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 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.

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 back-end. The transformations we propose are similar to those proposed by Molnar et al., but we apply them quite differently.

- Using a prototype implementation in the back-end of the LLVM compiler [31], we evaluate the cost and efficiency of a practical semi-automated implementation that relies on conditional execution as supported by the x86 architecture.

The remainder of this paper is structured as follows. Section II discusses program properties on which the execution time may depend. In other words, this section discusses potential side channels that can be the subject of timing-based attacks, revealing two potential side channels on recent x86 implementations that have not been reported so far. Section III details our approach to eliminate key-dependent control flow from applications in a compiler back-end. Section IV evaluates our approach, and Section V draws conclusions.

II. Execution Time on Modern Processors

This section discusses the influence of software and hardware properties on the timing behavior of software. To derive properties about a secret key K, timing-based attacks exploit the timing behavior of a program executed on a secret key K and on other data D that is under control of the attacker. To defend against such attacks, the timing behavior of an algorithm or program should be made independent of the value of the key K. In general, the time behavior of a program can depend on the control flow of a program, on its data flow, and on its contention over resources the program has to share with other running programs. If any of those aspects of a program's execution depends on the secret key K, the timing behavior may leak information about K.

A. Control Flow

Examples of control flow properties that influence the time behavior on modern processors [15] are

- the number and mix of executed instructions, their latencies and their resource conflicts during execution, i.e., the pipelining behavior of instructions in the instruction window,
- the number of instruction cache misses,
- the number of branch prediction misses.

Consider, for example, the following straightforward C implementation of modular exponentiation:

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

This do-while loop iterates over the bits in the exponent, and for each bit the `result` variable is updated if and only if that bit was set to one. On most modern processors, the execution time of this fragment depends on the number of ones in the binary representation of the exponent.

A sufficient condition for avoiding any influence by a key K on the time behavior of a program through control flow properties is that the control flow is independent of key K. In other words, when the control flow transfers in a program are independent of key K, the control flow itself through the program is guaranteed not to leak any information about K. Section III will propose compiler techniques to provide sufficient such guarantees by means of conditional execution. Assuming that there will be no key-dependent control flow left in the program after the proposed technique has been applied, the remaining key-dependent timing behavior can only result from key-dependent data flow. So in order to argue that the method proposed in Section III suffices to remove all key-dependent behavior, we first have to be convinced that key-dependent data flow poses no problems. This is studied in the remainder of this section.

B. Data Flow

If timing does not depend on a secret key through control flow, it can still depend on the key through data flow properties. Some of such potential properties are:

- data dependencies between instructions through registers,
- data dependencies between instructions through memory,
- variable instruction latencies that depend on the values and occurrence of earlier computations or on the value of the instruction operands itself.

All of these properties relate to the complex behavior of out-of-order execution pipelines in modern processors and the available instruction-level parallelism in the code being executed [15]. One well known example of the latter property is data cache behavior: depending on which cache lines are accessed and on the order in which they are accessed, more or less cache misses will be observed. Memory accesses are discussed in Section II-D. Here we focus on arithmetic and logic instructions.

On modern processors, very few arithmetic/logic instructions are implemented such that their execution latency depends on the operand values. Unfortunately, recent Intel processors do have such instructions. On the Intel Core Duo implementation of the x86 architecture, the micro-coded division algorithm that implements the division instruction (that computes a quotient as well as a remainder) exploits opportunities for so-called early exit [16]. The divide logic calculates in advance the number of iterations that are required to accomplish the micro-coded division, and once the required number of iterations is reached the divider wraps up the results. As we will see in Section IV, this data-dependent instruction latency is an important side channel.

Besides the aforementioned integer division instructions, floating-point division instructions and floating-point square-root instructions (which are not normally used in cryptographic code), Intel's documentation [17] indicates that there are no other variable-latency arithmetic or logic instructions on modern Intel x86 processors, including the Atom processor that targets the mobile applications market. A number of instructions with variable-latency have been discussed in the past as they occured on other processors, such as rotation and shift instructions as well as multiplications (on older x86 implementations [17]), so apparantly this behavior is not all that rare. The danger of this behavior for software security has also been recognized before [18]. Two types of software workarounds for this problem can be considered.

First, one can try to add compensation code that, when executed together with the variable-latency instruction, always results in the same execution time. For complex instructions such as division instructions, this will come at the price of severely lower performance. The reason is that the test that analyzes how many cycles are needed for a particular division is quite complex. Basically, the number of cycles depends on the number of bits in the quotient being set to one [16]. In addition to a complex test to be inserted to count this number of bits, the compensation code also needs to be able to compensate for a range of latencies. So it is clear that the analysis and compensation code would be very time-consuming. Moreover, it is unclear whether it is possible at all to design such a compensation code fragment. Due to the complex manner in which out-of-order processors execute code fragments, this is an open question for which we have yet not found a definitive answer. One potential solution we tried consisted of generating a code fragment with a data dependence graph in which a single instruction is dependent on both the original division and on a division that is guaranteed to take the longest number of cycles possible. The rationale of this attempt was that the processor would try to execute the two divisions at least partially in parallel, in which case the slowest division would determine the total execution time of the code fragment. However, because dividers consume a lot of chip area, only one is available in a processor core. So the two divisions are never executed in parallel, and the variable latency division still determines the total execution time.

The second workaround is to avoid the use of variable-latency instructions completely. In the case of divisions, this can be accomplished by implementing a division algorithm in software that only consists of simpler instructions with fixed latency (such as additions, subtractions, multiplications and shifts), and then ensuring that the control flow in that division code is independent of its inputs. This is the solution we have implemented and which will be evaluated in Section IV. Obviously, this solution also introduces a significant performance overhead. Depending on the fraction of an application's execution time that is spent in the key-dependent code fragments that need to be transformed, this overhead will be considered acceptable or not.

A third option would be to disable the early-exit optimization of variable-latency instructions in hardware. This could be permanent, or it could be software-controlled in similar ways to other processor features that are controlled by software. For example, on the x86 architecture the operating system can temporarilly disable interupts with the `cli` instruction, and it can enable them again by issuing a `sti` instruction. Similar instructions could be introduced for disabling/enabling early-exit in division instructions. The programmer or compiler then simply has to surround sensitive regions with these instructions. This is similar to cache locking to defend against cache-based side-channel attacks [6], [7].

C. Register Dependencies

Suppose that we can avoid any key-dependent control flow, and hence any dependence of timing on control flow by performing code transformations in the compiler. What then remains to be questioned are the effects of data flow on a program's execution time.

Consider the two following loop bodies (in which destination operands are specified left):

```
body 1:   mov ecx, edi
          add eax, ebx

body 2:   mov eax, edi
          add eax, ebx
```

In both loop bodies, the second instruction adds the value in register `ebx` to the value in register `abx`. If the first loop body is executed in a loop, all additions in subsequent iterations are dependent on each other: register `eax` serves as a kind of accumulator, to which the value in `ebx` is added repeatedly. In the second loop body, each iteration starts with a fresh value being copied into `eax`. By consequence, when that loop body is executed in a loop, the additions in the different iterations have become independent of each other. Out-of-order processors that apply register renaming will detect this, which allows them to execute the second loop much faster, as they can then execute a number of additions in parallel. We validated this observation experimentally by measuring that the execution of an unrolled loop of loop body 2 on an Intel Core Duo was 40% faster than the execution of an unrolled loop of body 1.

Now consider the following loop body:

```
body 3:   test edx,edx
          cmovcc eax, edi
          add eax, ebx
```

In this fragment, the `test` operation sets condition flags that describe properties of the value in `edx`: its sign, whether the value was zero or not, etc. If the right condition flag is set, the `cmovcc` instruction[1] is executed, and the value of `edi` is copied into `eax`. In that case, the loop performs the same computation as loop body 2 above. If the value of `edx` is such that the `cmovcc` instruction is not executed, `eax` is not overwritten. In that case the loop body basically performs the same computations as loop body 1 above. Since loop bodies 1 and 2 had different timing behavior, one naturally wonders whether the value of `edx` can influence the timing behavior of loop body 3.

On an Intel Core Duo, we measured the execution time of this loop body in an unrolled loop for different values of `edx`, i.e., for the case in which it the conditional move is executed, and for the case in which it is not executed. No significant

[1]In real code, the `cc` in `cmovcc` is replaced by a real condition. For example, `eq` is an instance of `cc` that specifies that the zero flag will be checked.

timing differences were observed, indicating that the answer to the above question is no. So whereas loop body 1 and loop body 2 had different dependencies between instructions, resulting in different timing behavior, the value of `edx` in loop body 3 has no influence on the timing behavior.

The reason is that x86 processors implement the `cmovcc` instruction as a kind of multiplexor instruction.[2] This instruction has two source operands, one of which is implicitly the same as the destination operand. From these two source operands, one is selected based on the condition flags, and then written to the destination operand. In loop body 3, the destination operand is `eax`, and the source operands are (the implicit) `eax` and `edi`. By implementing the `cmovcc` instruction like a multiplexor instruction, the `cmovcc` instruction in loop body 3 always depends on the `test` instruction, and the `add` instruction always depends on the `cmovcc` instruction, irrespectively of whether (from the programmer's point of view) the conditional move gets executed or not. In other words, with this implementation of the `cmovcc` instruction, the operation dependency chain as experienced by the processor does not depend on the condition flags or on any other data occurring in the program's execution. As a result, the timing behavior does not depend on the value of `edx`.

One can wonder whether this analysis holds for all processors. For all x86 processors we know and for ARM's recent out-of-order Cortex-A9 core [19], which is to the best of our knowledge the only out-of-order architecture supporting full conditional execution, the analysis holds.[3] The reason is that these out-of-order processors rely on the reordering of operations to achieve high performance. Fundamentally, these processors try to execute instructions as soon as they can. This means that the processor will consider an instruction ready for execution as soon as it can somehow determine that there are no more true data dependencies that require the instruction to wait for the result of other instructions stil being executed. The technique used to differentiate true data dependencies from false data dependencies is called register renaming [15]. Register renaming is simplified significantly if the processor knows early on in the processor pipeline which instructions depend on which other instructions. So register renaming is simplified by implementing a conditional move instruction by means of a multiplexor instruction. The drawback of this implementation is that even when the processors somehow knows beforehand that the condition flag is false, the processor cannot exploit this information to get rid of superfluous dependencies. On most processors this drawback is not big enough to warrant the implementation of more complex register renaming techniques. So it safe to assume that on x86 processors and on many other out-of-order processors, such as the ARM Cortex-A9, conditional execution will not introduce data-dependent timing behavior as long as the conditional execution is limited to move instructions.

Alternative micro-architecture techniques have been proposed in the literature [20], however, that do enable processors to optimize the code being executed for false condition flags. In doing so, these proposed techniques do make the timing behavior dependent on data flow. To the best of our knowledge, such techniques have not yet been applied in existing commercial processors. But when they would be applied to conditional move instructions, they would break our proposed solution to rely on conditional execution to get rid of data-dependent timing behavior.[4]

D. Data Dependencies through Memory

Consider the following program fragment:

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

If 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 such a loop, the cache behavior will not influence the timing behavior significantly if the loop has enough iterations.

We measured the timing behavior of such a loop on an Intel Core Duo, loading and storing 4-byte `int` values, and we varied the displacement between [`ebx`] and [`ecx`] over a large range in steps of 4 bytes. The resulting execution times are depicted in Figure 1. When the store goes to the same address as the load (displacement 0), a higher execution time is seen than when, e.g., there is a displacement of 12 bytes between the two accesses. The reason is a micro-architectural feature called load bypassing [15]. When the store instruction and the load instruction access the same memory location, the out-of-order processor detects a data dependency between them by comparing the addresses of the locations at which they access the memory. This dependency forces the processor to let the load wait for the store to finish executing. When the address comparison indicates that there is no such dependency, as when the displacement is 12 bytes, the load can be executed together with the store without having to loose time waiting.

Surprisingly, we observe the same slowdown whenever the displacement modulo 64 is in the range [0,7]. When we repeated the experiment for loads and stores of other widths, such as with 1-byte `char` accesses, we observed a similar slowdown. This illustrates that even independent memory operations that always hit in the cache can still cause timing effects as if they are dependent. These effects are not related to the particular combinations of cache lines or cache banks that are accessed. Instead these effects are caused by so-called *pessimistic load bypassing* [15]. Pessimistic in this context means that the processor only compares a few bits in the

[2]We did not find public information on this subject, but the correctness of our findings and our assumption on the implementation of the `cmovcc` instruction was confirmed by Intel engineers.

[3]For the Cortex-A9, ARM engineers confirmed this in private communication.

[4]If those techniques would be applied to the conditional execution of instructions other then moves, they would not break our defense mechanism since in that case we would simply instruct the compiler not to generate code with such conditional instructions wherever secret keys needs to be protected.

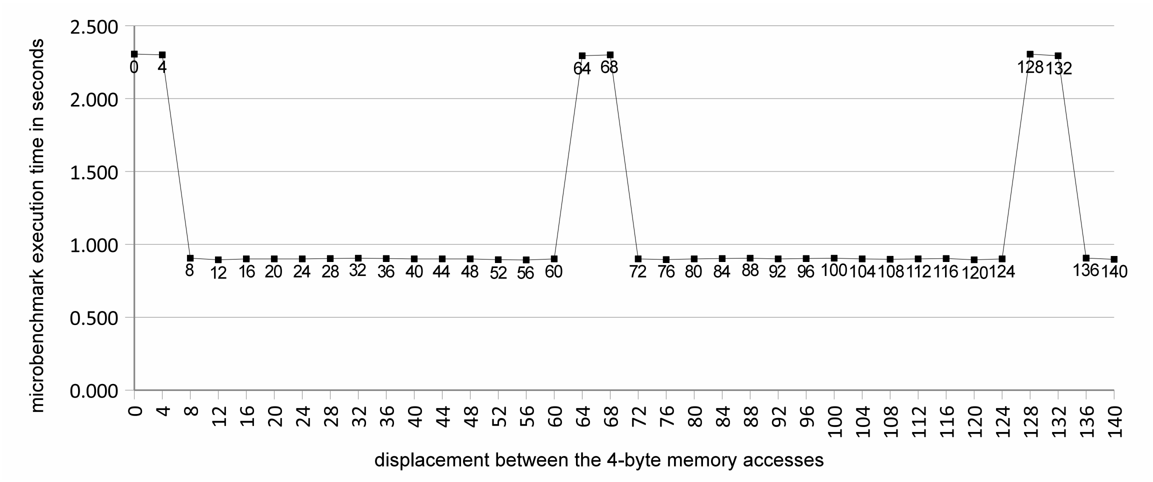

Fig. 1. Execution times of a microbenchmark loop with 4-byte load and store instructions executed for varying displacements between the accessed locations.

memory addresses to decide whether or not two memory accesses depend on each other. As soon as those bits are identical, the processor pessimistically concludes that there may be a data dependency. The execution is then slowed down, even when there is in fact no need to. In the Intel Core Duo, the pessimistic address comparison is limited to bits 3-5 of the addresses. For example, when comparing two accesses at addresses 0xcf0001c0 and 0xaffff307, both of which have value 000 for bits 3-5, the Intel Code Duo's load forwarding will pessimistically assume that the two accesses go to the same address, and it will slow down the execution, as can be observed from the timing results in Figure 1.

For our purpose of defending against timing attacks, this observation implies that even if we can avoid all timing dependencies on cache behavior (by locking or disabling the cache [6], [7] or by using randomized cache addressing [6], [7], or by using scratch-pad memories instead of a cache, or by replacing, e.g., AES table look-ups with new hardware instructions [21]), we still need to take care of other memory dependencies. These dependencies can be true data dependencies through memory, but they can also be dependencies between seemingly independent operations that are caused by micro-architectural pipeline implementation details.

E. Resource Contention

We already mentioned branch prediction and cache behavior as possible origins of control-dependent and data-dependent timing. But in the above discussion, we limited ourselves to intra-thread dependencies: dependencies of a program thread's time behavior on its control flow and data flow.

In modern processors, many resources are shared between multiple threads being executed. When this is the case, contention for those resources can result in inter-thread timing dependencies, in which case the execution of one thread influences the timing behavior of other threads. If an attacker has no direct access to the timing of a thread under attack, he can observe other threads contending for resources to derive information instead.

Many known attacks are based on this observation. For example, algorithms that implement AES by means of table lookups have been attacked by running other attacking threads in a shared cache. In the attacking thread, the shared cache is written and read in a specific pattern that interferes with the memory operations of the thread under attack [5], [8]. By observing this interference from within the attacking thread, information is obtained on the specific table elements that are accessed in the AES implementation, from which information secret keys can be derived. Similar attacks have been devised by means of shared branch prediction tables [11], and by means of contention for functional units in modern processors that support simultaneous multithreading. For example, when a code fragment similar to the above one was run together with an attack thread executing many multiplications on an SMT core, the number of ones in the thread under attack could be derived from the timing behavior of the attack thread [6]. As such, existing side channels not only include a thread's own execution time, but also its influence on the execution time of (fragments in) other threads through resource contention.

This paper studies timing-based side-channels and discusses software solutions to avoid information leakage through those channels. Other side channels, such as those based on power consumption, are mostly neglected in this paper. We believe this does not diminish the relevance of the subject of this paper, as there are many cases in which timing-based attacks are possible, and in which power-based attacks are not, simply because an attacker has no physical access to the device running the thread under attack, such as in remote attacks over the Internet [1], [10].

III. Eliminating Key-Dependent Control Flow

In order to eliminate side channels related to control flow, we will adapt the compiler back-end to eliminate all control flow transfers that depend on secret keys. If we succeed, then in the compiled code the same execution path will be followed for every key with which a piece of data is encrypted or decrypted. This path, and the timing behavior following from this control flow, will then not reveal any information about the secret key to an attacker. Indeed, branch prediction is then independent of the key, as are instruction mix, instruction order, data dependencies through registers, and instruction fetching from the instruction cache.

The only remaining possible side channels then relates to data cache flow and data dependencies through memory. For the former, we will assume that existing techniques such as blocked caches [6], [7] or random cache addressing [6], [7] can overcome them. For the latter, we will propose potential solutions in Section III-F.

A. Conditional Execution

For simple, acyclic code fragments not containing any function calls, conditional execution provides an excellent mechanism to get rid of key-dependent control flow. With conditional execution, control flow dependencies on diverging non-cyclic execution paths such as if-then-else constructs can be transformed into data flow dependencies on a single path.

On architectures that support full conditional execution, either by means of condition flags that activate or deactivate instructions or by means of real predicates that guard instructions, this transformation is trivial. Consider again the code fragment of Section II-A. The relevant part of that code can be rewritten as follows:

```
c = (exponent>>i) & 1;
if (c) {
  result = (result*a) % n;
}
```

which can in turn be rewritten as

```
c = (exponent>>i) & 1;
if (c) tmp = result*a;
if (c) result = tmp % n;
```

in which the last two lines will be compiled into two conditional instructions. The conversion from code with branches to code with conditional execution is called *if-conversion* [22], and it is well known in the field of compiler techniques, in particular for VLIW and EPIC types of architectures. Composing more complex conditional structures such as nested if-then-else structures poses no problem. It only requires a few more instructions to compute the right predicates or set the correct condition flags, depending on the expressiveness of the architecture with respect to conditions [23].

Two remarks should be made here. First of all, depending on potential side-effects of conditional instructions, these instructions may sometimes be converted to non-conditional ones again. In the above example, it is no problem to remove the condition of the first statement:

```
c = (exponent>>i) & 1;
tmp = result*a;
if (c) result = tmp % n;
```

This predicate elimination, also called predicate speculation [24] is fine because the transformed instruction only affects the local variables `tmp` that does not define the global program state, and because it cannot cause side effects such as exceptions.

However, consider the following contrived, but exemplary program fragment:

```
if (a != 0)
  d = 1 + b / a;
```

Rewriting this fragment as follows is incorrect:

```
c = (a != 0);
tmp = b / a;
if (c) d = 1 + tmp;
```

This is incorrect because the division might cause a division-by-zero exception. One potential solution consists of not applying predicate elimination for instructions with side effects.

This brings us to our second important remark: on many architectures, such as on the omni-present x86 architecture, not all instructions can be executed conditionally. On the x86, only `mov` instructions can be executed conditionally. So the above potential solution is not applicable on an x86 architecture. There we have to come up with an alternative solution. This alternative solution, to be applied by the compiler back-end, consists of the following:

1) Before merging a path into other paths with if-conversion, make sure that all instructions in that paths operate on local, temporary variables.
2) Before merging this path with if-conversion, insert so-called *safe-guard* instructions prior to all "unsafe" instructions that may cause exceptions or that change the global state (because they store something to memory). These safe-guard instructions are conditional move instructions that move an appropriate, safe value into the appropriate local variable when this path would not have been executed. So even if such "unsafe" instructions get executed after if-conversion when they would not have been executed in the original code (because another path was taken), the safe-guard instructions will now ensure that these instruction do not cause any harm.
3) At the end of this path, insert conditional move instructions that copy the local variables into global ones.

Consider the following example code fragment:

```
if (c) {
  *a = 10;
  d = x/y;
} else {
  b = 10;
}
```

With conditional execution this code would be rewritten as follows:

```
        tmp_a = a;
if (~c) tmp_a = dummy_location;
        *tmp_a = 10;
        tmp_y = y;
```

```
if (~c) tmp_y = 1;
        tmp_d = x / tmp_y;
        tmp_b = 10;
if (c)  d = tmp_d;
if (~c) b = tmp_b;
```

If `c` evaluates to false, the store operation will now write to a dummy location where it does not harm the real program state, and the division will be executed with divisor one, which will not cause an exception. Furthermore, in that case only the assignment to `b` will be executed, but not the one to `d`. If `c` evaluates to true, the store and the division will be executed as they should, and only the assignment to `d` will be executed, not the one to `b`.

The most common instructions that should be safe-guarded this way include divisions, loads and stores. For stores and divisions, one reason has already been mentioned. For loads and stores, an additional reason not mentioned yet is that memory accesses to disallowed addresses will result in page faults. Consider a code fragment

```
if (a != NULL)
  b = *a;
```

Clearly, the load from `a` should only be executed if `a` is not `NULL`. So this needs to be rewritten as

```
        tmp_a = a;
if (~c) tmp_a = dummy_location;
        tmp_b = *a;
if (c)  b = tmp_b;
```

One may wonder whether or not the execution[5] of loads or stores often depends on secret keys. The fact is that big integer functionality for cryptographic purposes, such as RSA, is usually implemented using memory arrays of smaller integers. When computing the result of an operation on such big integers, the final results, as well as the intermediate results, have to be written to memory into these arrays. So in such libraries, conditional loads and stores are as likely as any other conditional operation.

Please note that if-conversion should only be applied when the involved conditional branches depend on a secret key. Whether or not they do can be determined by means of a compiler data flow analysis. In case this analysis lacks the required precision to compute the correct dependencies, the compiler should either conservatively assume that there is a dependency on the secret key, or user annotations of the source code could clarify this for the compiler. In any case, the user should already inform the compiler what exactly constitutes the key from which control flow should be made independent, for example by means of so-called *attributes* or by means of *pragmas*. Requiring the addition of even more user-annotations or user-directives is hence not infeasible.

Please also note that compilers will likely optimize the above code fragment to get rid of the temporary variables. They will always do this conservatively, however, respecting the (correct) semantics of the rewritten code fragment.

Finally, we should point out that our implementation using conditional execution of `mov` instructions relies on the fact that conditional moves have data-independent timing behavior. As discussed in Section II-C, we believe this is a safe assumption.

B. *Cyclic Control Flow Graphs*

Cyclic control flow graphs occur for program fragments containing loops. In these loops, the number of iterations may be key-dependent or not.[6] A compiler can again either detect this by means of static data flow analysis, or it can rely on user-annotations in the program.

In cryptographic practice, it is not common for loops to have a number of iterations that depends on the value of a secret key. So in practice, this situation does not occur all that often. However, some cryptographic libraries are programmed generically, e.g., for different key lengths. Sometimes the number of iterations of certain loops then depends on the length of the key. Very likely, the compiler's data flow analysis will not be able to differentiate between depending on a key's value and depending on a key's width. So in such cases, user-annotations specifying that a loop's number of iterations is not dependent on a key will be required.

In the very rare case that a loop's iteration count does depend on the actual value of a secret key, the only solution consist of letting the compiler determine (or the programmer specify) an upper bound on the number of iterations, and to set the iteration count of the loop to this fixed upper bound. The loop body should then be executed conditionally as indicated in Section III-A. In this case the condition will keep track of whether real iterations are still being executed or whether additional iterations are being executed to reach the fixed upper bound.

Please note that from a performance point of view, increasing the number of iterations in a loop to a fixed upper bound might be detrimental. However, this is unavoidable. If it has a detrimental effect on performance, it had to be the case that there were significant key-dependent differences in execution time in the original code. In order to avoid those, each execution of a rewritten program on data D with whatever key K should have the same execution time, independent of K, which is hence at least the execution time of the slowest run of the original program with key K. So all runs that previously were faster than that slowest run need to become that slow as well, which then may cause the detrimental, but unavoidable run-time overhead.

[5]We mean the fact whether or not a load or store is executed, not the addresses at which they are performed.

[6]In lower-level code, this corresponds to a conditional branch that determines whether the loop is continued or exited. Such a branch, if taken, transfers control back to the loop entry point. As such, it cannot be omitted with simple if-conversion.

C. Function Calls

Any realistic program contains functions and hence function calls.[7] Because big integer libraries usually implement operations on big integers as function calls, this certainly also occurs in cryptographic code.

So it can very well occur that some functions are not invoked on all execution paths in the original code of a program, and that the invocation depends on the value of a secret key. In such cases, the rewritten program should invoke the function in the merged paths independently of the key, or otherwise the execution trace of the rewritten code will still depend on the key.

A first solution to this problem constitutes inlining. When a callee's function body is inlined in a caller's body, the call is removed and the callee's body can be executed conditionally just like the caller's code.

Because inlining may increase the code size of a program significantly, which may not be acceptable for embedded devices with small amounts of memory, an alternative solution is required. It consists of adding an additional parameter to a function which specifies whether or not the function would have been invoked in the original program. Instead of executing a call conditionally, each call will now be executed unconditionally with the additional parameter. Inside the function, all code is then executed conditionally as discussed in Section III-A, but now with the additional condition specified by the added parameter.

For example, consider the following code fragment:

```
void f(int x) {
  *a = x;
}

...
if (c)
  f(10);
```

This would be converted into

```
void f(int x, int c) {
  if (c) *a = x;
}

...
f(10,c);
```

after which the normal if-conversion of Section III-A is applied to function `f()`.

[7]In this paper, we only consider direct function calls. Virtual method calls such as in C++ or Java are rare in cryptographic code, and if they are present, it is usually possible to replace them by conditional function calls after a compiler has analyzed the class hierarchy of a program. This is not possible when run-time class loading is used for classes that were not known at compile time. But in such cases, those classes loaded at run time are not trusted since they might do anything with data fed to them. So in practice, secret keys will never be propagated into such classes, in which case there is no need to adapt them, and hence we can neglect them in this paper. Furthermore, we neglect system calls, as we have not found cases where the execution of system calls is conditioned on values of secret keys.

Please note that in practice, we duplicate a function like `f` and we add the additional parameter to the duplicate only. That way, any call to `f` that is not dependent on the secret key can still invoke the original function version without the additional parameter, which will be better for performance and which requires fewer code to be adapted.

D. Comparison with Molnar's approach

Several approaches have been proposed before to remove key-dependent control dependencies from cryptographic code. Specific code fragments and their alternatives have been studied [25]–[28]. Molnar et al. [13], [14] presented a generic approach relying on source-to-source transformations. Because conditional execution is not available on all architectures, Molnar et al. presented a variation on our use of conditional execution. Consider the following C code fragment, which is very similar to a fragment in their paper:

```
if (n != 0) {
  if (n % 2) {
    r = r * b;
    n = n - 1;
  } else {
    b = b * b;
    n = n / u;
  }
}
```

They rewrite this fragment as

```
m1 = -(n != 0);
m2 = m1 & (-(n % 2));
r = (m2 & (r * b)) | (~m2 & r);
n = (m2 & (n - 1)) | (~m2 & n);
m2 = m1 & ~ m2;
b = (m2 & (b * b)) | (~m2 & b);
n = (m2 & (n / u)) | (~m2 & n);
```

Rather than using conditional execution, Molnar et al. use bit masking. Conditions are used to generate masks `m1` and `m2` that consists of all zeros or all ones, and then they mimic the conditional execution by using the masks. As is obvious from the example, this allows them to handle nested conditions. However, there are very fundamental issues with their approach.

1) Compilers: As Molnar et al. indicate themselves, some compilers will translate the negation operator `!` into conditional branches. They found this to be the case for the very popular GCC compiler, and for that reason they could not use that compiler. Likewise, they found it necessary to write a static verifier that can validate whether the compiled code really provides key-independent control flow. This clearly contradicts their claim that a source-to-source approach is practical.

This also hints to another problem of their approach: if the compiler is able to see through their masking constructs, it will be able to optimize the code to, for example, get rid of unnecessary copy operations involving temporary variables or to propage constant values through the program [29]. But then one also risks that the compiler will be able to optimize the code by reintroducing conditional branches. In other words, if one uses a compiler that is smart enough to still optimize the rewritten code, one risks that the compiler is so smart that it will optimize the code too much (from a security point of view). And if the compiler is not smart enough too see through the masking, it will simply perform no optimizations.

By contrast, in our compiler back-end approach relying on conditional execution, the compiler has already optimized the code using the full potential of the original code that was much easier to analyze. As such, full compiler optimization and guaranteed key-independent control flow can be combined without any problem.

2) Exceptions and side-effects: Molnar et al. treat side effects such as exceptions and store instructions incorrectly. They claim that they can neglect exceptions because they only consider correct programs in the first place, in which no exceptions occur. This is incorrect for at least two reasons.

First, in the above code fragments, in the original code the division would only have been executed if `(n != 0)` and if `(n % 2)` was zero. In the original version, it hence does not matter what the value of `u` is if those conditions are not met. So it is perfectly fine to execute the original fragment with, for example, `n` being 5, and `u` being 0. No division-by-zero exception will occur. In the rewritten code however, the exception will occur, as the division is executed unconditionally in that code. This clearly changes the program behavior.

Secondly, obfuscation techniques exist [30] that rely on exception handling to replace control flow that is easy to reverse-engineer by control flow that is much harder to reverse-engineer. If such obfuscation techniques are used to protect cryptographic code, the exception handling even becomes part of the correct execution of a program. Clearly, exceptions cannot be neglected in this case.

Finally, Molnar et al. do not discuss how to handle conditional function calls, and conditional loads or stores. This is not a fundamental issue of their approach, it is merely an incompleteness, as we believe that similar techniques as the one we use to safe-guard instructions can be used in their approach.

E. A Practical Implementation

In order to make an implementation of our proposed technique practical, we need to present it to programmers in a usable way. If a programmer has to use many different tools, or if he has little influence on how and where the technique is applied, it cannot be considered really useful. As a proof-of-concept, we implemented our technique in a plugin for the compiler framework LLVM [31]. This plugin approach facilitates the integration of our technique in existing compilation flows.

In our proof-of-concept implementation, we have not yet implemented (or adapted) the necessary data flow analyses to let the compiler detect which code should be adapted. Instead, we implemented support for programmer annotations. The simple form of annotations we have implemented so far enables the programmer to specify which C functions should have data-independent control flow. In our implementation, we call this balanced control flow, hence our use of the keyword `balanced`. Consider the following code fragment:

```
int __attribute__((annotate("balanced")))
f(int a) {
  if (a > 0)
    return  1;
  else
    return  -1;
}

int g(int* a) {
  if (a!=0)
    return *a;
  else
    return 0;
}

int __attribute__((annotate("balanced")))
h(int a, int* b) {
  if (a & 2)
    return f(a);
  else
    return g(b);
}
```

In this fragment, functions `f()` and `h()` will be transformed immediately, but `g()` will not. However, because `h()` calls `g()`, a transformed version of `g()` with data-independent control flow will be generated as well, as described in Section III-C which will then be called from within `h()`.

Our current implementation lacks support for recursive function calls and for non-manifest loops, but adding support for those is merely an implementation issue that poses no fundamental challenges. Furthermore, that support is not needed to evaluate the effectiveness and efficiency of our technique.

F. Dealing with Timing Dependency on Memory Accesses

Using conditional execution and safe-guarding instructions, we remove all key-dependent differences in timing behavior due to control flow. However, as indicated in Section II-D, data dependencies or independencies through memory operations might still influence the execution time. This can in particular be the case if load and store instructions at dummy locations (as introduced in Section III-A) result in a timing behavior that differs from the behavior of those same load and store instructions when executed on the real addresses from the original code.

To avoid such behavior, dummy locations should be chosen very carefully. Unfortunately, however, no generic solution to this problem exists, as it may be that no valid real address can be computed. For example, in the last code fragment of

Section III-A, we only know that a real load at address a will not go to address zero. But it may very well be that this limited amount of information is all the compiler can derive. So if any other non-zero address can occur, how is the compiler then supposed to pick a correct dummy location?

We strongly believe that this situation will occur very rarely. In most cases, the real addresses will be known and fixed, and hence dummy locations can be chosen that result in the same pipeline behavior. In cases where the real addresses are not fixed, but known to be available at run time, safe-guarding code can be inserted that will choose an appropriate dummy location at run time, using the real address that is then available.

If even that is not possible, a third approach would be to profile the program off-line, and to let the compiler insert safe-guarding code that chooses dummy locations statistically in such a way that the timing differences to be expected are minimized. In this case, however, no strict guarantee can be provided.

Unfortunately, all of these solutions come with a major drawback: each processor generation or implementation can feature a different mechanism to determines how independent or dependent memory operations are executed. This implies that the compiler needs to know the exact implementation for which it is compiling the code. As such, it is impossible to provide a kind of binary compatibility guarantee for key-independent timing behavior. This is a fundamental problem, not only to our approach but for all approaches that combat timing-based attacks.

Our current prototype compiler implementation in LLVM does not yet support any of these solutions for timing dependencies on memory accesses. Implementing and evaluating these solutions is part of our ongoing research.

G. Feasibility of a Compiler back-end Approach

The proposed elimination of key-dependent control flow transfers will be performed in a compiler back-end. Using a compiler back-end, rather than a source-to-source transformation tool, gives the benefits of being able to apply the technique more easily to programs that are implemented in multiple, different programming languages. Furthermore, a back-end implementation eliminates the risk of the compiler middle-end reintroducing vulnerabilities by undoing source-to-source or intermediate-code-to-intermediate-code transformations in the middle-end or the front-end of the compiler. This guarantee allows compiler middle-ends to be developed independently of the security considerations, which is an important advantage given the huge complexity of modern compilers consisting of millions of lines of code.

One can argue that the source-to-source transformation approach proposed by Molnar et al. [13], [14] is more reliable than our approach because their approach includes a post-pass static code checker that checks if the compiler-generated code has no key-dependent control flow. Hence their trusted code base consists of the static checker only, while ours seems to consist of the full compiler or at least its back-end. So apparently the trusted code base of Molnar et al. is smaller than ours. This is of course a fake argument, as nothing stops us from using the same static checker in our approach.

Now at first sight, it may seem as if our approach is merely undoing or prohibiting compiler optimizations and micro-architectural performance optimizations. This is not the case, however. In the compiler-back-end approach that we propose, we only restrict the applied compiler optimizations to code fragments in which secret keys occur that need to be protected. For all other code constituting an application, the compiler can still apply all its optimizations. Similarly, the back-end only inserts countermeasures against hardware performance optimizations in the code where secret keys need to be protected.

With this paper and its evaluation section, we demonstrate that for at least one advanced x86 implementation, this semi-automated compiler back-end approach is able to make the timing behavior of a program independent of secret keys. In other words, given the right compiler transformations, it is alright for architectures to feature a number of leaking side channels that relate to both control flow and data flow.

Clearly, the required set of code transformations depends on the leaking features of the processor micro-architecture, which might change from one processor generation to the other, as already noted in Section III-F. So for each new processor implementation, the potential leaks have to be studied again, and appropriate code transformations have to be developed and implemented in the compiler back-end(s).

This may seem very unproductive and not practically feasible to cryptography researchers, but it is a familiar situation for compiler developers.

With all major compilers (ICC, GCC, ...), developers can use compiler flags to specify that they want to compile for a specific architecture implementation target. When such a flag is used, the compiler will apply additional optimizations for the specified target and it will tune its generic optimizations towards the specified target. The result is a faster compiled program, and the price to pay is that of lowered portability. Indeed, because the compiler knew that it was generating code for a specific target, it may have generated code that can only run on that target. If the developer chooses not to specify a particular target for his compiler, that compiler will apply a default set of optimizations with default tuning parameters that target a virtual architecture that resembles the least common denominator of all supported implementations of that architecture. The result will be a slower program, as it does not exploit specific features. But because the program only relies on features found on all covered targets, the program can be executed on any of them. In short, developers can trade-off portability for performance by selecting a desired level of target-dependent optimizations.

The development of such target-dependent compiler optimizations typically works as follows. During the development of a compiler, optimizations are first hardcoded for specific processor features when those processors and their new features hit the market. When other processors later arrive on the market with variations of those features, the implemented optimizations are adapted to become more generic and tunable.

Likewise, one can imagine the development of a compiler that applies all known side-channel countermeasures when no specific target micro-architecture is specified, thus introducing a lot of run-time overhead. The application will then run correctly and safely on all possible processor implementations covered by the combined countermeasures, albeit slowly. But when the compiler is invoked for a particular target, it will not apply the countermeasures that are not needed for that particular target. The result will be a faster application, and this will again be at the price of portability.

IV. Evaluation

To evaluate our approach and the extent to which the x86 architecture lends itself for the program counter security model, we used our implementation as described in Section III-E to protect a number of microbenchmarks against timing-based side-channel attacks.

A. Experiments

We used a variety of microbenchmarks to evaluate the efficiency and effectiveness of our proposed approach. Efficiency here corresponds to the performance and code size, i.e., the execution time overhead and code size overhead introduced by the if-conversion and elimination of variable-latency division instructions in the protected software. Effectiveness corresponds to the extent with which the protection is able to make the timing behavior independent of sensitive data such as secret keys. We measured this effectiveness by measuring the largest possible differences in execution times for different inputs, both before and after our code transformations. For the sake of completeness and to demonstrate that our technique can defend against side-channel attacks based on branch predictor behavior [11], we also measured differences in branch prediction behavior. This measurement was performed using performance counters.

The first set of microbenchmarks consist of a set of simple C functions of increasing complexity. This set, of which the code is included in the appendix, allows us to determine the cost of protecting increasingly complex code. Four microbenchmarks `f1`, `f2`, `f3`, and `f4` contain an increasing number of nested `if-then-else` constructs, and thus an increasing number of different execution paths, and two microbenchmarks `memread1` and `memread2` contain memory accesses that have to be safe-guarded with dummy addresses.

The second set of microbenchmarks consists of three handwritten implementations (similar to the code in Section II-A) of modular exponentation as it occurs in, e.g., RSA encryption:

1) The experiment **modexp32** uses 32-bit numbers for a modular exponentiation.
2) This experiment **modexp64** uses 64-bit numbers for a modular exponentiation. This is the native word width on 64-bit platforms such as the Core 2 Duo machine the code was tested on.
3) We implemented a minimal so-called big integer component **modexp256** that can do the computations for the modular exponentiation of 256-bit integers. This is implemented in C++, using calculations on 8 32-bit integers that are stored in memory rather than in registers.

When these microbenchmarks are compiled, the compiler maps the modulo computation onto x86 division instructions.

To demonstrate the effectiveness of our approach, we ran these three modular exponentiation microbenchmarks on inputs consisting of (1) randomly varying modulo values, (2) randomly varying base values, and (3) four different types of exponents.

- In the **all 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` (see the code fragment in Section II-A) 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 by the processor.
- In the **all 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. 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 by the processor. 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, and that the pattern is predicted very well by the branch predictor of the processor.
- 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 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.

Together, these four input sets allow us to study to what extent our proposed transformations are able to eliminate timing dependencies that originate from different amounts of code being executed for different keys or from branch prediction behavior that depends on keys.

Please note that the number of times each loop was invoked per experiment differs for the three microbenchmarks. For each benchmark, the number of invocations was choosen to be a good balance between short experimentation times and accurate measurements.

Finally we applied our approach to a function from the OpenSSL (http://www.openssl.org) library that can be used to implement RSA. The most interesting part of that function called `BN_sub` is included in the appendix. This library code is executed on two different inputs that result in different timing behavior in the original code. These inputs were obtained by running the original code on multiple random-generated inputs, and by selecting the two inputs with the most extreme behavior.

We ran all experiments on an Intel Core 2 Duo machine running at 2.2GHz under the Linux operating system. All versions were executed 20 times on all inputs to collect statistics on the timing behavior and branch prediction. All binary code was generated using LLVM's standard compiler options to generate 64-bit code, except for the OpenSSL code. For the OpenSSL code we disabled some compiler middle-end optimizations because they resulted in if-converted code even in the unadapted compiler. Disabling this if-conversion in the original compiler is not unrepresentative of the typical behavior of optimizing compilers for at least three reasons. Firstly, we verified that other optimizing compilers, including recent versions of GCC, also generate assembly code for this fragment that is not if-converted. Secondly, disabling compiler optimizations is often done to obtain more usable debugging information. Finally, had the code been slightly more complex, the LLVM compiler would also not have applied if-conversion. Because of these three reasons, we believe that compiling this real-world code with some compiler optimizations disabled still makes a good representative of the real-word code to which our proposed technique can be applied.

B. Effectiveness

Table I presents average execution times and average branch mispredications, as well as other statistics on the various modular exponentiation microbenchmarks and on the OpenSSL benchmark for the different inputs.

Table I(a) presents, for each microbenchmark version and for appropriate input set, the execution times averaged over 20 runs. Table I(b) presents the standard deviation of the measured times. Clearly, for all of the benchmarks, the original versions behave quite differently for their different inputs. Hence, these benchmark versions leak information about the secret inputs.

The if-converted benchmarks have significanlty longer execution times, because of the overhead introduced by the if-conversion, but the execution times obtained with the different inputs now display significantly more similarity. The same can be observed for the benchmarks after if-conversion and division elimination, indicating that our approach has indeed removed all timing dependence on the secret inputs.

To assess the confidence with which we can draw the above conclusion, we have performed multiple t-tests. Table I(c) depicts the p-values obtained from t-tests applied to three combinations of inputs: combination all zero - all one, combination regular - random, and combination input1 - input2.

For the 32-bit and 64-bit modular exponentiation, the p-values indicate that we can conclude with high confidence that our approach of if-conversion and division instruction elimination does in fact result in key-independent timing behavior. For the same microbenchmarks, if-conversion alone does not suffice to achieve this. This follows from the fact that the variable-latency division instruction is still present.

For the 256-bit modular exponentiation, however, if-conversion alone proves to be sufficient to eliminate all key-depenent timing behavior. The reason is that in this 256-bit big integer implementation, the divisor of all executed division instructions is the fixed 64-bit value 0x00000000ffffffff. The compiler replaces this divide-by-a-constant by a multiplication and a shift. As as result, no division instructions occur in the compiled code, and we only need to apply if-conversion to eliminate the key-dependent timing behavior.

For the OpenSSL code, the same reasoning holds: there are no division instructions, so we only need to apply if-conversion to make this code's timing behavior independent of the input key.

Tables I(d), I(e) and I(f) present similar statistical information for the number of branch mispredictions measured with performance counters. Similar conclusions can be drawn. Here as well, we can be confident that if-conversion and division instruction elimination are successful in eliminating timing behavior dependences on the secret inputs. Still, two remarks need to be made here.

Firstly, very high standard deviations were obtained in the numbers of measured branch mispredictions with several versions of modexp32. We repeated these experiments multiple times, but never obtained more consistent results. We can not explain why the standard deviations of 1193.2 and 514.8 are as high as they are. Clearly, however, these high standard deviations do not occur in the if-converted code, which is what matters to us.

Secondly, the confidence scores 0.0077 and 0.2701 for the number of branch mispredicts of the if-converted modexp32 and modexp64 benchmarks are relatively low, notwithstanding the fact that there is no key-dependent control flow present in those benchmarks. The explanation for this result is found in the complex pipeline behavior of the Intel Core 2 Duo pipeline. Remember that there still are variable-latency division instructions present in these benchmark versions. The actual latencies of those instructions occurring during the execution of the program have an effect on the number of branches that are fetched by the processor and issued speculatively [15]. So while the number of truly executed branches (so-called retired branches) is independent of the secret inputs of these benchmarks, the number of speculatively issued branches is not. And hence the number of mispredicted branches, some of which were speculative, is not independent either. If division instructions are eliminated as well, this effect does not play anymore, and then we can conclude with much higher confidence that the number of branch mispredictions does not depend on the secret inputs anymore.

C. Performance Overhead

Figure 2 displays the performance overhead of applying if-conversion and, where necessary, the elimination of division

	original				*if-converted*				*if-converted + div elimination*			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	0.785	1.377	1.027	1.223	1.504	1.535	1.524	1.515	26.473	26.473	26.474	26.474
modexp64	0.911	1.816	1.354	1.405	1.847	1.897	1.871	1.877	21.109	21.110	21.109	21.109
modexp256	3.642	7.374	5.532	5.531	16.764	16.757	16.759	16.760				

	input 1	*input 2*	*input 1*	*input 2*
OpenSSL	1.580	2.255	4.606	4.606

(a) average execution times (in seconds)

	original				*if-converted*				*if-converted + div elimination*			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	0.015	0.000	0.001	0.003	0.001	0.001	0.001	0.001	0.007	0.007	0.007	0.007
modexp64	0.000	0.001	0.000	0.000	0.000	0.001	0.000	0.001	0.005	0.005	0.005	0.004
modexp256	0.009	0.011	0.004	0.007	0.029	0.018	0.007	0.007				

	input 1	*input 2*	*input 1*	*input 2*
OpenSSL	0.060	0.013	0.015	0.025

(b) standard deviation of execution times

	original		*if-converted*		*if-converted + div elimination*	
	all zero - all one	*regular-random*	*all zero - all one*	*regular-random*	*all zero - all one*	*regular-random*
modexp32	0.0000	0.0000	0.0000	0.0000	0.8525	0.9557
modexp64	0.0000	0.0000	0.0000	0.0000	0.6028	0.9644
modexp256	0.0000	0.0000	0.3539	0.6781		

	input 1 – input 2	*input 1 – input 2*
OpenSSL	0.0000	0.9405

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

	original				*if-converted*				*if-converted + div elimination*			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	8698	11	2539	44043	11	11	11	11	91	91	91	91
modexp64	1173	16	2866	17230	507	507	507	507	65510	65510	65509	65509
modexp256	1155	25521	19249	19123	270	269	269	267				

	input 1	*input 2*	*input 1*	*input 2*
OpenSSL	14	55311	17	17

(d) average number of branch mispredictions (x1000)

	original				*if-converted*				*if-converted + div elimination*			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	1193.2	0.1	57.1	514.8	0.1	0.1	0.1	0.2	0.6	0.5	0.7	0.8
modexp64	58.4	0.2	7.1	11.2	1.2	1.1	1.0	1.1	0.8	0.5	0.3	0.4
modexp256	38.5	11.7	14.1	19.9	8.0	7.3	8.7	9.4				

	input 1	*input 2*	*input 1*	*input 2*
OpenSSL	0.1	1401.7	0.3	0.3

(e) standard deviation of the number branch mispredictions (x1000)

	original		*if-converted*		*if-converted + div elimination*	
	all zero - all one	*regular-random*	*all zero - all one*	*regular-random*	*all zero - all one*	*regular-random*
modexp32	0.0000	0.0000	0.0077	0.3370	0.9487	0.8205
modexp64	0.0000	0.0000	0.2701	0.6406	0.9916	0.7508
modexp256	0.0000	0.6194	0.6909	0.6106		

	input 1 – input 2	*input 1 – input 2*
OpenSSL	0.0000	0.3569

(f) p-value of the t-test applied to the number of branch mispredictions

TABLE I
STATISTICAL RESULTS OF IF-CONVERSION AND THE ELIMINATION OF VARIABLE-LATENCY DIVISION INSTRUCTIONS

instructions. We averaged this overhead over a large number of pseudo-random inputs to get realistic results. Because the LLVM middle-end itself already if-converted the code in *f2*, and because the code in *f1* only features a single execution path, these fragments undergo no additional transformations in our approach. So we observe no slowdown for them. Suprisingly, the simple functions *f3* and *f4* became faster after if-conversion. This can be attributed to the improved branch prediction. As there are less branches to predict in the converted code, and in particular less branches that depend on pseudo-random inputs, less cycles are lost after mispredictions.

For the other microbenchmarks, increasing complexity results in additional overhead. The maximum overhead observed corresponds to a slowdown with a factor 24.0. This is very large, but it will only occur in those program fragments that (1) involve computations on sensitive keys, and (2) include division instructions. So on the total execution time of real-applications, in particular of applications that only perform

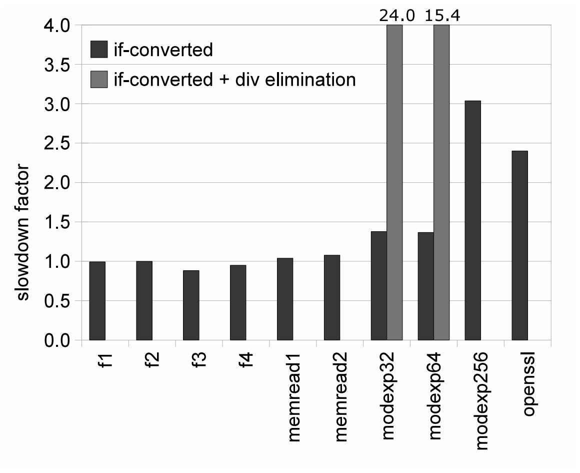

Fig. 2. Average (over pseudo-random inputs) execution slowdown after applying if-conversion and elimination of variable-latency division instructions

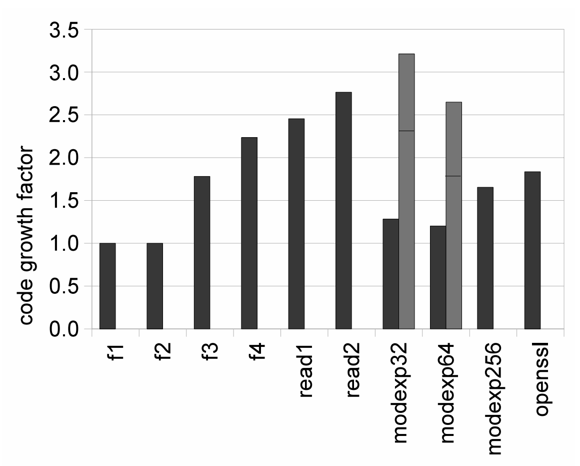

Fig. 3. Code size increase after applying if-conversion and elimination of variable latency division instructions

cryptographic functions occasionally, the influence on the total application execution time will be limited.

D. Code Size Overhead

The increase in code size due to if-conversion and elimination of divisions is depicted in Figure 3. As is to be expected, our control experiment, *f1*, has kept the same size, since the original function already had straight-line control flow. There is also no increase in code size for experiment *f2*, because the compiler middle-end had already applied if-conversion. The other experiments behave as expected: the more complicated the control flow, the greater the increase in code size after if-conversion. Similarly, the more memory operations there are, the more the code size increases. Without the elimination of variable-latency division instructions, the code becomes up to 2.75 times larger. When we also replace the division instructions by calls to a subroutine, the code sizes become up to 3.21 times larger.

To assess these results correctly, two observations need to be made. First, in real applications, these code size increases will only occur on those fragments that need to be protected because they involve key-dependent control flow. The remaining part of the code will remain the same, so the overal code size increase will be much more limited. Secondly, in the code sizes after division instruction elimination, the size of the subroutines that implement the division operations are included. Their relative contribution to the total code size is marked with the horizontal line in the bars in Figure 3: the fraction above the line comes from the newly included division subroutines. Their absolute sizes are 77 bytes for the 32-bit version, and 80 bytes for the 64-bit version. This clearly indicates that their contribution to the code size increase in real applications will be minimal.

V. Conclusion

This paper investigated the extent to which current x86 implementations support the so-called program counter security model in which all execution behavior needs to be independent of data values or data flow, and in which sofware removes all control-flow dependencies on secret keys.

We proposed a new compiler technique to remove key-dependent control flow from x86 programs. With this technique, compilers can defend against timing-based side-channel attacks on program properties that relate to control flow. The effectiveness and efficiency of the technique was evaluated, from which we can conclude that it is indeed feasible to remove key-dependent control flow, albeit with significant performance overhead.

We also discovered two micro-architectural features, being variable-latency division instructions and pessimistic load bypassing, that make the timing behavior of software depend on the data flow on Intel's most recent x86 implementation. These two cases can potentially be targeted by timing-based side-channel attacks. From this, we must conclude that the Intel Core 2 Duo implementation does not support the program counter security model, at least not for all programs.

For at least one of those micro-architectural features, namely the variable-latency division instruction, our experiments have shown that compilers can work around them effectively, albeit again at a significant performance overhead. For the data-dependent behavior resulting from pessimistic load bypassing, our research is ongoing in the directions indicated in this paper.

References

[1] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[2] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *CARDIS*, 1998, pp. 167–182.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO 99, LNCS 1666*. Springer-Verlag, 1999, pp. 388–397.

[4] O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-Cache," in *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*. Springer-Verlag, 2006, pp. 1–20.

[6] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 473–482.

[7] ——, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 494–505, 2007.

[8] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems CHES 2006, LNCS 4249*. Springer, 2006, pp. 201–215.

[9] E. Brickell, G. Graunke, M. Neve, and J. pierre Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities. iacr eprint archive, report 2006/052," 2006.

[10] D. J. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.

[11] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *ASIACCS '07*, 2007.

[12] G. Agosta, L. Breveglieri, G. Pelosi, and I. Koren, "Countermeasures against branch target buffer attacks," in *FDTC '07: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–79.

[13] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," pp. 156–168, 2005.

[14] ——, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[15] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.

[16] J. Coke, H. Balig, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles, "Improvements in the Intel Core 2 processor family architecture and microarchitecture," *Intel Technology Journal*, vol. 12, no. 03, pp. 179–192, 2008.

[17] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, Dec 2008.

[18] B. Schneier and D. Whiting, "Twofish on smart cards," in *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, 2000, pp. 265–276.

[19] "The ARM Cortex-A9 processors," http://www.arm.com.

[20] E. Quiñones, J.-M. Parcerisa, and A. Gonzalez, "Selective predicate prediction for out-of-order processors," in *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 46–54.

[21] S. Gueron, "Advanced encryption standard (AES) instructions set," Intel Mobility Group, Tech. Rep., 2008.

[22] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983.

[23] D. I. August, J. W. Sias, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. mei W. Hwu, "The program decision logic approach to predicated execution," in *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 208–219.

[24] M. Schlansker, S. Mahlke, and R. Johnson, "Control cpr: a branch height reduction optimization for epic architectures," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1999, pp. 155–168.

[25] F. Sano, M. Koike, S. Kawamura, and M. Shiba, "Performance evaluation of AES finalists on the high-end smart card," in *AES Candidate Conference*, 2000, pp. 82–93.

[26] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 1999, pp. 292–302.

[27] C. Clavier and M. Joye, "Universal exponentiation algorithm," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2001, pp. 300–308.

[28] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (AES)," in *FSE*, 2003.

[29] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[30] D. Dolz and G. Parra, "Using exception handling to build opaque predicates in intermediate code obfuscation techniques," *Journal of Computer Science & Technology*, vol. 8, no. 2, 2008.

[31] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2003-2380, Sep 2003.

APPENDIX
SAMPLE CODE FRAGMENTS

This appendix presents the sample code fragments that were used in Section IV to measure the performance overhead corresponding to the transformations of code fragments with varying degrees of control and data flow complexity.

A. f1

```
int __attribute__((annotate("balance")))
f1(int a, int b, int c, int d) {
  return a+b;
}
```

B. f2

```
int __attribute__((annotate("balance")))
f2(int a, int b, int c, int d) {
  if (a < b)
    return a+b;
  else
    return c+d;
}
```

C. f3

```
int __attribute__((annotate("balance")))
f3(int a, int b, int c, int d) {
  if (a < b) {
    if ( c < d )
      return c+d;
    else
      return c-d;
  } else {
    if (a > d)
      return a-d;
    else
      return a+d;
  }
}
```

D. f4

```
int __attribute__((annotate("balance")))
f4(int a, int b, int c, int d) {
  if (a < b) {
    if ( c < d ) {
      if ( a < 0 )
        return c+d;
      else
        return c-d;
      } else {
      if ( b < 0 )
        return b+c;
      else
        return a+b;
      }
    } else {
      if (a > d) {
        if ( d < 0 )
          return a-d;
            else
          return a+d;
      } else {
        if ( c < 0 )
          return c+a;
        else
          return c-a;
      }
    }
  }
```

E. memread1

```
int __attribute__((annotate("balance")))
memread1(int a, char* b) {
  if (a == 0) {
    return *b;
  } else {
    return a;
  }
}
```

F. memread2

```
int __attribute__((annotate("balance")))
memread2(int a, char* b) {
  if (a == 0) {
    return b[0] + b[1];
  } else {
    return a;
  }
}
```

G. OpenSSL fragment

```
for (i = min; i != 0; i--){
  t1= *(ap++);
  t2= *(bp++);
  if (carry) {
    carry=(t1 <= t2);
    t1=(t1-t2-1)&BN_MASK2;
  }
  else {
    carry=(t1 < t2);
    t1=(t1-t2)&BN_MASK2;
  }
  *(rp++)=t1&BN_MASK2;
}
```