Adaptive Compiler Strategies for Mitigating Timing Side Channel Attacks

Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere

Abstract—Existing compiler techniques can transform code to make its timing behavior independent of sensitive values to prevent information leakage through time side channels. Those techniques are hampered, however, by their static nature and dependence on details of the processor targeted during the compilation. This paper presents a dynamic compiler approach based on offline profiles and JIT compiler strategies. This approach reduces overhead significantly and enables a trade-off between provided protection and overhead. Furthermore, it supports adaptive policies in which the protection adapts to run-time changes in the requirements. A prototype implementation in the Jikes Research VM is evaluated on RSA encryption, HMAC key verification, and IDEA encryption.

Index Terms—JIT compiler, adaptive protection, time side channels, code transformations, efficiency, effectiveness, trade-off, profiling

1 Introduction

Cryptographic and other software operates on sensitive data on a regular basis. When implemented properly, its IO behavior provides strong guarantees against attacks on privacy, authentication, and other security requirements.

In practice, however, the IO behavior is not the only observable property. Depending on the kind of access (i.e., physical or networked) to devices, attackers can observe electromagnetic radiation, power and 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 software handling sensitive data.

Depending on the granularity at which an attacker can observe side-channel information, he can mount different types of attacks. Trace-driven attacks extract enough information to reconstruct whole execution traces of attacked algorithms [1], [2], [3], [4], [5], [6]. Access-driven attacks only reconstruct the access patterns to architectural components such as caches, branch predictions tables, and execution units [7], [8], [9], [10], [11], [12], [13], [14]. Time-based attacks solely observe execution time [15], [16], [17], [18], [19], [20], [21]. For many attacks, the dependence between secret data 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, for example by considering mutual information [22].

Mitigation strategies range from the hardware level [16], [23], [14], [24], over software [16], [25], [26], [27], [28], [29], [30], [31], [32], to the algorithmic level [33], [34], [21], [35], and combinations thereof [36].

In some approaches, static compilers transform the code such that the generated binary code no longer leaks information via time side channels by ensuring that the compiled code's execution time no longer depends on secret data [26], [29], [37]. For example, the code is transformed such that no conditional branches depend on values derived from secret data. Some techniques enable a trade-off between efficiency (i.e., the overhead introduced by a transformation) and effectiveness (i.e., the level of protection provided) by giving the developer the option to choose or parameterize alternative forms of a protection. For example, Van Cleemput et al. showed how the information leakage correlates with execution slowdown when injecting less or more mitigation code into an application to push variable-latency instructions off a program's critical paths [37].

However, the static nature of the existing compiler techniques comes with major drawbacks. For out-of-order architectures like Intel's x86 processors, the effectiveness and the efficiency of the transformations depend on the details of the execution pipeline. As a result, the optimal combination of transformations differs from one generation to another, even within processor families that offer exactly the same instruction set architecture. This is a serious issue at times when developers see their apps downloaded and installed on a multitude of different devices, when users upload their applications into the cloud onto machines of which they might not know the exact pipeline features, and when applications running on heterogeneous multiprocessors migrate from one type of core to another transparently [38].

Moreover, even if the developer and user know the exact processor on which their software runs, the run-time circumstances and protection requirements can still vary over time. For example, at any point in time (i) a device might be offline or online in networks that introduce varying amounts of jitter, which enables different types of attacks [39]; (ii) the application might be handling very sensitive or rather insensitive data; (iii) the application might be or not be sharing resources with other applications that can execute attacks based on resource contention [6], [40]. Current static compilers support only one protection level per generated binary. While static binaries can in theory include code compiled for several scenarios, the resulting code bloat impedes the generation of code for many different combinations of usage scenarios and target processors.

As an alternative, we propose just-in-time (JIT) compilation to generate leak-free code specifically for the processor

The authors thank the Agency for Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research - Flanders.

on which the software runs. Moreover, JIT compilers can limit the protective code transformations to the leaking code fragments actually being executed on sensitive data for the provided inputs. Finally, JIT compilers can adapt the level of protection to the observed run-time environment. With JIT compilation the deployed protection can be tuned for the actual usage scenario, thus minimizing the code bloat and performance overhead. By flushing the software cache and regenerating code, the protection level can even be adapted dynamically when the security requirements change.

To identify the minimal set of code fragments that should be considered for protection, we base the JIT protection on offline profiles. We rely on the developer to annotate the top methods to be protected and to run the software on multiple inputs to detect which fragments show diverging timing behavior that depends on sensitive data and that can hence leak information. By exercising the program on adequately chosen inputs, the developer can ensure that a minimal, but sufficient set of diverging behaviors are considered for minimizing the set of fragments to be transformed, and hence for minimizing the introduced run-time overhead.

We opt for a profile-based approach as opposed to pure static analysis because we are interested in protecting code that is actually executed. The dynamic information provided by a profile-based approach significantly improves the precision of the resulting analyses. Furthermore, no formalization of the relevant features of the inputs is required. We selected offline profiling over online profiling because online profiling suffers from the same imprecision problem as static analysis: For protecting against attacks in which the software is repeatedly executed on the same input, a purely online technique needs to make (overly) conservative observations about all possible other inputs.

Although our profile-based approach is also applicable in the context of static compilation, it would require the developer to have all the necessary inputs in his training set for profiling, which is not necessarily trivial. To compensate for when a developer overlooks certain relevant inputs, the JIT compiler injects monitors along the execution paths not covered by the developer's training input set. When such a path is triggered at run time, the existing code that was generated under the now disproven assumption that the path was not realizable, is flushed from the compiler's software cache and replaced by new code generated on the basis of the old profile information expanded with the new information collected by the monitor. This implies that an attacker can collect at most one time sample for each execution path not covered by the training phase. In many attack scenarios, in particular the vast majority of attacks on cryptographic software, collecting one sample does not suffice to mount an attack.

The main contributions of this paper are the presentation of the combined offline-online approach, and the evaluation of a prototype implementation in JikesRVM on a three cryptographic algorithms from real-life security libraries, with which we demonstrate that the performance overhead of the pre-existing state-of-the-art static compiler techniques can be reduced with 75%-82%. This prototype and all the raw experimental data will be open sourced after publication.

Section 2 presents the protective transformations our JIT compiler applies. Section 3 presents our approach for selecting fragments to transform and transformations to apply. Section 4 present an experimental evaluation, Section 5 discusses related work, and Section 6 draws conclusions.

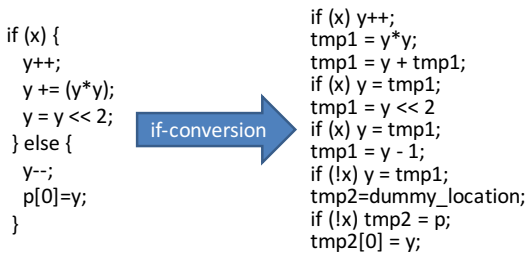

Fig. 1: If-conversion by means of conditional moves.

2 Mitigating Transformations

Existing compiler techniques have proven effective to protect small code fragments [26], [37]. Here, we describe and extend the existing transformations to cover more complex code fragments typically found in large software libraries.

2.1 Control Flow Transformations

Execution time variation can be caused by control and data flow being dependent on security-sensitive values in an application. Control flow has, in many cases, the biggest impact on execution time variation [26].

If-conversion [41] has been used successfully to mitigate timing attacks by transforming control flow dependencies into data flow dependencies [26]. On architectures that support full predication, if-conversion deletes branches around blocks of instructions and replaces them with predicated instructions. Figure 1 demonstrates the basic transformation on architectures that lack full predication, but that support conditional move instructions or select instructions, such as the Intel x86 architecture [42] and the ARMv8 architecture [43]. Such instructions allow the encoding of all statements starting with `if (...)` on the right of Figure 1 in a single instruction, of which the latency is typically independent of the values of the source operands, including the condition.

As an alternative to predication, masking operations have been proposed [29]. Masking cannot handle all operations, however, and also with if-conversion, some instructions require special care. This includes instructions with side-effects like loads, stores, and exception-throwing instructions such as divisions [26].

Special care is needed for converting procedure calls. A call instruction contributes not only its own execution time, but also the execution time of the callee's body and possibly the bodies of callees further down the call chain. To make the call's execution time constant, the call instruction is not executed conditionally; only the code inside the callee is executed conditionally under the control of an additional predicate parameter. Figure 2 demonstrates this: The conditional call to `f1` on the left is replaced by an unconditional call to `f2`, the if-converted replacement of `f1` in which the code is adapted to only update the global program state when the original call would have been executed.

Throughout this paper we use the term *predicate* of a conditionally executed instruction, basic block or method in

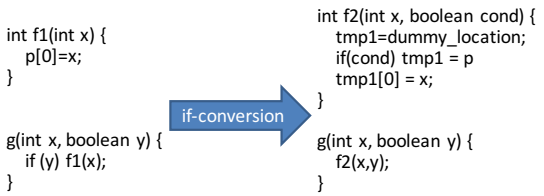

Fig. 2: If-conversion of conditional function calls.

if-converted code to denote the logical value that indicates whether or not that instruction, basic block or method was to be executed at some point in time in the original program.

2.2 Data Flow Transformations

Even if the control flow does not depend on a security-sensitive value, data flow dependencies can still cause execution time variation. These variations are often the result of performance optimizations in the underlying hardware.

Variable-latency arithmetic instructions occupy execution units in modern processor pipelines for a varying amount of cycles, depending on the values of their input operands. Examples of such instructions that have proven susceptible to side channel attacks [40], [26] are divisions and multiplications implemented by means of early-exit algorithms to optimize the use of their scarce resources. Several existing compiler techniques can solve this.

First, variable-latency instructions can be replaced by calls to fixed-latency emulation routines that do not contain variable-latency instructions [26], [35]. This completely eliminates data-dependent timing variation for those instructions. It comes with a high execution time overhead, but offers the advantage that the emulation routines typically can be reused for whole processor families, within which the set of variable-latency instructions typically does not vary.

Alternatively, complex code fragments replace single variable-latency instructions such that a fixed latency is enforced on all variable-latency instructions by manipulating their operands [37]. This achieves data-independent timing behavior at a much lower overhead. However, in order to apply it successfully, it is necessary to know the exact relation between input values and latencies of the variable-latency instructions, which are known to change even within processor families [37]. In static compilers, this technique therefore suffers from limited portability. In our JIT approach, there is of course no portability issue, as the exact processor is known at compile time.

A third technique injects chains of fixed-latency operations into the code's data dependence graphs to push the variable-latency operations off the critical paths. This technique can reduce timing variations to minimal levels, but it cannot eliminate them completely [26], because of resource contention between instructions on and off the critical paths. Still, this technique enables a trade-off of run-time overhead vs. achieved protection: Longer/shorter injected critical paths result in a more/less reduced timing variation at higher/lower overhead [37]. The precise level of remaining variation for a specific injected critical path cannot (easily) be predicted, however.

Data dependencies through memory can also introduce time variation, e.g., when the cache access pattern depends on the sensitive data [44]. This impact on execution time is amplified by the way if-conversion handles memory accesses in literature [26], [37]: Memory operations that are predicated false but are executed unconditionally in the if-converted code are fed a dummy operand value to access a single dummy location. See the store through p in Figure 1. This avoids unwanted change to the program state and page faults, but the memory access pattern then strongly depends on the predicate, and hence its timing can leak information.

Although it is in many cases impossible to avoid these cache-related timing variations entirely, we have observed that it is quite possible to significantly reduce them with the following two adaptations. First, instead of one single, global dummy location, provide a different dummy memory location for each if-converted memory instruction. While this will increase memory usage, it will avoid repeated accesses to the same memory location by memory instructions that were not executed in the original program.

Secondly, allocate dummy arrays instead of single memory locations when performing array stores and loads, and index them using the index that would have been used if the instruction was predicated true, modulo the dummy array size. For example, if the original instruction would read from an array in a manifest loop, the protected instruction would read from the dummy array using a similar access pattern when predicated false, instead of reading from the same memory location each iteration. The program semantics are not changed, and the cache pattern of code predicated false resembles that of the code predicated true.

This transformation can still be refined, e.g., by using alias analysis or profiling to let operations that likely accesses the same locations in memory when predicated true also access the same dummy arrays when predicated false. Most of the improvement in variable execution time can already be absorbed by the simple scheme, however.

A second, less obvious cause of data-dependent timing variation is the load-store forwarding and load bypassing logic in out-of-order processor pipelines. Current state-of-the-art techniques minimize timing variation by increasing the distance between load and store instructions that may alias, for example by inserting no-op instructions, such that potential candidates are no longer together in the processor's load-store forwarding/bypassing instruction window. With enough no-ops, the variation can even be eliminated completely [37]. The minimal number of no-ops needed to achieve a certain level of protection cannot easily be predicted, however, due to undocumented pipeline behavior. So in essence, the no-op insertion is a best effort mitigation.

So while in theory it is still possible to craft specific input keys to force measurable execution time difference due to cache behavior and memory pipeline behavior, in practice we found that applying the above transformations in a JIT compiler environment can effectively protect real applications using the methods described in the next section.

3 Profile-Based JIT Protection

This section discusses how profiles can be used to extend the protection techniques described in the previous section and to reduce their overhead by automating and optimizing the selection of protected code regions.

3.1 Profile collection

The developer collects a separate application profile for each input in a training input set $\mathbb{I}$. Each profile contains edge counts, i.e., the number of times each edge in the program's call graph and its procedures' CFGs are followed during the program's execution. The edge counts and the instruction counts derived from them are collected by running an instrumented program version. In addition, this version contains counters to count the number of iterations in each loop and to measure recursion depths.

Ideally, the training inputs cover all extreme timing behaviors dependent on secret data. This implies they at least trigger the execution of all code fragments of which the execution depends on secret data. Developers can rely on their test inputs, real-life workloads, and more advanced tools such as fuzzers (relying, e.g., on concolic execution) to collect a large enough set of inputs. As we will discuss in Section 3.3, we require the developer to tag so-called root methods, i.e., methods whose timing behavior (including that of their callees) should be made independent of data values. This tagging at once marks the code region in which a fuzzer should try to increase the coverage.

In practice, we found that cryptographic code typically requires only a small set of training inputs to provide enough coverage to successfully protect cryptography applications against timing attacks.

As we already mentioned in the introduction, it is not strictly necessary for the developer to cover all extreme timing behaviors or secret-data-dependent execution paths. Section 3.6 discusses this in more detail.

3.2 Transforming Loops and Recursive Calls

Loops with variable iteration counts require extra attention. Obviously the number of executed iterations has an impact on the execution time of a program. Yet the conditional branches that control loop exits cannot be if-converted. Coppens et al. already suggested a possible solution based on introducing an extra loop counter with a fixed, manually selected iteration count [26]. If-conversion is then applied on the loop body such that only the iterations that need to be executed contribute to the global program state.

Our first adaptation to this scheme is minor: For each loop, we automatically extract the largest iteration count observed in the collected profiles. This automatic approach facilitates the protection of legacy applications or third-party libraries of which a developer has little knowledge.

Loops by means of recursive calls are handled similarly. In that case, we add a counter parameter to the function to pass the run-time recursion depth, and the recursion is executed (with the body if-converted) until the maximal depth that was observed during the profiling.

3.3 Automatic Detection of Code Regions to Transform

Existing compiler techniques to transform code to obtain constant execution time require that developers tag one or more methods as sensitive, implying that those handle sensitive information. During compilation those methods and all of their direct and indirect callees are then transformed with the protections discussed in Section 2, resulting in a secure application. If some callees should not be transformed, the developer has to tag those explicitly.

```
public byte[] encrypt(byte[] plaintext, Key k, int type){
   switch(type){
      case 1:  return encrypt_RSA(plaintext,k)
      case 2:  return encrypt_DSA(plaintext,k)
      case 3:  return encrypt_ECC(plaintext,k)
      default: throw new InvalidKeyException();
    }
}
```

Listing 1: Example method to be protected

That manual tagging is susceptible to human error. The developer might overestimate or underestimate the procedures that need to be transformed, as it may not be clear which ones actually leak. In case he underestimates them, applications might still leak information. If he overestimates them, it results in unnecessarily high overhead.

In addition, applying if-conversion naively on all direct and indirect callees of a procedure that needs to be protected can cause a large, and often unnecessary overhead. This is especially true for libraries with multiple levels of indirection and large call trees. Consider for example the library method in Listing 1. A switch statement selects one of several encryption algorithms. If we would blindly deploy if-conversion on this code fragment and its callees, a protected version of each algorithm would be executed. However, if this method is always passed the same options in some main program, such that the same encryption routine is always invoked, much of this overhead is completely unnecessary. The shown method can then remain untransformed.

This problem of selecting the methods to protect and the extent to which they need to be if-converted is addressed by using profiles to automate a large part of the selection of code fragments to transform. In our approach, the developer still manually needs to tag the root methods to protect, i.e., the methods of which the execution time, incl. that of its callees, should become independent of secret data. Importantly, the developer does not need to select or tag which (direct or indirect) callees actually need to be transformed. That part will be automated. In other words, our approach only requires the developer to specify the security requirement on a root method, not which code needs to be transformed to meet that requirement. (In fact, it might even be that the root method itself need not be transformed at all.) Our approach is therefore much less error prone.

The automated selection of code fragments to transform then happens at two levels. First we use call edge counts and instruction counts to determine the set of methods $\mathbb{S}$ to be transformed. Secondly, we optimize how if-conversion is applied within these methods based on branch profiles.

On the basis of the set $\mathbb{R}$ of root methods, we iteratively build the set $\mathbb{S}$ of methods to be transformed, starting with an empty set. Each iteration adds methods to $\mathbb{S}$, thus increasing the protection level. The end result is a fully protected application in which only a reduced subset of methods are transformed. Each intermediate set $\mathbb{S}$ can be considered as a partially protected application suitable in scenarios with less strict requirements, e.g., because attacks can only occur over a network that introduces jitter on the measurements [39].

1 $\mathbb{R} = \{m \in \mathbb{M} \mid m$ is tagged as a root method $\}$
2 $\mathbb{C} = \{m \in \mathbb{M} \mid m$ is an (in)direct callee of a method in $\mathbb{R}\}$
3 $\mathbb{E}_{equal} = \{e \in \mathbb{E} \mid \forall (i,j) \in \mathbb{I}^2 : \mathrm{Ec}_i(e) = \mathrm{Ec}_j(e)\}$
4 $\mathbb{S} \leftarrow \emptyset$
5 $n \leftarrow 1$
6 **do**
7 $\quad \mathbb{A} \leftarrow \emptyset$
8 $\quad$ **foreach** $m \in \mathbb{C} \setminus \mathbb{S}$ **do**
9 $\quad\quad$ **if** $(\forall e \in In(m) : e \in \mathbb{E}_{equal})$
10 $\quad\quad \wedge\ (\exists e \in Out(m) : e \notin \mathbb{E}_{equal})$ **then**
11 $\quad\quad\quad \mathbb{A} \leftarrow \mathbb{A} \cup \{m\}$
12 $\quad\quad\quad \mathbb{E}_{equal} \leftarrow \mathbb{E}_{equal} \cup \mathrm{Out}(m)$
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 $\quad \mathbb{S} \leftarrow \mathbb{S} \cup \mathbb{A}$
16 $\quad \mathbb{S}e_n \leftarrow \mathbb{S}$
17 $\quad n \leftarrow n + 1$
18 **while** $\mathbb{A} \neq \emptyset$;

Algorithm 1: Method selection based on call edge counts.

3.3.1 Phase 1: Call Edge Heuristic

In a first phase, we build on the simple heuristic that in the sensitive code, all method calls need to be executed exactly the same number of times, independently of the input data. The rationale is that if this requirement is not met, the execution time will likely vary too much to provide any useful level of protection.

Let $\mathbb{R}$ be the set of root methods that need to be secured, $\mathbb{M}$ the set of all application methods, $\mathbb{C}$ the subset of $\mathbb{M}$ containing direct and indirect callees of the root methods in $\mathbb{R}$, $\mathbb{S}$ the complete set of methods that need to be secured, $\mathbb{S}e_n$ the set of methods that need to be secured at iteration n in the algorithm, $\mathbb{I}$ the set of input values of the application, and $\mathbb{E}$ the set of call edges between methods. Let $\mathbb{E}_{equal} \subseteq \mathbb{E}$ initially be the set of call edges with equal edge counts for all inputs $\mathbb{I}$. Define $\mathrm{Ec} : \mathbb{I} \times \mathbb{E} \mapsto \mathbb{N}$, with $Ec_i(e)$ the function that returns the execution count of edge e under input i. Define $\mathrm{In} : \mathbb{M} \mapsto \mathcal{P}(\mathbb{E})$, with $\mathrm{In}(m)$ the set of incoming call edges of method m. Define $\mathrm{Out} : \mathbb{M} \mapsto \mathcal{P}(\mathbb{E})$, with $\mathrm{Out}(m)$ the set of outgoing call edges of method m.

Algorithm 1 starts by building a set of methods $\mathbb{C}$ of direct and indirect callees of the root methods $\mathbb{R}$. These root methods are tagged by either a developer or library user to be secured against timing attacks. They are used to determine the scope of candidate methods to protect.

Each iteration over lines 6–18 adds methods to the secured set $\mathbb{S}$ if they meet the following conditions: on the following conditions: (i) The method has equal edge count for all of its incoming edges, and (ii) the method has at least one outgoing edge that has unequal edge counts.

Because all methods in $\mathbb{S}$ will eventually be protected, their outgoing edge counts will become equal for all input values in $\mathbb{I}$ in the protected program. The algorithm already simulates this by adding the outgoing edges of each selected method to $\mathbb{E}_{equal}$ on line 11. After the algorithm is finished, the following statement holds, indicating that in the protected application all methods from set $\mathbb{C}$ will have equal outgoing edge counts: $\forall m \in \mathbb{C} \mid \forall e \in \mathrm{Out}(m) : e \in \mathbb{E}_{equal}$.

Figure 3 illustrates the first three iterations of this process on a fictitious call graph containing all methods in $\mathbb{C}$. For each iteration, dark gray squares represent methods in the set of secured methods $\mathbb{S}$ before the start of that iteration.

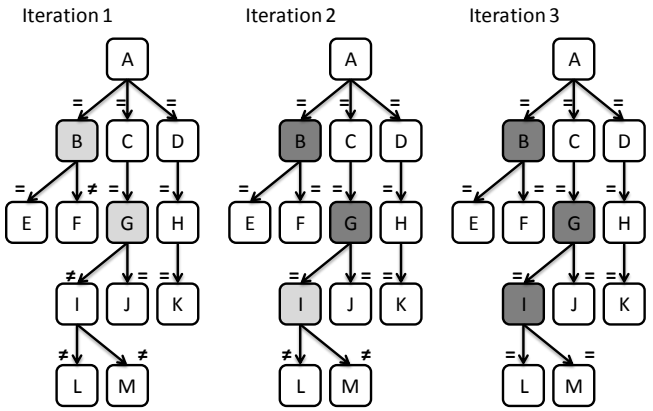

Fig. 3: First three iterations of call-edge-based code selection.

1 **foreach** $m \in \mathbb{C} \setminus \mathbb{S}$ **do**
2 $\quad$ **if** $\exists t \in \mathrm{Inst}(m)\ .\ \exists (i,j) \in \mathbb{I}^2 : \mathrm{Ic}_i(t) \neq \mathrm{Ic}_j(t)$ **then**
3 $\quad\quad \mathbb{S} = \mathbb{S} \cup \{m\}$
4 $\quad$ **end**
5 **end**
6 $\mathbb{S}i = \mathbb{S}$

Algorithm 2: Method selection based on execution counts.

Light gray squares represent methods in $\mathbb{A}$ added to $\mathbb{S}$ in the current iteration. Edges in $\mathbb{E}_{equal}$ are marked with an $=$, all other edges with $\neq$. In the first iteration methods B and G are the only methods that meet the requirements to be protected. By adding them to $\mathbb{S}$ and their outgoing edges to $\mathbb{E}_{equal}$ method I can be added in iteration 2. In iteration 3 all edges have been added to $\mathbb{E}_{equal}$ and the algorithm finishes, with $\mathbb{S} = \{B, G, I\}$.

In each iteration n, the set of methods that have been selected for protection up to that point is saved in a separate set $\mathbb{S}e_n$ (line 16). This enables the (potentially dynamic) selection of one of these intermediate results when complete protection is not required at some point during the actual deployment of the application or library.

3.3.2 Phase 2: Instruction Count Heuristic

The set of methods selected based on call edge counts in phase 1 is only a coarse-grained estimation of the methods that might leak information. Its intermediate and final results will not provide meaningful protection in all scenarios. So in a second phase, we use instruction execution counts to identify those locations where intraprocedural control flow can contribute to leaks. Let $\mathbb{T}$ be the set of application instructions; $\mathrm{Inst} : \mathbb{M} \mapsto \mathcal{P}(\mathbb{T})$, with $\mathrm{Inst}(m)$ the function that returns the set of instructions in method m; $\mathrm{Ic} : \mathbb{I} \times \mathbb{T} \mapsto \mathbb{N}$, with $Ic_i(t)$ the function that returns the execution count of instruction t under input i.

Algorithm 2 iterates over all methods in $\mathbb{C}$ that were not yet added to $\mathbb{S}$ in phase 1. If a method has a different instruction count for at least one of its instructions for two or more of the inputs in $\mathbb{I}$, it has input-dependent control flow, and is hence added to $\mathbb{S}$ on line 3.

Because the total execution time difference caused by methods tagged in this phase is not influenced by their callees (unless they are added to $\mathbb{S}$ themselves during this phase), protecting methods tagged in this phase will on

average result in smaller reductions of execution time difference compared to methods tagged in phase 1. The overhead of protecting these additionally added methods is also more limited, however, because tagging them never triggers the additional tagging of more methods down the call chain.

The end result of this phase is a set of methods $\mathbb{S}_i$. If this set is protected during execution, the application will no longer feature input-dependent control flow. So at that point we effectively protect against both instruction cache attacks and branch prediction attacks [1], [7], [4], [3].

In practice we observed that applying all control flow and data flow transformations to this set of methods suffices to eliminate most of execution time variation.

3.3.3 *Input-Dependent Data Flow*

The algorithms described above use input-dependent control flow variation as an indicator for execution time variation. This allows fast detection of the majority of code regions to protect based on a single profile run for each input in $\mathbb{I}$. However, it does not detect methods that only leak timing information through data flow features such as variable latency instructions, memory access delays and interactions between instructions in the pipeline.

In our experiments on implementations of cryptographic algorithms, we did not encounter any additional methods leaking timing information on top of those already selected based on control flow variation in Phases 1 & 2. In case there would be such methods, however, it is also possible to detect them using profile information, albeit with a much slower process. Concretely, sample-based execution time profiling during many runs of the fully if-converted methods can detect statistically relevant differences between methods' execution times for different inputs. It is worth noting here, that some of the most frequently used methods for collecting per method timing information in other contexts, such as code optimization, are not useful in this context. In particular, the collection of time stamp counters by means of injected RDTSC instruction and the necessary bookkeeping code is not useful, as the insertion of this instruction affects the pipeline behavior too much to measure the effect of variable-latency instructions in non-instrumented code.

3.4 Selective If-Conversion

Current state-of-the-art techniques as described in Section 2 if-convert whole procedures. This includes conditional branches that are taken in one direction or switches of which only a few cases are actually executed. Unexecuted paths or code fragments in executed functions occur for several reasons. One is library functionality that is not executed in a specific program, as already discussed in Section 3.3. A second reason involves code that is present because of coding practices and guidelines but that is never executed, such as superfluous default cases in switch statements. A third reason involves checks that always evaluate to true or always evaluate to false under "normal" circumstances, including all scenarios that might leak information through side channels. This includes checks for detecting out-of-memory issues or for the presence of files to be opened. Using branch profile information, we can select the branches that actually exhibit input-dependent behavior and only convert those to reduce the overhead.

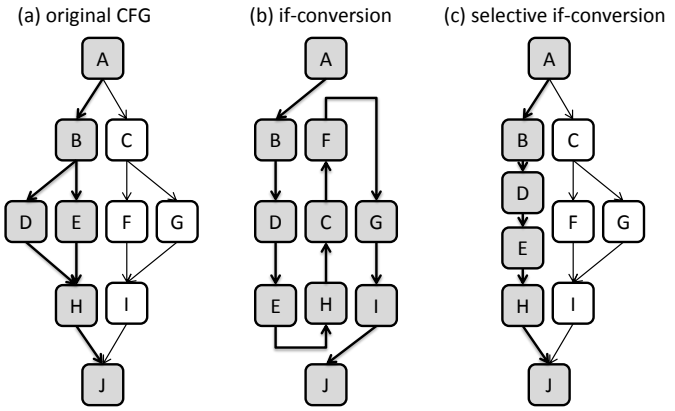

Fig. 4: Example of complete and selective if-conversion.

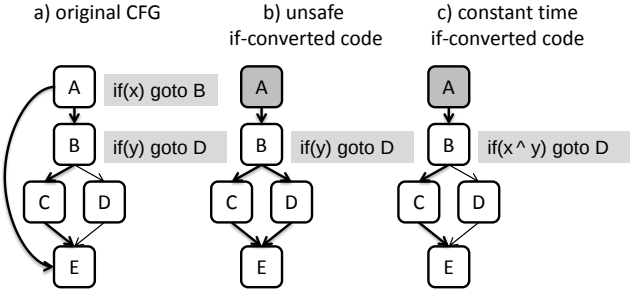

Fig. 5: Example of correct selective if-conversion.

Figure 4a illustrates the CFG of an unprotected application. Gray squares and bold arrows represent basic blocks and control flow edges resp. that are executed at least once for at least one of the program inputs. In this simple case, it is clear that only the conditional branch in basic block B can cause input-dependent control flow. Compared to the whole procedure being if-converted as shown in Figure 4b, the selective if-conversion result shown in Figure 4c will yield much less performance overhead.

In our current implementation, branches are left unprotected only if they are profiled as always going in the same direction for all inputs in $\mathbb{I}$. Although branches with identical branch history for all inputs (e.g. always taken in the first half of the whole execution, and always not-taken in the second half) do not leak timing information, enforcing the same branch sequence in the protected application is not trivial and currently not supported.

Selective if-conversion has to be performed with great care, because it can actually introduce control flow variation in the unconverted branches where no such variation was present in the original program. Figure 5a shows a simple example CFG, with only two executed paths during profiling: $A \rightarrow E$ and $A \rightarrow B \rightarrow C \rightarrow E$. Assuming the value of y is always evaluated to $false$ in this particular example, basic block D will never be executed.

When the branch in basic block A is if-converted as shown in Figure 5b, block B is always executed. If block B is predicated false (when the jump in A was taken) the value of y might be undefined or evaluate differently than profiled, allowing the jump in block B to go in either direction.

To avoid this, the jump condition of unconverted branches needs to be rewritten as follows: Let $original$

be the original jump condition of the branch, $predicate$ a boolean value calculated at run time indicating if the basic block containing the branch instruction was executed in the original program. The new jump condition new for unconverted branches that are profiled as always taken then becomes $new = \neg predicate \vee original$. For never-taken branches, it becomes $new = predicate \wedge original$. This ensures the new jump direction defaults to the profiled branch direction when the instruction is predicated false and jumps in the same direction as the original branch condition when predicated true. Figure 5c shows the CFG with the branch condition of the branch in basic block B rewritten according to the rules above. After if-converting the branch in basic block A and rewriting the jump condition in basic block B, the same path $A \rightarrow B \rightarrow C \rightarrow E$ will be taken as long as not both x and y evaluate to true at the same time, i.e., as long as they behave as during the profiling. Section 3.6 discusses how we handle a situation in which x and y would diverge from their behavior during profiling.

3.5 Security/Performance Trade-off

As mentioned in the introduction, the required level of protection depends on the usage scenario. For example, in Section 4 we evaluate three cryptographic use cases, for which the leakage of timing information has widely varying impacts on the search spaces of attackers searching for secret keys. Moreover, the amount of useful timing information leaked to a (remote) attacker will also be influenced by the network setup, timing source, and whether or not an attacker has control over the algorithms' inputs [39], [18].

In some scenarios, reducing the timing variation instead of eliminating it might therefore suffice. As a reduction typically requires less transformations than a full elimination, the performance overhead will also be smaller. It is hence useful to support a security/performance trade-off. A reduction instead of elimination of timing variation can be achieved by (i) protecting only a subset of methods $\mathbb{S}' \subseteq \mathbb{S}$, and (ii) by applying only a subset of protection techniques $\mathbb{P}' \subseteq \mathbb{P}$. Each subset $\mathbb{S}'$ of $\mathbb{S}$ combined with a set of mitigation strategies $\mathbb{P}'$ of $\mathbb{P}$ used to protect these methods can be seen as design point in a security/performance trade-off.

While we cannot yet present a good (i.e., user-friendly, transparent, and reliable) method to support making this trade-off in practice, we analyze our approach's potential for to this trade-off for one evaluation use case in Section 4.3.2.

3.6 Execution Paths Not Covered by Profiling Inputs

Our mitigation strategies are designed to correctly handle all inputs, i.e., to conserve semantic behavior of the application even when execution paths occur at run time that were not observed during profiling. In the partially if-converted CFG on the right of Figure 5, the path $A \rightarrow B \rightarrow D \rightarrow E$ will still be executed whenever x and y evaluate to true. Likewise, converted loops and recursive call chains are still allowed to complete all their iterations resp. calls, even if their number exceeds the upper bounds observed during profiling.

When a new execution path is triggered, invariably some timing information leaks. The amount of leaked information is easily minimized, however. On the paths that were not triggered during the profiling (incl. loops with higher number of iterations than initially observed), but that are still present in the transformed, protected code, we inject monitors. If those get triggered, two actions are initiated. First, the stored profile information is updated persistently to reflect that some previously untaken path from now on has to be considered taken or that some observed upper bound has to be increased. Secondly, the involved methods are flushed from the JIT compiler's software cache and recompiled taking into account the new information.

If the initial profiling by the developer was not incomplete but not entirely inadequate, these monitors have a minimal impact on the performance of the protected code: After being triggered once, they either disappear entirely from the recompiled CFGs, or they are replaced by other monitors further down in the CFGs to monitor subpaths of the originally excluded paths. Those new monitors can be replaced iteratively, but if the original profiling was any good, very few iterations can occur. So very few monitors will ever be triggered in practice, and being located off the normally executed paths in the CFGs, they don't hurt regular performance significantly. In this regard, we should note that after a monitor is triggered, the profile information is updated persistently so future runs of the program automatically incorporate all behaviors of previous runs.

The same reasoning applies to the amount of leaked information. Most timing attacks require an attacker to collect a significant number of timing samples. While triggered monitors will result in a limited number of samples with relevant information for the attacker, he will then not be able to collect enough samples to let an attack succeed.

In exceptional cases, timing attacks might require very few samples to obtain useful information. One well-known case involves string matching: in case a pre-check first compares the lengths of the strings, not covering both outcomes of that pre-check during the profiling will result in a protected program that can still leak the length of a secret string. In cases like this, the developer can rely on fuzzing and code coverage tools as already mentioned in Section 3.1. In cryptographic code, the difference between different secret values is typically observed in how often certain code paths are triggered and in how the data flow impacts the execution time, not in which code paths are triggered. We hence conclude that in most scenarios, choosing sufficient profiling inputs to obtain strong protection is feasible. This includes the three use cases we evaluate in Section 4.

Finally, we want to point out that the insertion of run-time monitors and the run-time recompilation of code implies that our approach cannot be used as is with static compilers. Only in case a developer is confident that his profile inputs do cover all necessary behaviors, and he knows the exact platform and protection requirements, a static compiler can be used instead of a JIT compiler.

4 Experimental Evaluation

We evaluated a prototype implementation of our approach on three real-world cases known to leak timing information.

4.1 Prototype Implementation

For the run-time JIT compilation phase, we use the Jikes Research Virtual Machine [45]. Our secure JIT compiler

makes use of the JikesRVM Adaptive Optimization System (AOS) and its optimizing compiler [46] to (re)compile code for different security scenarios. It never interprets Java bytecode. Instead all bytecode is compiled to ensure that our protections are applied on all code that is ever invoked.

To support our mitigation strategies, we implemented four main modifications to JikesRVM infrastructure: the compilation strategy database, a customized classloader, a patched dynamic linker, and several extra compiler passes.

To minimize the overhead of the protections, it is important to realize that many of the sensitive code fragments invoked from within the sensitive root methods are often also invoked from within non-sensitive regions. To avoid the protection overhead when executing the latter regions, and to avoid that we need to recompile code when transferring control between sensitive to insensitive regions, we adapted the VM to support two versions of all methods: a secured version and an unsecured version.

4.2 Methodology & Experimental Setup

To mimic the strongest possible attacker that can observe many controlled executions in a low-noise setup, we ran all time measurements locally on an otherwise unloaded system. This avoids noise from network delay jitter. We also disabled address-space layout randomization and frequency scaling, and forced JikesRVM (v3.1.2) to use only one CPU and to pin the Java processes to that CPU.

We then fed the appropriate options to JikesRVM to make it bulk compile all methods at highest optimization level, and to disable recompilations not related to changing security levels. This way, all timing measurements are performed in the so-called steady state execution of the software. This avoids that large variations in execution time due to the (non-deterministic) invocation of compilers mask the statistically relevant variations caused by the data-dependent behavior that an attacker wants to exploit. To reduce noise introduced by garbage collection we increased the initial heap size to 2GB.

To measure execution times without intrusion in the evaluated code's pipeline behavior, we put x86 time stamp counter instructions RDTSC preceded by CPUID (for instruction serialization) in a harness around the code.

Special care was taken for the measurement of execution times on which statistical tests are performed. Such measurements of the same code version on multiple secret key values were always conducted in a single invocation of the VM, with a harness that alternates between different key values. This eliminates (non-deterministic) timing differences between different VM invocations. Our experiments show that execution times of the same experiment with the same key can differ up to 0.23% (almost 3ms for the RSA encryption) between VM invocations, which is much higher than the noise in our individual test cases. This also accounts for intermittent variations due to potential external influences such as CPU temperature or garbage collection.

To assess the impact of processor implementation details on the opportunities for optimizing the overhead of the available protection, and to assess the benefits of our adaptive, training-based approach that can tune code for specific architectures, we performed experiments on two different generations of Intel processors: ® Core™ 2 Duo (CPU E8400) and Intel® Core™ i7 (CPU 870). They feature different pipeline designs, with, amongst others, different load/store forwarding logic and different early-exit algorithms for division instructions. The Core™ 2 Duo features 6 early-exit points, and hence 6 possible latencies, whereas the Core™ i7 features 7 points and latencies. Despite these differences, the input combinations used for training and evaluation gave extreme timing behavior on both processors. The reason is of course that control flow dependencies have a much bigger impact than data-flow-related variable latencies.

iteration	0	1	2	3	4	5	6
nr. of calls	0	8	20	35	41	44	45
nr. of methods	1	7	15	25	30	31	32

TABLE 1: Selected code regions based on call edge profiles.

4.3 BouncyCastle RSA Encryption

First, we evaluated our approach on the RSA encryption algorithm in a recent version of the BouncyCastle cryptographic library (v1.52). The RSA algorithm can be used to sign messages with a private key. Any key-dependent timing variation therefore possibly leaks sensitive information. By default the encryption algorithm uses blinding. This common countermeasure against timing attack reduces the amount of information leaked, but the complete information is still eventually revealed to the attacker [47]. For its mathematical operations BouncyCastle relies heavily on core Java libraries, for which we used the GNU Classpath(V0.97.2) implementation that is also used to build JikesRVM. To ensure security of the encryption algorithm, both the algorithm and the underlying libraries need to be protected.

This implementation of RSA encryption features relatively deep call trees. Its control flow is highly dependent on the input data consisting of the secret keys and plaintext input. Excluding exception handling calls and native method calls, the call chain from the root doFinal(byte[]) method, which starts the encryption process, contains 104 methods and 270 calls between those methods. This is much larger and complex than the simple functions previously reported for static compiler techniques [26], [29], [37].

4.3.1 *Profile-Based Detection of Code Regions*

We used an RSA key generator to generate hundreds of 1024 bit RSA keys. From this set of keys we selected the two keys with the most variation in execution time, which we then used as the input set $\mathbb{I}$ for our profile-based code selection. We then profiled the algorithm for each key in $\mathbb{I}$ using identical sets of 64-bit pseudo-random plaintext messages.

Table 1 shows the results of applying Algorithm 1 to the sample application. The algorithm needs six iterations to collect all methods and calls to be transformed. After the final iteration, $45/270 = 16.7\%$ of all calls have been marked to be replaced by calls to a protected version their callees, for which $32/104 = 30.8\%$ of the methods need to be protected. In this experiment, Algorithm 1 sufficed. Algorithm 2 did not result in additional methods being selected.

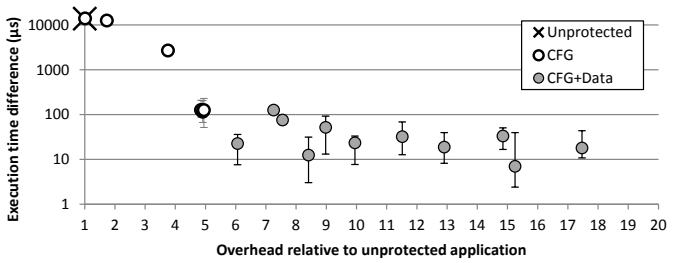

Fig. 6: Intel Core™ 2 Duo security/performance trade-off.

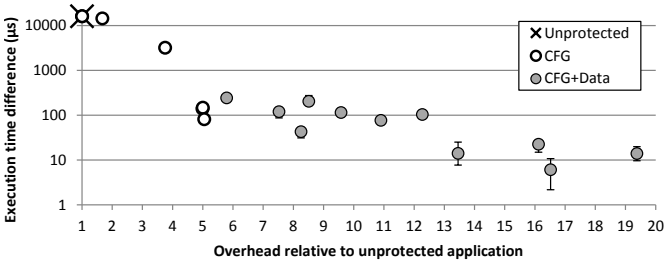

Fig. 7: Intel Core™ i7 security/performance trade-off.

	Compilation Strategy				Intel Core 2									Intel Core i7								
	S	array	div	nop	OH	Sample Size 20	50	100	200	300	500	700	990	OH	Sample Size 20	50	100	200	300	500	700	990
1	1	0	0	0	1.00									1.00								
2	7	0	0	0	1.01									1.00								
3	15	0	0	0	1.73									1.67								
4	25	0	0	0	3.76									3.75								
5	30	0	0	0	4.84	11	6	4	1					4.99	1							
6	31	0	0	0	4.91	11	9	9	8	5	1	2	1	5.00	1							
7	32	0	0	0	4.95	12	10	8	6	3				5.05	10	9	6					
8	32	64	0	0	6.06	12	12	12	12	12	12	6	4	5.79								
9	32	0	1	0	7.25	12	9	5	1					7.53	9	8	2					
10	32	0	0	6	7.55	10	2							8.26	1	2						
11	32	64	1	0	8.40	12	10	12	12	12	11	12	12	8.51								
12	32	64	0	6	8.98	12	10	7	1					9.58								
13	32	0	1	6	9.94	12	11	7	6	3	2	2	2	10.90								
14	32	64	1	6	11.52	12	10	6	1					12.27								
15	32	0	0	12	12.90	12	12	12	12	10	9	11	10	13.46	10	9	11	9	9	9	7	7
16	32	64	0	12	14.85	9	9	9	6	6	1	1	1	16.52	12	12	12	12	12	11	12	11
17	32	0	1	12	15.24	10	12	12	12	10	9	12	12	16.12	11	9	10	5	4	3	7	1
18	32	64	1	12	17.46	11	10	11	12	11	11	11	11	19.38	12	12	12	12	11	10	9	8

TABLE 2: Test results for different compilation strategies.

4.3.2 Run-time Overhead vs. Execution Time Variation

The protection's run-time overhead depends heavily on the desired level of protection. To illustrate this we protected the encryption algorithm with 18 different combinations of protection techniques, i.e., 18 design points specifically chosen to represent the broad scope of all possible combinations. For each of those compilation points we measured the average execution time difference between the two input keys based on 1000 encryptions of a fixed 64-bit plaintext message. The *Compilation Strategy* columns in Table 2 describe the 18 combinations of mitigation techniques. *S* indicates the number of methods in the set $\mathbb{S}$ as selected with Algorithm 1. These numbers correspond to those in Table 1. The column *array* gives the size of the dummy array objects; *div* indicates whether or not the division instructions are replaced by constant time implementations; *nop* indicates how many nop instructions are inserted in between memory instructions. The column marked *OH* (overhead) shows protected execution times relative to the execution time of the slowest key in the unprotected application.

Figures 6 and 7 show the average absolute difference in execution time between input keys and the relative execution times (relative to the slowest unprotected version) for each design point for our two processors. The whiskers on the graph represent the 5th and 95th percentile. The x marks the unprotected application; circles represent compilation points with different subsets of $\mathbb{S}$ protected with only CFG transformations; the gray filled circles represent compilation points protecting the full set $\mathbb{S}$ with CFG transformations and different combinations of data flow transformations. To interpret the absolute numbers correctly, it is useful to know that the unprotected application using the slowest key takes 69.97ms on the Core™ 2 and 68.66ms on the Core™ i7 in steady state. So the variations on the unprotected application are in the order of 20–23%.

Using only CFG transformations, the delta in execution time between input keys can be reduced by more than two factors of magnitude, i.e., to 126µs on the Core™ 2 and to 82µs on the Core™ i7 at a remaining overhead of 4.96x for the Core™ 2 and 5.05x for the Core™ i7. Besides protection against timing attacks over noisy communication channels, this compilation strategy also protects against instruction cache attacks and branch prediction attacks because all input-dependent control flow is if-converted.

Applying various data flow transformations further reduces the observed difference in execution time, although there is a clear difference between architectures. On the Core™ i7, only the 4 design points with the highest overhead (rows 15, 16, 17, and 18 of Table 2) significantly reduce the execution time difference compared to the design points with only CFG transformations. All of these four compiler strategies insert 12 nops in between memory operations to avoid data-dependent load/store forwarding. On the Core™ 2 processor almost all compilation points with data flow mitigation techniques reduce the difference in execution time compared to CFG-only compilation points.

In summary, the majority of key-dependent timing variation can be removed by applying CFG tranformations. Data flow transformations to further reduce the variation are architecture-specific and come with a significant overhead.

4.3.3 Statistical Evaluation

Next, we apply statistical tests to identify for which design points the execution times of the two input keys are indistinguishable, and thus provide complete protection. We use the Anderson-Darling test [48] to evaluate how many samples an attacker (with direct access to a system to perform timing measurements) needs to collect to reliably distinguish between the two input keys, and thus to evaluate how difficult it is to perform an actual attack. For each compilation strategy and for different sample set sizes ranging from 20 to 990, we use the test to compare the timing results. These 20–990 samples used in the statistical test always exclude the first 10 collected samples during the timing measurements to eliminate the influence of the initial compilation on the processor pipeline, caches, buffers, etc.

In our initial test results, we observed that the test occasionally reported false positives, such as when it reported a significant difference in execution times over two samples sets collected for the same input. Given the very small

time scale on which we try to measure differences, and the chaotic nature of computers, such false positives are to be expected [49], [50]. To prevent drawing false conclusions from occasional false-positive results, we completely reran each data collection and statistical testing 12 times for each compilation strategy and sample set size, i.e., using a different sample set for each of the 12 experiments.

The columns labeled 20–990 in Table 2 show the results. Each cell contains the number of Anderson-Darling tests that indicated it is impossible to distinguish the execution times of the different input keys for a given sample set size. Values on a dark background indicate that the majority of the statistical tests indicate no significant difference between the execution times, and thus that the application can be considered secure for the given sample size. The lighter the background, the more tests indicate a difference in execution time, and hence the easier it is to perform an attack for an attacker that can collect that amount of samples.

Again, we observe a clear difference between the two processors. If an attacker has 990 timing samples for each key, the JIT compiler provides full protection on the Core™ 2 at the lowest overhead using the compilation strategy on row 11. This strategy protects all methods in $\mathbb{S}$, creates dummy array objects of size 64 and replaces divisions by a constant time implementation. For this design point, none of the 12 tests were able to distinguish between the two input keys. On the Core™ i7 complete protection can be provided at the lowest overhead by compiling the application using the design point on row 16. The application needs to have all methods in $\mathbb{S}$ protected, dummy array objects of size 64 and 12 nop instructions inserted between memory instructions. For this design point, 11 of the 12 statistical tests indicate no significant difference between the two sample sets.

When an attacker in some scenario would only be able to collect 20 samples, protected is possible with less overhead. On the Core™ 2 the results for the compilation strategy on row 5 show that only 30 of the 32 methods in $\mathbb{S}$ need to be protected with only the CFG transformations for 11/12 tests to indicate no significant difference. On the Core™ i7protecting all methods in $\mathbb{S}$ with CFG transformations results in 10/12 statistical tests indicating no difference between the inputs, as shown by the results on row 7.

We conclude that each processor has its optimal compilation strategies, depending on the attackers strength, and we proved statistically that no timing information leaks.

4.3.4 Memory Usage and Garbage Collection

To measure the memory and garbage collection overhead of our approach we performed 300 encryptions using 7 different versions of our application: An unprotected version running on a clean JikesRVM build without modifications, an unprotected version running on our modified JIT compiler and five equally protected versions (row 11 of Table 2), but each with differently sized dummy arrays ranging from 1 to 64 bytes. For each of these versions Table 3 reports the total heap memory collected, the minimum required amount of heap memory to run the encryption algorithm and the percentage of time spent garbage collecting in steady state.

Between the clean build and the unprotected version, 203MB additional memory is collected and the minimum required heap size is increased by 2MB. The latter is mainly due to the additional infrastructure needed for our protection framework. When protection is enabled with dummy array sizes of 1, the total collected memory increases by 1445MB and increases further with growing dummy array sizes. The minimum required heap size, however, remains the same as when executing the unprotected application. This is because the maximum heap size is reached during the initial startup phase, when the VM is booted and the application is loaded and compiled.

There is no difference in the amount of time spent garbage collecting in steady state execution between the clean build and unprotected version. When protecting the application, the percentage of time spent in garbage collecting increases with the size of the allocated dummy arrays. The influence of garbage collection on the total execution time remains minimal, with a maximum of 0.101% of total execution time spent garbage collecting in the protected application with dummy array sizes of 64.

Fig. 8: Core™ 2 JIT compilation times.

4.3.5 Compilation Overhead

The compilation time increases for three reasons. First, more code needs to be compiled: In case a method is invoked from within and from outside of the protected code region, the extended compiler needs to generate a secured one and a non-secured one version. Secondly, the extended JIT compiler applies its additional transformations, which also requires computation time. Lastly, the modifications made to the compiler infrastructure introduce some extra compiler overhead, even when the application runs unprotected.

Figure 8 shows the JIT compilation times to compile the application on the Core™ 2 for the unmodified JikesRVM compiling an unprotected application, and for the modified framework compiling a fully protected (with all transformations applied to all methods in $\mathbb{S}$) and an unprotected version. Due the modifications made in the compiler, the unprotected application compiles 22.5% slower. We believe there is a lot of room for improvement with additional engineering because until now optimizing compilation time was not a primary goal in our prototype implementation effort. The compilation overhead of the protected application compared to the unprotected application is 11.3%. This overhead can be divided into additional compiled methods (2.9%) and the extra security transformations (8.5%).

The reported slowdowns will be experienced during the startup of the application (more precisely, at the first invocation of a sensitive region), which will be slowed down as all code then needs to be compiled from scratch. Furthermore, whenever code needs to be recompiled because a new protection requirement is imposed, the price of full recompilation of the methods in the secured code regions will have to be paid, which will be experienced as a temporary halting

	clean	unprotected	protected(1)	protected(8)	protected(16)	protected(32)	protected(64)
total memory collected (MB)	906	1109	2554	3585	4737	7069	11660
minimum required heap size (MB)	30	32	32	32	32	32	32
% of time spent in GC	0.015	0.015	0.017	0.024	0.035	0.057	0.101

TABLE 3: Memory and garbage collection statistics for different application versions running on the Intel Core™ 2 Duo.

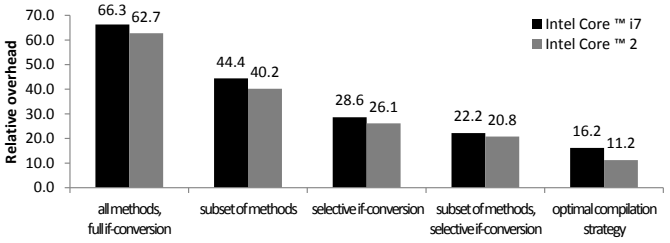

Fig. 9: Slowdowns for modPow with different protections.

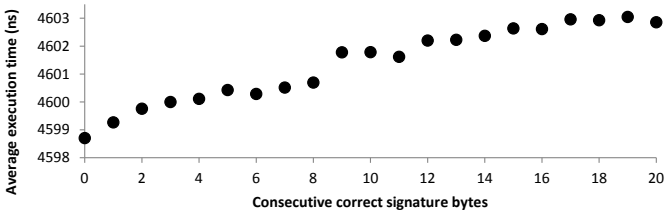

Fig. 10: Average HMAC execution time on the Core™ 2.

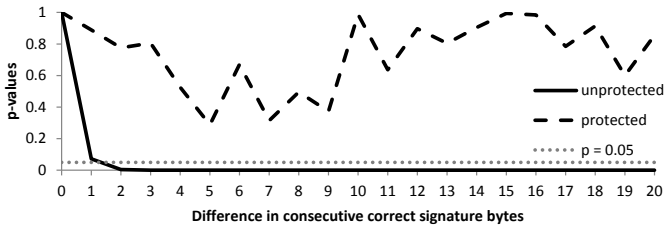

Fig. 11: Anderson-Darling test results on the Core™ 2.

of the application. In case the protection requirements are lowered, it is conceivable to perform the recompilation in the background while continuing to run the more heavily protected code to avoid stuttering execution. Previously compiled versions of methods can be stored in a cache to avoid the cost of recompilation when switching between protection levels that have already been compiled.

4.3.6 *Comparison to Previous State Of The Art*

Figure 9 reports the overhead of our protection techniques for the BigInteger modPow(BigInteger exponent, BigInteger modulus) method. This is the method detected in iteration 1 of Algorithm 1 as the source of timing variation for the RSA encryption routine. The Y-axis denotes the execution times relative to the execution time of the unprotected application using the input key yielding the slowest execution time. Protecting all methods reachable from the modPow method, if-converting all branches within those methods and applying all available protection techniques, which is the approach used in the previous state of the art [37], results in an overhead of 66.3x on the Core™ i7. Protecting only a subset of methods reduces the overhead to 44.4x and partial if-conversion reduces the overhead to 28.6x. Combining the two techniques reduces the overhead further to 22.2x. The graph shows similar results on the Core™ 2.

When compiling with the optimal security strategy the overhead can be reduced further: On the Core™ i7 to 16.2x using the compilation strategy on row 17 in Table 2 and on the core 2 to 11.2x using the strategy in row 12. In total we reduce the overhead by 75.6% on the Core™ i7 and 82.2% on the Core™ 2 while still providing the same level of protection. As can be seen in Table 2, less secure application versions have even lower overhead.

The cause of the still significant overhead of our transformations on the RSA algorithm is that a large call tree has to be protected. This is partly due to the structure of the algorithm itself, and the fact that general purpose class BigInteger is used internally by the BouncyCastle library to do most computations. Furthermore, the BigInteger class stores its values in memory, causing data flow mitigation techniques to introduce additional overhead.

4.4 Keyczar HMAC Verification

For a second experiment, we used an older version of the Google Keyczar library. The HMAC key verification in this version is known to leak timing information because it uses a standard Arrays.equals(byte[],byte[]) function call to compare HMAC signatures [51]. This function leaks information because it compares two byte arrays byte by byte and returns as soon as it encounters two unequal bytes. Although this particular leak has been fixed since May 2009, it can still serve to show how our profile-based approach can easily detect and fix this type of early-exit timing leaks.

Figure 10 shows the average execution time based on 10.000 samples per signature of the unprotected verification function for 21 different signatures with increasing amounts of consecutive correct bytes. Signature 0 with 0 correct bytes has the fastest verification time, the correct signature 20 (in theory) takes the longest to verify. Although the ordering is not 100% correct due to noisy measurements, there is a clear relationship between the amount of consecutive correct signature bytes and the execution time. We marked the Signer.verify(byte[] text,byte[] signature) as the root of protection and applied our profile-based protection techniques, using a correct and completely incorrect key as test inputs. In this experiment all (in)direct callees of the root method have equal outgoing edge counts for both keys, hence Algorithm 1 did not detect any methods to protect. However, Algorithm 2, based on instruction counts, correctly tagged Arrays.equals(byte[],byte[]) as the source of timing variation.

The optimal protection strategy in this experiment consists of all control flow transformations and inserting 12 nops between memory operations. Figure 11 shows the results of the Anderson-Darling test comparing keys with different consecutive correct signature bytes for 10,000 samples

per key. P-values below the critical value of 0.05 indicate that we can reject the null-hypothesis that there is no statistical difference between the sample sets. In other words, a p-value above 0.05 indicates it is impossible for an attacker to differentiate between them. In the unprotected program version we are able to differentiate between keys that have two or more bytes difference, while in the protected version there is no statistical difference between the signatures.

The overhead of our mitigation technique in this test case is minimal: 7.5% with all mitigating transformatins enabled and only 5.7% using the optimal protection strategy.

4.5 BouncyCastle IDEA encryption

Finally, we mitigated a well-known chosen-plaintext attack on the IDEA encryption algorithm. The attack relies on the timing behavior of the modular multiplication algorithm used during encryption, which has a faster execution time when the 16 least significant bits of the modulus or exponent are 0. This timing side channel allows an attacker to reconstruct the 16 least significant bits of the secret key. We reproduced the attack described by Lux et al. [52]: For a given secret key, we timed the encryption of 16,777,215 8-byte pseudo-random plaintext messages. We then grouped these timings in 65,536 clusters, based on the 16 least significant bits of the produced ciphertext. The 16 bits (X) corresponding to the cluster with the fastest average execution time then yield part of the secret key as $key[70:85] \leftarrow (2^{16}+1) - X$.

We reproduced the attack using the unprotected BouncyCastle implementation of the IDEA algorithm. Figure 12a) shows the clusters with the ten fastest average execution times, generated by performing the above attack using `0x00e148d92641ecf99d428836b7bf0150` as secret key. The cluster corresponding to `0xAF5F`, indicated in gray, clearly has the fastest execution time.

In this case, the call edge heuristic of Algorithm 1 did not find any methods to protect. The instruction count heuristic of 2, however, correctly tagged `private int mul(int x, int y)` of the `IDEAEngine` class as the cause of timing variation. Profile information marked a single branch in this method for if-conversion. Since the protected method does not contain any variable latency instructions or memory instructions, no data flow transformations were needed at all.

Figure 12b) shows the timing results obtained on the protected code. There is no significant difference between the ten clusters with the lowest average execution times, so the attack will fail on the protected application. Additional statistical Anderson-Darling tests indicate that there is no significant difference between the cluster corresponding to the `AF5F` bits and the other clusters. We repeated the experiment on the protected version of the application with 10 different encryption keys with identical results. The overhead of our mitigation technique on this test case is 16%.

4.6 Limitations & Future Work

Our mitigation techniques, as implemented today, come with some important limitations.

Granularity of applied transformations: In our current implementation, the applied data flow transformations are the same for all methods, and they are applied to their whole bodies. In the future, we plan to refine this, such that different strategies can be applied to different methods and even to parts of methods, and such that multiple secured versions can co-exist, such that differently protected versions can be invoked in different call chains.

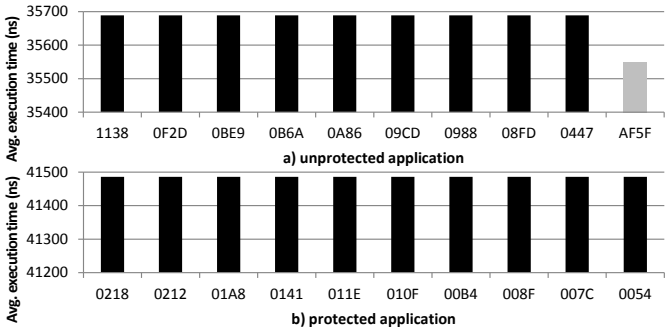

Fig. 12: Ten clusters with lowest average execution time for the protected and unprotected application.

Optimal strategy: Research is required to automate the search for the optimal compilation strategies for an application and target architecture based on code characteristics, without having to rely on predefined compilation strategies. Furthermore, more research is needed into a user-friendly way for users to express their protection requirements for different run-time scenarios, and to decide on the corresponding compilation strategy.

Exceptions: We currently assume exceptions are a rare occurrence and are not part of the time-critical sensitive code. Code within an exception handling region can of course be secured, but the throwing of an exception is not if-converted. In cryptographic code, we observed exceptions to be rare except for validating the correct format of inputs. More research is needed, however, to protect code with exceptions in other applications. Currently, the automated support is limited to producing a warning if exceptions occur in code regions tagged for protection by our algorithms.

Native Code and System Calls: Our setup currently only supports protecting bytecode against timing attacks. System calls and native code can still cause the application to leak timing information. For native code a solution could be to provide both a normal version and one secured with static techniques [26], [29], [37].

Multithreading: We currently protect only single-threaded code regions, i.e., code that executes within one thread. That thread can, however, be part of a multithreaded application.

Virtual method calls: Our current implementation assumes that target methods of virtual calls remain constant during the execution of the application. In other words, the run-time type of the object on which the method is invoked has to remain the same. This restriction does not pose problems for the crypto libraries we experimented with. If the need would arise in the future, it is possible to extend our implementation based on class hierarchy analysis and points-to analysis, and by converting polymorphic virtual calls into switch-like constructs.

VM side channels: We assume side-channel free implementations of VM service routines (i.e., routines invoked by the compiled bytecode) and native methods dealing with sensitive information. In our experiments, it sufficed

to patch the intrinsics generating code for long conditional move and long multiplications, which were leaking timing information. Native functions can be compiled using existing static compiler techniques [37]. To what extent other VM aspects have to be adapted to become leak-free is future work. At least for cryptographic code, we could not identify parts that might leak information. We already took care of the JIT compiler, and the amount of allocated memory (and hence the behavior of garbage collection) does not depend on the values of (correctly formated) secret data in cryptographic code.

Taint tracking: Rather than marking a root method, taint tracking might be used to identify code regions that need protection. That would likely be more user-friendly. As mentioned in the introduction, the identification by means of taint tracking alone might be overly conservative, however. In particular, existing techniques would either need to be adapted to identify not the code of which the behavior depends on secret values, but the code of which the timing behavior depends on them, or taint tracking would still have to be combined with our profiling-based approach.

5 Related Work

Compared to existing (static) compiler techniques [37], [26] our approach uses profile-based analysis to automate and optimize the selection of the protected code base, reducing the overhead up to 90%. This makes our approach suitable to protect real-life security libraries, where existing techniques would introduce unacceptable overhead. Secondly, JIT compilation allows to optimize security transformations for a target architecture and required security level. Only one (unprotected) version of the application needs to be distributed where static compilation techniques require separate binaries for each architecture and security requirement.

Our approach does not have to rely on operating system modifications to inject enough noise, e.g., in the measurable cache-behavior [30] or pad the execution time of methods and provide resource isolation [53]. Our approach ensures an application can run protected in any target environment, without directly affecting the other processes that happen to share resources with the protected application. Other approaches using JIT compilation rely on dynamic software diversity to mask information leakage [54], [55] while our approach inherently provides complete protection against branch prediction attacks and instruction cache attacks. Whereas we provide statistical evidence that no differences in execution time can be observed, the overhead of our approach can be significantly higher.

While we do not provide a formal proof of our techniques as has been done for complete language-based solutions [32], we provide statistical evidence to show the effectiveness of our approach.

Both hardware [16], [23], [14], [24] and algorithmic [33], [34], [21], [35] approaches can be easily integrated in our framework. The compiler can, e.g., automatically replace instructions that leak timing information by calls to fixed library functions [35], or generate code to take advantage of hardware features such as the Intel AES instruction set [23] or side channel free cache designs [24].

6 Conclusions

We presented a combined offline/online approach for mitigating timing side channels. A JIT compiler generates code for sensitive code regions such that the execution time becomes completely or largely independent of sensitive data values. This approach supports adaptive protection with regards to changes in the underlying hardware and changes in the protection requirements, without requiring code duplication or specialization before the code is distributed.

We presented a profile-based approach with which we significantly reduce the overhead of full protection, and that allows (at least in theory) a trade-off between the provided level of protection and the incurred overhead.

The approach was evaluated on real-life use cases. We were able to protect an RSA encryption algorithm at an overhead of 8.4x and 16.5x on Core™ 2 and Core™ i7 systems respectively. Compared to existing state-of-the-art techniques, our approach reduced the overhead of protecting the modular exponentiation algorithm from 66.3x to 16.2x on the Core™ i7 and from 62.7x to 11.2x on the Core™ 2. Our approach also pinpointed a single method causing leakage in an HMAC verification routine and secured it automatically at an of 5.7%. Finally, we mitigated a timing attack against an IDEA algorithm with an overhead of 16%.

References

[1] O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-Cache," in *Proc. ACM workshop on Computer security architecture (CSAW'07)*, 2007, pp. 11–18.
[2] O. Aciiçmez and Ç. Koç, "Trace-driven cache attacks on AES," in *Information and Communications Security*, ser. Lecture Notes in Computer Science, 2006, vol. 4307, pp. 112–121.
[3] O. Aciiçmez *et al.*, "On the power of simple branch prediction analysis," in *Proc. 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, 2007, pp. 312–320.
[4] ——, "Predicting secret keys via branch prediction," in *Topics in Cryptology - The Cryptographers' Track at the RSA Conf. (CT-RSA'07)*, 2007, pp. 225–242.
[5] P. C. Kocher *et al.*, "Differential power analysis," in *Proc. 19th Annual Int'l Cryptology Conf. on Advances in Cryptology (CRYPTO'99)*, 1999, pp. 388–397.
[6] C. Lauradoux, "Collision attacks on processors with cache and countermeasures," in *Western European Workshop on Research in Cryptology (WEWoRC'05)*, 2005, pp. 76–85.
[7] O. Aciiçmez *et al.*, "New results on instruction cache attacks," in *Proc. 12th Int'l Conf. on Cryptographic Hardware and Embedded Systems (CHES'10)*, 2010, pp. 110–124.
[8] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Proc. 15th Int'l Conf. on the Theory and Application of Cryptology and Information Security (ASIACRYPT '09)*, 2009, pp. 667–684.
[9] D. Gullasch *et al.*, "Cache games - bringing access based cache attacks on AES to practice," Cryptology ePrint Archive, Report 2010/594, 2010.
[10] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *Proc. 13th Int'l Conf. on Selected Areas in Cryptography (SAC'06)*, 2007, pp. 147–162.
[11] D. A. Osvik *et al.*, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology - The Cryptographers Track at the RSA Conf. (CT-RSA'06)*, 2006, pp. 1–20.
[12] T. Ristenpart *et al.*, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proc. 16th ACM Conf. on Computer and Communications Security (CCS'09)*, 2009, pp. 199–212.
[13] L. Uhsadel *et al.*, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08)*, 8 2008, pp. 59–67.
[14] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. 22nd Annual Computer Security Applications Conf. (ACSAC'06)*, 2006, pp. 473–482.

[15] O. Aciiçmez *et al.*, "Cache based remote timing attack on the AES," in *Topics in Cryptology - The Cryptographers' Track at the RSA Conf. (CT-RSA'07)*, 2007, pp. 271–286.
[16] D. J. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.
[17] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Proc. Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES'06)*, 2006, pp. 201–215.
[18] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, August 2005.
[19] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," Cryptology ePrint Archive, Report 2011/232, 2011.
[20] J.-F. Dhem *et al.*, "A practical implementation of the timing attack," in *Proc. The Int'l Conf. on Smart Card Research and Applications (CARDIS'98)*, 1998, pp. 167–182.
[21] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. 16th Annual Int'l Cryptology Conf. on Advances in Cryptology*, 1996, pp. 104–113.
[22] B. Gierlichs *et al.*, "Mutual information analysis," in *Proc. 10th Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES'08)*, 2008, pp. 426–442.
[23] S. Gueron, "Advanced encryption standard (AES) instructions set," Intel Mobility Group, Tech. Rep., 2008.
[24] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 494–505, 2007.
[25] E. Brickell *et al.*, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," Cryptology ePrint Archive, Report 2006/052, 2006.
[26] B. Coppens *et al.*, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proc. 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009, pp. 45–60.
[27] D. Hedin and D. Sands, "Timing Aware Information Flow Security for a JavaCard-like Bytecode," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 1, pp. 163–182, Dec. 2005.
[28] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF'09)*, 2009, pp. 324–335.
[29] D. Molnar *et al.*, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Proc. Int'l Conf. Information Security and Cryptology (ICISC'05)*, 2005, pp. 156–168.
[30] Y. Zhang and M. K. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proc. ACM SIGSAC conf. on Computer & communications security*. ACM, 2013, pp. 827–838.
[31] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proc. 18th ACM Conf. on Computer and Communications Security*, 2011, pp. 563–574.
[32] J. Agat, "Transforming out timing leaks," in *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, 2000, pp. 40–53.
[33] J. Guajardo and B. Mennink, "Towards side-channel resistant block cipher usage or can we encrypt without side-channel countermeasures," Cryptology ePrint Archive, Report 2010/015, 2010.
[34] M. Joye and S.-M. Yen, "The montgomery powering ladder," in *Revised Papers from the 4th Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, 2003, pp. 291–302.
[35] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE Symposium on Security and Privacy*, 2015, pp. 623–639.
[36] A. G. Bayrak *et al.*, "A first step towards automatic application of power analysis countermeasures," in *Proc. 48th Design Automation Conf. (DAC'11)*, 2011, pp. 230–235.
[37] J. Van Cleemput *et al.*, "Compiler mitigations for time attacks on modern x86 processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 23, 2012.
[38] M. Kim *et al.*, "Utilization-aware load balancing for the energy efficient operation of the Big.LITTLE processor," in *Proc. Conf. on Design, Automation & Test in Europe*, 2014, pp. 223:1–223:4.
[39] S. A. Crosby *et al.*, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security*, vol. 12, no. 3, pp. 17:1–17:29, January 2009.
[40] J. Groszschaedl *et al.*, "Side channel analysis of cryptographic software via early-terminating multiplications," in *Proc. 12th Int'l Conf. on Information security and cryptology*, 2009, pp. 176–192.
[41] J. R. Allen *et al.*, "Conversion of control dependence to data dependence," in *Proc. 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'83)*, 1983.
[42] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2014.
[43] *ARMv8 Instruction Set Overview*, ARM, 2014.
[44] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.
[45] B. Alpern *et al.*, "The jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.
[46] M. Arnold *et al.*, "Architecture and policy for adaptive optimization in virtual machines," IBM Research, Tech. Rep. 23429, 2004.
[47] M. Backes and B. Köpf, "Formally bounding the side-channel leakage in unknown-message attacks," in *Computer Security-ESORICS 2008*. Springer Berlin Heidelberg, 2008, pp. 517–532.
[48] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.
[49] T. Mytkowicz, A. Diwan, and E. Bradley, "Computer systems are dynamical systems." *Chaos (Woodbury, N.Y.)*, vol. 19, no. 3, p. 033124, Sept. 2009.
[50] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proc. 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, 2009, pp. 265–276.
[51] N. Lawson, "Timing attack in Google Keyczar library," http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/, 2009, [Online; accessed 20-August-2015].
[52] A. Lux and A. Starostin, "A tool for static detection of timing channels in java," *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 303–313, 2011.
[53] B. A. Braun *et al.*, "Robust and efficient elimination of cache and timing side channels," *CoRR*, vol. abs/1506.00189, 2015.
[54] S. Crane *et al.*, "Thwarting cache side-channel attacks through dynamic software diversity," in *Network And Distributed System Security Symposium, NDSS*, vol. 15, 2015.
[55] M. Hataba *et al.*, "Diversified remote code execution using dynamic obfuscation of conditional branches," in *Proc. IEEE 35th Int'l Conf. on Distributed Computing Systems Workshops (ICDCSW)*, 2015, pp. 120–127.

Jeroen Van Cleemput worked as a postdoctoral researcher at Ghent University in the Computer Systems Lab. He obtained his Msc. degree in Computer Science from Ghent University's Faculty of Engineering in 2009 and his Ph.D. degree in Computer Science from Ghent University's Faculty of Engineering in 2016. His research focuses on compiler based protection techniques against side-channel attacks.

Koen De Bosschere is professor at Ghent University, where he teaches courses on computer architecture and operating systems. His current research interests are binary translation, virtualization, and software protection. He authored and co-authored over 170 papers. He is the coordinator of the HiPEAC network, the ACACES summer school and the the student entrepreneurship program at Ghent University.

Bjorn De Sutter is professor at Ghent University in the Computer Systems Lab. He obtained his Msc. and Ph.D. degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques and run-time techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection. He published over 80 peer-reviewed papers on these topics.