points. Since only two memory locations are touched in this loop, the cache will not influence the timing significantly when the loop has enough iterations. A number of other micro-architectural features do influence the execution time of such a loop, however. Some of these features only relate to the static properties such as the instruction mix and instruction ordering in the assembly code. Their influence on the execution time of a program will hence be constant over all possible program executions. Consequently we can safely ignore these with regard to the time-based side channels.

Other features do depend on the actual memory locations accessed and will cause a different program execution time for different program inputs, causing potential information leakage. The most important such features present on some but not all Intel processors are optimistic and pessimistic load bypassing, store forwarding, 4K aliasing, memory disambiguation (i.e., conflict speculation), alignment, partial aliasing, and bank conflicts. While code optimization manuals document these features [Intel Corporation 2011; Doweck 2006; Intel Corporation 2010; Shen and Lipasti 2005], no specification of their exact implementation, interaction and timing behavior is available. Moreover, their implementation and their behavior differ significantly from one architecture generation to the other. In addition, the influence on execution time depends to a large extent on the arithmetic code that surrounds the memory accesses. Assuming no cache misses or branch mispredictions, the execution progress is mainly limited



(a) Core 2 Duo 4-byte access latency classes

(b) Atom byte-access latency classes

Fig. 3. Execution time differences on a Core 2 Duo and Atom processor of a microbenchmark loop with aligned 32-bit and 8-bit resp. load and store instructions executed for varying displacements between the accessed locations and for varying alignments of the addresses.

by saturating buffers in the instruction pipeline that cause pipeline bubbles or stalls. Which buffers cause stalls first, and hence which specific slow down is experienced, depends on the instruction mix as well as on buffer sizes.

In summary, it is very difficult if not impossible to predict the execution speed of memory access streams. In fact, it is even difficult if not impossible to pinpoint the exact reasons for slowdowns in observed executions.

For example, we measured the execution time of the above loop body storing aligned 4-byte values on an Intel Core 2 Duo for different loop-invariant base addresses in `ebx` and `ecx`. Two different execution times were observed, depending on the offset between the load and store addresses, and on their alignment. Figure 3(a) depicts the precise relation we observed. Even for this simple loop, we cannot explain this relation completely. Most of the light area (i.e., faster execution) and the first, dark column can be explained by means of load bypassing. When the load and store addresses differ, load bypassing speeds up the program. When they are the same, there is a dependency that slows down the program. The fact that the same behavior is observed for all offsets modulo 64, i.e., the fact that the behavior is periodic with period 64, indicates that the load bypassing is pessimistic and that only bits 3 to 5 of the addresses are used to determine if load bypassing is allowed. However, for the slowdowns observed in the columns 4, 52, 56, and 60, we have no satisfactory explanation. We suspect that they are caused by pipeline features that optimize the handling of (unaligned and partially overlapping) 64-bit and 128-bit (SSE) memory accesses, but cannot confirm this. We studied several Intel manuals, collected many program traces including all possible performance counters (including ones that count store buffer saturation, the stall cycles resulting from it, the number of overlapping loads and stores, the numbers of loads and stores accessing multiple cache lines, etc.) and contacted Intel engineers, but none of these information sources provided satisfying explanations.

At this point, it is important to repeat that it is not necessary to understand the causal relation between secret data and observable side channel behavior to mount a successful side channel attack. Techniques have been developed that detect any existing correlation automatically and that exploit them to retrieve additional information about the secret data [Gierlichs et al. 2008; Batina et al. 2011].

Data dependencies through memory do not only occur on complex out-of-order architectures. Even on the less complex Atom architecture we observe timing variations due to data dependencies through memory. Consider the following program fragment.

```
loop body:   mov byte [ebx], 2    // store
             add al, [ecx]        // load
```

Instead of 4-byte words, this fragment accesses individual bytes. As this is an in-order architecture, load bypassing cannot be the cause of difference in execution time depending on addresses. However, on this processor we also observe different timings, as visualized in Figure 3(b). In this case, the difference in timing seems to be caused by different ways of overlapping of forwarded data. Intel documentation only states that different latencies can occur when accessing multiple bytes within a 4-byte word. Clearly, the differences for this code and processor occur along very different patterns.

We conclude this section with pointing out that each of the two visualized patterns only occur on one processor architecture. The first pattern of Figure 3 as observed on the Core 2 Duo processor is completely absent when running the same 4-byte memory access microbenchmark on the Xeon Nehalem (which has larger buffers and a different memory hierarchy architecture) or on the Atom processor (which is in-order). Vice versa, the second pattern of Figure 3 as observed on an Atom processor is completely absent on the two out-of-order cores.

3. MITIGATION STRATEGIES

This section discusses several potential mitigation strategies for variable-latency arithmetic instructions and for interacting memory operations.

The most obvious mitigation strategy is to avoid the variable-latency instructions altogether. For the division instruction, e.g., it is easy to write a library function that performs the division in constant time, without division instructions. Replacing each division by a call to such a function solves the problem, albeit at a significant overhead as we demonstrated in our previous work [Coppens et al. 2009]. Moreover, for other instructions such as memory accesses, it is simply impossible to avoid them.

In this paper, we study alternative solutions that do include the execution of variable latency instructions, but in such a way that they do not influence the total execution time. Two different code rewriting strategies are studied to achieve this. The first strategy is to rewrite the code such that the total execution time no longer depends on the variable latency of individual instructions. The second strategy is to rewrite the code such that all instructions that have variable latency in general, now get executed in a context that forces a constant latency on them.

3.1. Strategy One: Variable-Latency Compensation Code

With this strategy, we try to add *compensation code* to program fragments such that the execution of the original variable-latency code and the compensation code combined always results in the same total execution time.

When we neglect all instruction caches, instruction branch target and translation look-aside buffers, as we can after conditional branches have been eliminated, the total execution time of a sensitive code fragment on a processor is determined by (1) the state of the processor upon entry of the code fragment, (2) the data consumed by the fragment, and (3) the DDG of the code fragment itself.

Forcing the entry state to some predetermined state upon entry of a sensitive variable-latency code fragment is impossible without huge performance overhead. As a result, it is impossible to make the execution time of any sensitive code fragment truly constant with low overhead. However, we don't need that time to be constant. We only need it to be independent of the secret data. That implies that variations in

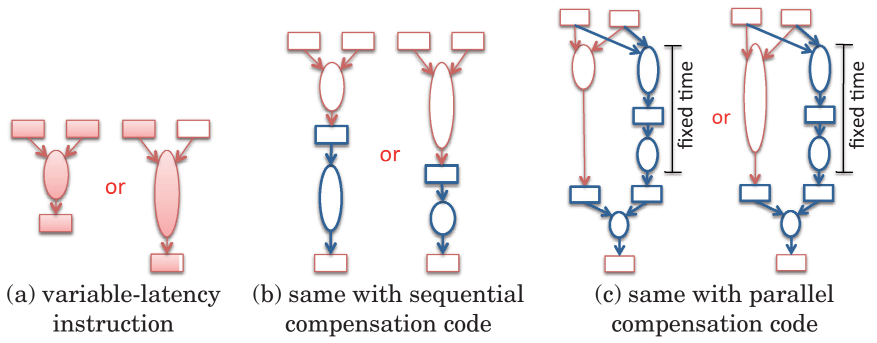

or
or
fixed time
or
fixed time
(a) variable-latency instruction
(b) same with sequential compensation code
(c) same with parallel compensation code

Fig. 4. A simple DDG in which visual height models instruction latency and execution time.

the entry state that do not depend on secret data can be tolerated. As such, we can reason along the lines of mathematical induction: when the entry state of a fragment is independent of secret data and when the fragment's own execution time and processor state transition is independent of secret data (because it does not consume secret data or because we apply a mitigation technique), the exit state is also independent. As such, the entry state of the next fragment is independent of secret data. Clearly, it is possible to make the entry state of the program independent of secret data. So we don't need to consider the entry state to a sensitive program fragment as long as we can take care of the fragment itself.

Changing the data consumed by a sensitive code fragment is generally not feasible for a compiler when that data concerns the secret key and input data or some derivatives thereof. Only the algorithm designer or programmer can do that. So the only remaining option is to transform the DDG of the code fragment such that irrespective of the data operated on, the fragment's execution time is independent of the secret data on which it operates. We study two potential DDG transformations to achieve this.

First, we can try to make the total latency of the path in the DDG containing the variable-latency instruction constant by adding *sequential compensation code* such that the sum of the latencies on the resulting path always equals a constant. Consider the original variable-latency instruction as depicted in Figure 4(a). The two alternative executions of the DDG in the figure model the fact that whenever the instruction is executed, it will execute with one of two latencies. In Figure 4(b) sequential compensation code has been added. This compensation code is visualized in the figure as one dark blue operation node for the sake of clarity. In practice, however, the compensation code cannot simply be one instruction, as it needs to compute what should be compensated and before it can actually perform the necessary compensation.

Secondly, we can try to ensure that the variable-latency instruction is not on the critical path of the DDG by inserting parallel compensation code that needs more cycles to execute than the maximum of the variable latencies. This concept is visualized in Figure 4(c). For a combination of reasons, this parallel compensation code will typically also have to consist of more than one instruction, as depicted in the figure. In order to hide the variable latency completely, the parallel compensation code obviously needs to be executed in parallel with the variable-latency code. So it must consist of instructions that do not execute on the same pipeline components. Hence the parallel compensation code cannot contain the variable-latency instruction itself, or variations thereof. Furthermore, the very reason for instructions being implemented with variable latency is a high maximum latency, up to 116 cycles for the integer division on some Intel x86 processors. This implies that the parallel compensation code has to be built from multiple, shorter, fixed-latency instructions.

Considering the pipelines of modern x86 instructions, several instructions are available for the parallel compensation of variable-latency divisions, including sequences of multiplication. Section 4 reports on the results obtained with such sequences.

We should note that for indirect attacks, the parallel compensation method cannot provide any solution. Such attacks do not measure the execution time of the process under attack but instead measure that process' effect on another process caused by resource contention. This effect can even be observable through many software layers. For example, we verified that on an x86 processor with simultaneous multithreading (SMT), one Java program executing divisions in one Java VM running on top of an operating system virtualized with Xen and pinned to a specific SMT core, can approximately measure the occupation of the divider by another Java program running in another Java VM, in another Xen VM, but pinned onto the same SMT core.

With parallel compensation code, the occupation of the execution units used by variable-latency operations remains variable, so parallel compensation code cannot close this indirect time side-channel.

3.2. Strategy 2: Forcing Invariable Latencies

An alternative strategy is to manipulate the operands of variable-latency instructions or the way in which memory accesses interact to force a constant latency on them.

In the case of division, for example, one could imagine shifting the operands before the division, and then shifting back after the division to force maximum latency. Or by trying to replace a division A/B by $2(A+2B)/2B-2$ which would never cause early exit. However, given that most division instructions in cryptographic software are executed to compute remainders, we have not found any such sequence which does not suffer from either rounding errors or from overflow in parts of the operand range. So this is not a generic solution. In some cases, however, this type of mitigation is feasible, such as when the divisor is public knowledge. This is often the case in public key cryptography where the modulus of the encryption and decryption is part of the the public key, and where it is most of the time a fixed value throughout the computation [Menezes et al. 2001]. The execution time is then only dependent on the most significant bit in the dividend. To eliminate execution time variation, it suffices to shift the dividend to the left such that the most significant bit is always set to true. The divisor does not need to be shifted, and hence overflow and rounding errors are not a problem.

An alternative, generic solution consists of building a DDG in which multiple copies of the variable-latency instruction are executed, with exactly one copy per possible latency, and by forcing the operands of those copies to be such that all of them have the desired constant latency. In addition, forcing the operands to appropriate values has to happen in such a way that at least one copy executes the proper, original division. Then after all divisions have been computed, code needs to select the correct result among all computed results by means of conditional moves.

This solution is visualized in Figure 5. Whereas Figure 4 depicts alternative executions of a DDG, this figure depicts only one DDG. The inserted nodes at the top model the code that computes suitable operands from the original operands such that all copies of the original instruction will have a different, but constant execution time. The node inserted at the end models the selection of the correct result among all computed ones. Clearly, this solution will also involve some overhead. Using the already mentioned `bsr` instruction, the overhead stays limited. We evaluate it in Section 4.

This solution also provides excellent protection against indirect attacks: as the resource use is now constant, its influence on other processes becomes constant as well.

For memory operations that have variable latency because their interaction depends on concrete addresses on which they operate, the above solutions are not applicable. We can, however, force them to have an invariable execution time by excluding the

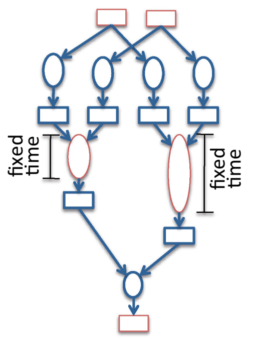


Fig. 5. An extended DDG that includes all latency variations of a variable-latency instruction.

unwanted interaction altogether. For example, in the case of interacting stores and loads, all of the pipeline optimizations discussed in Section 2.3 only matter when the store and load are in flight in the processor pipeline simultaneously. So whenever such a pair of instructions occurs in a program fragment of which the compiler cannot determine that the addresses operated upon will lead to fixed timing or that they are independent of any secret data, it suffices to pull the instructions apart.

This can be done by simply inserting no-ops. While it may come as a surprise that simple no-op insertion can work, processor designers do not expect compilers or programmers to insert no-ops in sequential code. While no-ops are inserted to optimize code alignment, the inserted no-ops are then merely padding that almost never gets executed. So the processor designers do not implement any pipeline optimization to get rid of no-ops in instruction streams [Intel Corporation 2011]. Consequently, the inserted no-ops result in pipeline bubbles, which can effectively force loads and stores to be executed separately, without any interaction and hence without variable execution times as a result. This strategy is also evaluated in Section 4.

With respect to indirect time attacks, this method of inserting no-ops provides as many guarantees as for direct time attacks. When the interaction between loads and stores is avoided (and still neglecting caches for which other solutions exist), their occupation of buffers and other resources in the pipeline also becomes data-independent. So resource contention with other processor cannot leak any information.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the proposed mitigation techniques. This evaluation includes the mitigation success, the performance overhead, and the feasibility of implementing the proposed mitigations in a compiler backend. All results for variable-latency arithmetic are based on compiled and handwritten assembly implementations of modular exponentiation, of which a basic C implementation looks as follows.

```
result = 1;
do {
  result = (result*result) % n;
  if ((exponent>>i) & 1)
    result = (result*a) % n;
  i--;
} while (i >= 0);
```

To eliminate the conditional control flow, we relied on the support implemented in LLVM and discussed in our previous work [Coppens et al. 2009]. The generated code forms the basis for the experiments described here. In this code, the division instruction is used to perform the modulo n computation.

To evaluate the effectiveness of the proposed mitigation techniques, we ran the modular exponentiation code on inputs consisting of (1) randomly varying modulo values, (2) randomly varying base values, and (3) four different types of exponents.

In the *Zero* input set, the exponent in binary format consists of all zeroes except for the two most-significant bits set that are set to one. This ensures that the variable `result` does not remain constant throughout the whole loop. Having all other bits set to zero ensures that the conditional code in the original loop will only be executed twice per loop. This pattern results in very accurate branch prediction. In the *One* input set, all bits in the exponent are set to one. This ensures that the conditional code in the loop is executed in every iteration in the original, unprotected code. So in total, the conditional code is then executed 32/64/256 times per loop for 32/64/256-bit numbers. This pattern also results in very accurate branch prediction. So when this input is fed to a benchmark, much more code is executed than with all-zero input, but the branch predictor performs similarly. In the *Regular* input set, half of the bits are set to one in a regular pattern. This implies that the conditional code is executed in half of the iterations in the original unprotected code, and that the pattern is predicted very well by the branch predictor. In the *Random* input set, half of the bits are set to one as well, but now the pattern of zeroes and ones is generated by a pseudo-random number generator. Consequently, this input will result in the same amount of code executed as for the regular input set, but branch prediction will be much less accurate, resulting in more branch misses and higher execution times in the original, unprotected code.

Please note that the number of times each loop was invoked per experiment differs from one experiment to the other. For each benchmark, the number of invocations was chosen to be a good balance between fast experiments and accurate measurements.

4.1. Strategy One: Variable-Latency Compensation Code

To test the mitigation strength of compensation code as discussed in Section 3.1, we developed an LLVM [Lattner and Adve 2003] plugin to insert parallel compensation code. This plugin operates on the LLVM high intermediate representation. The generated compensation code for this example takes the dividend as input and invariantly computes the value 1 using a number of shifts and mostly multiplications, as illustrated in the equivalent C code in Figure 7. That resulting 1 is then multiplied with the remainder result of the division instruction. That final multiplication serves as the bottom node of Figure 4(b). It take the division instruction off the critical path.

Several versions of this loop were generated, with increasing numbers of multiplications to discover the minimal required number to omit the variable-latency division from the critical path. Figure 6(a) demonstrates that the average execution times for the four inputs converge after 6 multiplications. So surely this mitigation technique helps in reducing the amount of useful information leaked via the time side channel. The performance overhead is also limited in this case. For 6 multiplications, the overhead is about 7%.

To test whether all useful information is eliminated, more rigorous statistical testing is needed. To that extent we performed t-tests on the sets of 100 timings we obtained for each of the four inputs. Figure 6(b) shows the p-values of those tests obtained from comparing the One input set to the Zero input set and from comparing the Regular input set to the Random input set. This figure shows that only the versions with 15 and 39 multiplications survive the t-tests. So according to these tests, only those versions are likely free of leakage.

To understand why precisely these two versions provide more security, we studied the execution of the mitigated code versions using a wide range of performance counters. We discovered that even when the compensation code is executed on different functional units to enable its execution in parallel with the division instruction, there

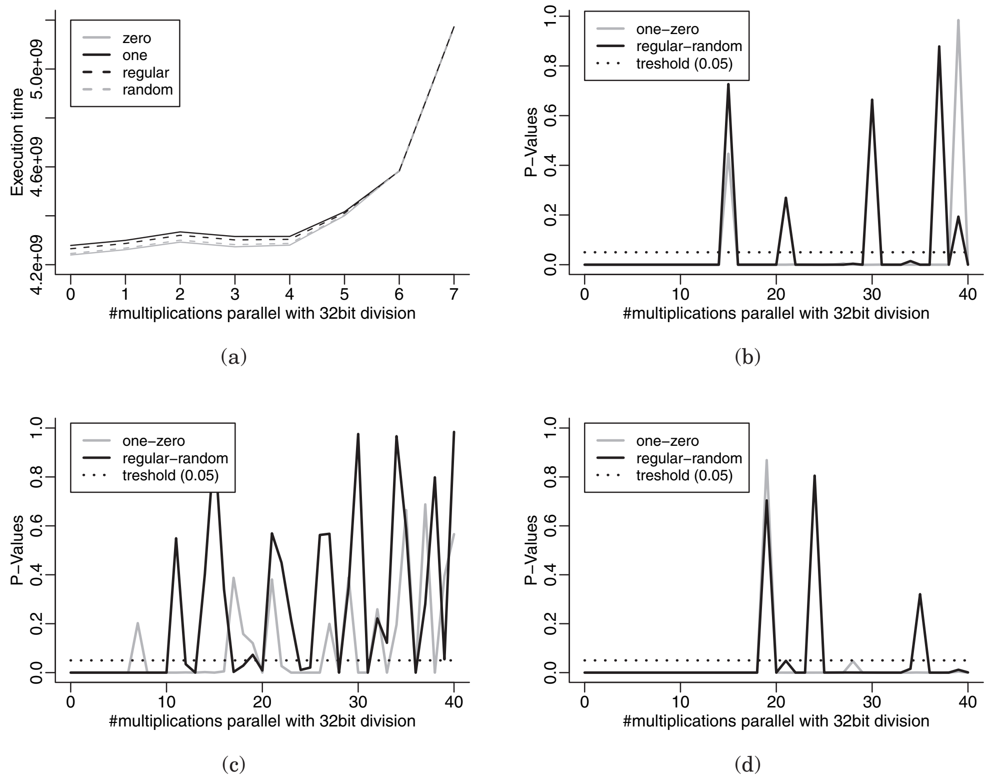


Fig. 6. (a) Average execution times (number of cycles on a Core 2 Duo) of a 100 executions on a Core 2 Duo processor of a loop performing modular exponentiations, for four different inputs. Processor time stamp counters are used to measure the execution times. (b) p-values of t-tests to test distinguishability between different inputs on the Core 2 Duo. (c) p-values for same program on a Xeon processor. (d) p-values for a minimally changed program on the same Core 2 Duo.

```
tmp1 = (dividend >> 31) | 1; // tmp 1 is always 1 or -1
quotient = dividend / divisor;// original division intended to be executed in parallel with multiplications
tmp2 = tmp1 * tmp1;           // tmp2 and beyond are always 1
...
tmp = tmpn * tmpn;            // n multiplications in total
quotient = quotient * tmp;    // final multiplication to take division off critical path
```

Fig. 7. C code equivalent of variable-latency division in parallel with fixed-latency multiplications.

are various other pipeline components such as buffers and ports for which there is contention between the compensation code and the division. Through this contention, the variable latency of the division instruction can still influences the execution of the compensation code, thus influencing the total execution time. For this precise combination of processor architecture and code fragment, 15 and 39 proved to be the right numbers of multiplications to eliminate the time dependence completely.

We have no knowledge of publicly available hardware models that would allow a compiler to predict this. Furthermore, when we make small changes to the original code before inserting the parallel compensation code or when we run the code on other out-of-order processors with variable-latency division instructions, similar results are obtained, that peaks in the t-test results occur for different numbers of multiplications. Figure 6(c) shows the t-test result for the same code fragment on our Xeon processor, while Figure 6(d) shows the t-test result for the same Core 2 Duo processor and for a software version in which we removed 4 padding bytes before the function containing the sensitive code fragment.

Similar experiments in which we used long-latency load operations (through forced cache misses) as parallel compensation code, gave results along similar lines.

From these experiments, we draw the following conclusions.

—In at least the presented case, parallel compensation code is able to limit the amount of useful information leakage significantly.
—The performance overhead is relatively low when no strict security guarantees are needed, as in the case where close but non-identical averages suffice.
—In at least the presented case, parallel compensation code is able to eliminate all leakage.
—However, this stricter security guarantee can only be achieved at a much higher overhead.
—A specific instance of the mitigation, such as a specific number of multiplications, provides no portable security guarantees for different processor versions.
—It is unpredictable which specific instance of the mitigation is most effective.
—The effectiveness of a precise instance of the mitigation is highly sensitive to the precise form and even location of the code fragment to be protected.

With respect to the sequential compensation code strategy, we simply did not succeed in write concrete, effective mitigation code. And even if sequential compensation code can be crafted that satisfies the needed security guarantees for a specific software-hardware combination, all of the mentioned predictability, portability and sensitivity issues will still apply.

4.2. Strategy 2: Forcing Invariable Latencies

Next, we manually rewrote the 32-bit version of the modular exponentiation code to implement the strategy depicted in Figure 5. The resulting average execution times and p-values are presented in Figure 9. Results are again presented for four inputs and two t-tests. Four software versions have been measured in this experiment:

(1) *original*: the original version as compiled with LLVM without any mitigation;
(2) *if-conversion*: the version generated by a modified LLVM that eliminates the conditional branches [Coppens et al. 2009];
(3) *if-conversion + nodiv function*: an if-converted version in which LLVM replaced the division instruction by a call to a fixed-time library function that emulates division without executing a single division instruction.
(4) *if-conversion + 6 div function*: an if-converted version in which LLVM replaced the 32-bit division instruction by a call to a manually written assembly function in which 6 divisions with forced invariable latencies are executed, corresponding to the 6 latency classes of the 32-bit division instruction on the Core 2 architecture. Figure 8 shows the main body of this function in equivalent C code.

These results indicate that the proposed solution is capable of completely closing the time side-channel leak due to the variable latency division instruction. Equally important, additional experiments demonstrated that this solution does not depend on the exact form of the code fragment to be protected. Whatever the surrounding code of the division looks like, the proposed solution works.

The overhead of this solution is more than a factor 4, however, which is considerably higher than the overhead resulting from parallel compensation code as measured in the previous section. Still, compared to using a library function, this novel mitigation technique is about 3.5 times more efficient.

```
class = divisor < 2 ? 1 : divisor < 0x20 ? 2 : divisor < 0x200 ? 3 : divisor < 0x2000 ? 4 : ...;
leading_zeroes = 31 - bsrl(dividend);   // using fixed-latency bit-scan-reverse instruction
dividend <<= leading_zeroes;
result1 = shifted_dividend / ( class == 1 ? divisor : 0x2);       // fixed latency!
result2 = shifted_dividend / ( class == 2 ? divisor : 0x20);      // fixed latency!
result3 = shifted_dividend / ( class == 3 ? divisor : 0x200);     // fixed latency!
result4 = shifted_dividend / ( class == 4 ? divisor : 0x2000);    // fixed latency!
result5 = shifted_dividend / ( class == 5 ? divisor : 0x20000);   // fixed latency!
result6 = shifted_dividend / ( class == 6 ? divisor : 0x200000);  // fixed latency!
quotient = class == 1 ? result1 : class == 2 ? result2 ; class == 3 ? result 4 ? ...;
quotient >>= leading_zeroes;
remainder = dividend - (quotient * divisor);
```

Fig. 8. C code equivalent of unsigned division computation using only invariable-latency divisions. All selection statements a?b:c are implemented with fixed-latency conditional moves. The (re)computation of the remainder is necessary because the remainders computing using shifted dividends are not correct. Similar code can be used for signed division, but then the sign needs to be corrected afterwards.

original				*if-converted*				*if-converted + nodiv function*				*if-converted + 6 div function*			
all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
0,611	1,127	0,845	0,978	1,242	1,255	1,251	1,245	20,800	20,800	20,800	20,800	5,774	5,774	5,774	5,774

(a) average execution times (in seconds)

original				*if-converted*				*if-converted + nodiv function*				*if-converted + 6 div function*			
all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
0,007	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,001	0,001	0,001	0,001

(b) standard deviation of execution times

original		*if-converted*		*if-converted + nodiv function*		*if-converted + 6 div function*	
all zero - all one	*regular-random*	*all zero - all one*	*regular-random*	*all zero - all one*	*regular-random*	*all zero - all one*	*regular-random*
0,0000	0,0000	0,0000	0,0000	0,2882	0,1350	0,4816	0,2146

(c) p-value of the t-test applied to the execution times

Fig. 9. Execution times and statistical information on the successful mitigation of variable-latency divisions in modular exponentiation.

At this point, it is useful to keep in mind that only the sensitive code in an application needs to be protected. So in most applications, the aforementioned overhead will only be observed on small fractions of the execution time, and might hence very well be negligible in the total execution time.

With respect to portability, this solution can be extended to provide security portability over all existing processor versions. For each different processor version, a specific secured code version can be provided in the software. When the software is executed, the `cpuid` instruction is used to query the processor for its version in order to invoke the appropriate code version. Obviously this will introduce some additional performance overhead as well as significant code size overhead. With respect to future processors, an application could refuse to continue when executed on a processor version for which it has no secured code. This may be not very user-friendly, but at least the security guarantee is not broken. We can conclude that:

—predictable and strict security guarantees insensitive to code fragment properties can be provided using this mitigating transformations;
—the overhead is very high, however;
—true portability can only be provided for existing but not for future processors, and comes with an additional overhead, in particular in the form of increased code size.

4.3. No-Ops for Avoiding Variable Interaction between Memory Operations

As a final experiment, we report on the mitigation based on no-ops against variable execution times caused by the variable interaction between consecutive store and load operations. For this experiment, we let our LLVM plugin insert no-ops in between the store and load operations in the example code at the beginning of Section 2.3.

The resulting execution times for different offsets between store and load addresses are depicted in Figure 10. These execution times are measured for a store address

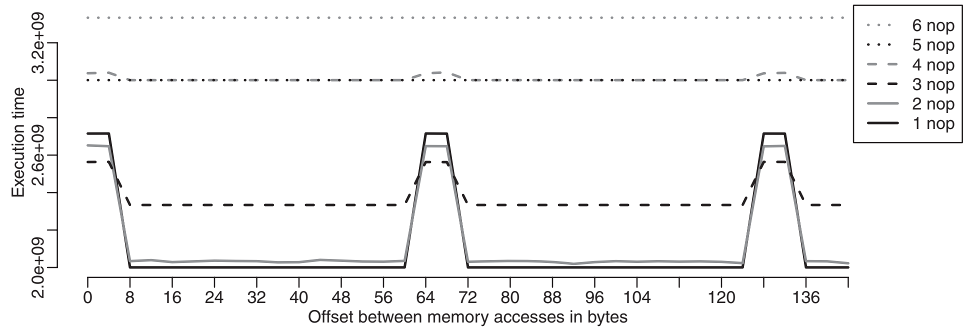


Fig. 10. Execution time (number of cycles on a Core 2 Duo processor) of a loop consisting of a pair of store/load instructions, in function of the offset between the accessed loop-invariant memory locations, for an increasing number of no-ops (0 to 7) inserted in between them.

that is aligned on a 64-byte boundary, corresponding to the top row of Figure 3. The different lines correspond to different numbers of no-ops inserted, from zero to seven, from bottom to top.

From five no-ops on, the t-tests showed that the curve for this experiment became indistinguishably flat, and thus that the timing behavior did not leak any information about the addresses accessed by the memory locations. For other similar code fragments, similarly looking results were obtained, indicating that with enough no-ops inserted where needed, this time side channel can be closed.

As with the previous solution, the performance overhead can be quite big. In this experiment, it is about a factor 1.85. Again, however, this overhead is limited to the code fragments to be protected.

This type of mitigation proves to be very predictable, portable, and insensitive to specific code fragment properties. So we can conclude that

—the technique is portable, predictable and insensitive to code fragment features when the overhead is not minimized, i.e., when a number of no-ops is inserted that is guaranteed high enough;
—the technique does come with a significant overhead.

4.4. Feasibility of Compiler-Based Mitigation

We studied compiler support for mitigating side channel attacks because compared to the manual mitigation of algorithms or source code, automated mitigation in a compiler offers the potential advantage of increasing the developer's productivity. Compared to x86 hardware support to mitigate side channels, which is currently only available for very specific cryptographic computations such as AES table look-ups [Gueron 2008], generic compilers offer the potential advantage of being able to protect any code that handles sensitive data.

However, our experiments have demonstrated that static compilers cannot always provide the highest level of security at low performance overhead. When both are needed, the developers remain responsible for ensuring that their implementations do not leak information. Our experiments have shown that this is not a simple task, and that the developers need to be aware of many computer architecture artifacts. Occasional security programmers are therefore not recommended to try to develop ad hoc solutions, but to reuse existing libraries into which the security community has grown confidence instead. Alternatively, hardware designers might be persuaded to provide hardware support in the future. Support for fixed-latency arithmetic is available on

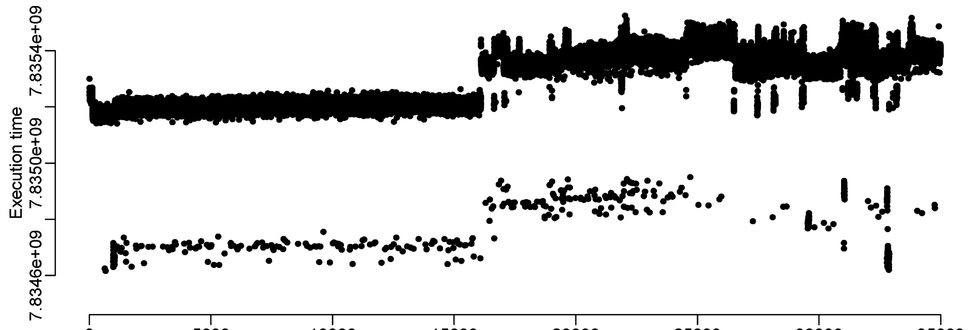


Fig. 11. Phased execution time behavior of execution time over 2 days. On the X-axis, the order of measurements is indicated, on the Y-axis the execution time in number of cycles.

the ARM architecture [ARM Limited 2004] but as of today, not on x86 processors. For other potential side-channels such as load/store forwarding, we know of no existing hardware support. Given the complexity of memory data paths on modern out-of-order processors, we believe further research is needed to assess the cost of such support.

Lacking hardware support in existing processors, our experiments have demonstrated that when a very high level of overhead is acceptable, static compilers are in fact able to provide leakage free solutions that are portable over existing processors. When a low level of security is needed and a limited amount of overhead is acceptable, such as when the average execution times need to be similar but not identical, static compilers can also provide portable solutions.

Anywhere in between, performance-wise as well as overhead-wise, static compilers can only provide non-portable solutions. Moreover, they can only do so when the infrastructure and development time is available to iteratively generate and test many specific instances of the mitigations: iterative code generation and testing is the only option to get high confidence in the provided level of security.

Reliable testing is a problem on its own, however. To validate the absence of any measurable correlations between secret data and execution times, extremely accurate and precise test measurements must be done. It is known from the literature that extreme care needs to be taken to measure supposed performance improvements [Mytkowicz et al. 2009] and that rigorous statistical analysis is needed [Georges et al. 2007]. In order to conduct our experiments and get trustworthy timing results, we took the following precautions: reduce the number of interrupts, for example by disconnecting the network cable and other peripherals like keyboard, and by disabling the USB ports; disable a number of operating systems features such as clock frequency scaling, turbo boost mode, address space layout randomization, and dynamic linking; stop a number of services such as deamons and cron jobs; pin software to one specific core on the multicore processors and disable hyperthreading where available; make sure the software measurement environment is invariable, including the length of all variables in scripts, paths, inputs, environment variables, etc.

Even when all these precautions had been taken, the resulting time measurements were sometimes not usable. For example, the graph in Figure 11 depicts over 30.000 consecutive time measurements of a single program executed over 30.000 times on the same input in an experiment running for two days. Despite of all the precautions taken, there is a clear phase behavior that manifests itself after one day and that we cannot explain. Furthermore, even within a single phase (e.g., day one of the measurements) there are a number of outliers that clearly demonstrate that the execution times are not distributed according to a Gaussian distribution. In another experiment, we used a bootloader developed in house to run our microbenchmarks on bare metal, i.e., without installing any operating system. For both the bare metal and the OS-supported measurements, we measured bimodal distributions. Surprisingly, however, we noticed that the standard deviation on the measured timings (measured by means of processor time stamp counters) was two orders of magnitude bigger for the version running on the bare metal than for the version running on top of the operating system.

Compared to execution times considered relevant in performance-oriented compiler research, the relative changes in behavior observed in Figure 11 are extremely small, as are all standard deviations we observed in our experiments. Such changes would probably go unnoticed in typical compiler research. To provide strict security guarantees, however, such small differences are relevant.

Even if such testing is considered an option for static compilers, it certainly is no option for dynamic compilers. So we have to conclude that the range of mitigations and requirements for which static compilation is not feasible because of portability issues, also poses fundamental problems for dynamic compilers. In the latter case, the problem relates to a lack of predictability and testability of the provided security.

In the cases where static compilers can provide some portable, guaranteed higher levels of security, dynamic compilers can do so as well. Dynamic compilers will be able to do so at a lower code size overhead and at a lower performance overhead, for example because they know exactly how many division latency classes the used processor has and because they know the precise number of no-ops that needs to be inserted for that processor (rather than the maximum number of no-ops needed for all possible processors). However, the performance overhead will still be considerable.

5. CONCLUSION AND FUTURE WORK

In this paper, we discussed several ways in which variable-latency instructions on modern x86 processors can leak timing information that is useful for time-based side channel attacks on cryptographic software. We discussed several potential code transformations to mitigate these side channels. Some of those transformations provide strong protection, albeit that they introduce significant overhead, in particular when they have to protect the software on a range of microarchitectures. And in any case forward portability remains a problem.

Other transformations can provide weak or in some cases stronger but non-portable protection. The effectiveness of a concrete transformation is unpredictable and highly sensitive to the precise shape of the code to be protected and to the microarchitectural details of the processor architecture. Combined with the difficulty to test the effectiveness, we most conclude that compiler-based mitigation is not practical in many contexts, and in particular not in those contexts with strict security requirements.

For that reason, future work should look into the potential of hardware support, as it already exists on ARM architectures, and on the automatic exploitation of that support in compiler backends.

Complementary, we will also research the potential of integrating dynamic resource allocation with dynamic code generation. Side-channel-aware code generators will limit the amount of information that code leaks into side channels where necessary, while side-channel-aware resource allocators will close and block side channels by adapting the resource allocation and scheduling of potentially attacking processes. By integrating the code generation and resource allocation, we hope to achieve a more efficient and more effective co-optimization of performance and security.

REFERENCES

Acıiçmez, O. 2007. Yet another microarchitectural attack: exploiting I-Cache. In *Proceedings of the ACM Workshop on Computer Security Architecture (CSAW'07)*. 11–18.

ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. 2010. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES'10)*. 110–124.

ACIIÇMEZ, O. AND KOÇ, Ç. 2006. Trace-driven cache attacks on AES. In *Information and Communications Security*. Lecture Notes in Computer Science Series, vol. 4307, 112–121.

ACIIÇMEZ, O., KOÇ, Ç., AND SEIFERT, J.-P. 2007a. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*. 312–320.

ACIIÇMEZ, O., KOÇ, Ç., AND SEIFERT, J.-P. 2007b. Predicting secret keys via branch prediction. In *Topics in Cryptology, the Cryptographers Track at the RSA Conference (CT-RSA'07)*. 225–242.

ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, Ç. 2007. Cache based remote timing attack on the AES. In *Topics in Cryptology, The Cryptographers Track at the RSA Conference (CT-RSA'07)*. 271–286.

ARM Limited 2004. *ARM7TDMI Technical Reference Manual* (Revision r4p1). ARM Limited.

BATINA, L., GIERLICHS, B., PROUFF, E., RIVAIN, M., STANDAERT, F.-X., AND VEYRAT-CHARVILLON, N. 2011. Mutual information analysis: a comprehensive study. *J. Cryptology 24,* 2, 269–291.

BAYRAK, A. G., REGAZZONI, F., BRISK, P., STANDAERT, F.-X., AND IENNE, P. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. 230–235.

BERNSTEIN, D. J. 2005. Cache-timing attacks on AES. Tech. rep., The University of Illinois at Chicago.

BONNEAU, J. AND MIRONOV, I. 2006. Cache-collision timing attacks against AES. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES'06)*. 201–215.

BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, rep. 2006/052.

BRUMLEY, B. B. AND HAKALA, R. M. 2009. Cache-timing template attacks. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT '09)*. 667–684.

BRUMLEY, B. B. AND TUVERI, N. 2011. Remote timing attacks are still practical. Cryptology ePrint Archive, rep. 2011/232.

BRUMLEY, D. AND BONEH, D. 2005. Remote timing attacks are practical. *Computer Netw. 48,* 5, 701–716.

COKE, J., BALIG, H., COORAY, N., GAMSARAGAN, E., SMITH, P., YOON, K., ABEL, J., AND VALLES, A. 2008. Improvements in the Intel Core 2 processor family architecture and microarchitecture. *Intel Technol. J. 12,* 3, 179–192.

COPPENS, B., VERBAUWHEDE, I., DE BOSSCHERE, K., AND DE SUTTER, B. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*. 45–60.

CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. 2009. Opportunities and limits of remote timing attacks. *ACM Trans. Info. Syst. Sec. 12,* 3, 17:1–17:29.

DHEM, J.-F., KOEUNE, F., LEROUX, P.-A., MESTRÉ, P., QUISQUATER, J.-J., AND WILLEMS, J.-L. 1998. A practical implementation of the timing attack. In *Proceedings of the International Conference on Smart Card Research and Applications (CARDIS'98)*. 167–182.

DOWECK, J. 2006. Inside Intel Core microarchitecture and smart memory access. Tech. rep., Intel Corporation.

FOG, A. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Tech. rep., Copenhagen University of Engineering.

GEORGES, A., BUYTAERT, D., AND EECKHOUT, L. 2007. Statistically rigorous java performance evaluation. *SIGPLAN Notices 42,* 10, 57–76.

GIERLICHS, B., BATINA, L., TUYLS, P., AND PRENEEL, B. 2008. Mutual information analysis. In *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'08)*. 426–442.

GRANLUND, T. 2011. Instruction latencies and throughput for AMD and Intel x86 processors. Tech. rep.

GROSZSCHAEDL, J., OSWALD, E., PAGE, D., AND TUNSTALL, M. 2009. Side channel analysis of cryptographic software via early-terminating multiplications. In *Proceedings of the 12th International Conference on Information Security and Cryptology (ICISC'09)*. 176–192.

GUAJARDO, J. AND MENNINK, B. 2010. Towards side-channel resistant block cipher usage or can we encrypt without side-channel countermeasures. Cryptology ePrint Archive, rep. 2010/015.

GUERON, S. 2008. Advanced encryption standard (AES) instructions set. Tech. rep., Intel Mobility Group.

GULLASCH, D., BANGERTER, E., AND KRENN, S. 2010. Cache games - bringing access based cache attacks on aes to practice. Cryptology ePrint Archive, rep. 2010/594.

HEDIN, D. AND SANDS, D. 2005. Timing aware information flow security for a Javacard-like bytecode. *Electron. Notes Theoret. Comput. Science 141,* 1, 163–182.

INTEL CORPORATION 2010. Using Intel VTune performance analyzer to optimize software on Intel Core i7 processors. Intel Corporation.

INTEL CORPORATION 2011. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

JOYE, M. AND YEN, S.-M. 2003. The montgomery powering ladder. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*. 291–302.

KOCHER, P. C. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96)*. 104–113.

KOCHER, P. C., JAFFE, J., AND JUN, B. 1999. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'99)*. 388–397.

KÖPF, B. AND BASIN, D. 2007. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM CAonference on Computer and Communications Security (CCS'07)*. 286–296.

KÖPF, B. AND DÜRMUTH, M. 2009. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF'09)*. 324–335.

LATTNER, C. AND ADVE, V. 2003. LLVM: A compilation framework for lifelong program analysis & transformation. Tech. rep., Univ. of Illinois at Urbana-Champaign.

LAURADOUX, C. 2005. Collision attacks on processors with cache and countermeasures. In *Proceedings of the Western European Workshop on Research in Cryptology (WEWoRC'05)*. 76–85.

MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 2001. *Handbook of Applied Cryptography*. CRC Press.

MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the International Conference Information Security and Cryptology (ICISC'05)*. 156–168.

MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. 2009. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 265–276.

NEVE, M. AND SEIFERT, J.-P. 2007. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography (SAC'06)*. 147–162.

OSVIK, D. A., SHAMIR, A., AND TROMER, E. 2006. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology, The Cryptographers Track at the RSA Conference (CT-RSA'06)*. 1–20.

RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 199–212.

SHEN, J. AND LIPASTI, M. 2005. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill.

UHSADEL, L., GEORGES, A., AND VERBAUWHEDE, I. 2008. Exploiting hardware performance counters. In *Proceedings of the 5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08)*. 59–67.

WANG, Z. AND LEE, R. B. 2006. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 473–482.

WANG, Z. AND LEE, R. B. 2007. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Architec. News 35,* 2, 494–505.

Received July 2011; revised October 2011 and November 2011; accepted December 2011