Dude, is my code constant time?

Oscar Reparaz, Josep Balasch and Ingrid Verbauwhede
KU Leuven/COSIC and imec
Leuven, Belgium

***Abstract*—This paper introduces `dudect`: a tool to assess whether a piece of code runs in constant time or not on a given platform. We base our approach on leakage detection techniques, resulting in a very compact, easy to use and easy to maintain tool. Our methodology fits in around 300 lines of C and runs on the target platform. The approach is substantially different from previous solutions. Contrary to others, our solution requires no modeling of hardware behavior. Our solution can be used in black-box testing, yet benefits from implementation details if available. We show the effectiveness of our approach by detecting several variable-time cryptographic implementations. We place a prototype implementation of `dudect` in the public domain.**

Keywords. Timing attacks · leakage detection · tools for secure software implementations · secure coding methodology · constant-time implementations.

I. Introduction

Timing attacks are a broad class of side-channel attacks that measure the execution time of a cryptographic implementation in order to infer secrets, such as keys or passwords. Kocher writes in 1996 his seminal paper on recovering cryptographic keys by measuring the execution time of cryptographic implementations [Koc96]. More concretely, he exploits data-dependent variance in the execution time of RSA, DSS or Diffie-Hellman to recover secret key bits in a divide-and-conquer fashion. Since then, timing attacks have been broken numerous variable-time implementations, including high-impact systems such as TLS [AP13]. In comparison to other side-channel attacks, timing attacks require minimal interaction with the target and can be mounted remotely [BB03].

Assessing whether an implementation runs in constant time is not a trivial task. Even implementations that were supposed to be constant-time turned out not to be so [GBY16], [Cry16]—reinforcing the argument that timing leaks may not be easy to detect, but can have serious consequences.

Security-conscious practitioners have traditionally relied on manual inspection of assembly code to assess information leakage by timing channels. The practitioner typically checks that there are no memory indices or branches that are secret-dependent. Higher-level code that gets compiled can be inspected (at the compiled code level) for constant-time in the same way. This manual approach can be very time consuming already for moderately sized code bases, and should be repeated for every exact combination of compiler flags being used.

There are several tools already for detection of variable-time code. Langley's `ctgrind` [Lan10] extends Valgrind, a dynamic analysis tool. This tool tracks secret-dependent paths or memory accesses. Flow-tracker is a tool by Rodrigues Silva that "finds one or more static traces of instructions" that lead to timing variability [RQaPA16]. Almeida and others' `ctverif` builds on a fruitful series of formal methods tools to build a static analyzer for assessing whether a piece of code is constant time or not [ABB+16].

A common drawback is that these methods have to model somehow the hardware. However, this is extremely difficult. CPU manufacturers publish little information on the inner workings of the CPU. Furthermore, this behavior is subject to change by e.g. a micro-code update. In short, while the principles the previous tools rely on are sound, correct hardware models are not easy to build [Ber14].

Our contribution. In this paper we present `dudect`: a tool to detect whether a piece of code runs in constant-time or not on a given platform. Our approach is very simple in nature. We leverage concepts from the hardware side-channel domain and port them to our context. More precisely, we base our approach on leakage detection tests. The result is a very compact tool that can be used to test timing variability in cryptographic functions in an easy way. Our tool does not rely on static analysis but on statistical analysis of execution timing measurements on the target platform. In this way, we circumvent the problem of modeling the underlying hardware.

II. Our approach: timing leakage detection

Our approach in a nutshell is to perform a leakage detection test on the execution time. We first measure the execution time for two different input data classes, and then check whether the two timing distributions are statistically different. Leakage detection tests were introduced by Coron, Naccache and Kocher [CKN00], [CNK04] shortly after the introduction of DPA [KJJ99] and were targeted towards hardware side-channel evaluations.

A. Step 1: Measure execution time

First off, we repeatedly measure the execution time of the cryptographic function under study for inputs belonging to two different input data classes.

a) Classes definition: Leakage detection techniques take two sets of measurements for two different input data classes. There are several families of leakage detection tests, mostly differing in the way the input data classes are defined. A strand of leakage detection tests that is known to capture a wide range of potential leakages is fix-vs-random tests [GJJR11], [CDG+13]. Typically, in a fix-vs-random leakage detection test, the first class input data is fixed to a constant value, and the second class input data is chosen at random for each measurement. The fixed value might be chosen to trigger certain

"special" corner-case processing (such as low-weight input for arithmetic operations).

b) Cycle counters: Modern CPUs provide cycle counters (such as the TSC register in the x86 family; or the systick peripheral available in ARM), that can be used to accurately acquire execution time measurements. In lower-end processors one can resort to high-resolution timers when available, or use external equipment.

c) Environmental conditions: To minimize the effect of environmental changes in the results, each measurement corresponds to a randomly chosen class.[1] The class assignment and input preparation tasks are performed prior to the measurement phase.

B. Step 2: Apply post-processing

The practitioner may apply some post-processing to each individual measurement prior to statistical analysis.

a) Cropping: Typical timing distributions are positively skewed towards larger execution times. This may be caused by measurement artifacts, the main process being interrupted by the OS or other extraneous activities. In this case, it may be convenient to crop the measurements (or discard measurements that are larger than a fixed, class-independent threshold).

b) Higher-order preprocessing: Depending on the statistical test applied, it may be beneficial to apply some higher-order pre-processing, such as centered product [CJRR99] mimicking higher-order DPA attacks. Higher-order leakage detection tests already appeared in other contexts [SM15].

C. Step 3: Apply statistical test

The third step is to apply a statistical test to try to disprove the null hypothesis "the two timing distributions are equal". There are several possibilities for statistical tests.

a) t-test: A simple statistical test to use and implement is Welch's t-test. This statistic tests for equivalence of means, and hence failure of this test trivially implies that there is a first-order timing information leakage (that is, the first order statistical moment carries information). Note that when a t-test is coupled with cropping pre-processing, one is no longer testing only for equality of means but also for higher order statistical moments since the cropping is a non-linear transform (in the same way as one uses a non-linear pre-processing function to perform a higher-order DPA).

b) Non-parametric tests: One can use more general non-parametric tests, such as Kolmogorov-Smirnov [WOM11]. The advantage is that these tests typically rely on fewer assumptions about the underlying distributions; the disadvantage is that they may converge slower and require more samples.

III. Results

A. Implementation

A prototype was built in around 300 lines of C and is available from https://github.com/oreparaz/dudect. This shows the inherent simplicity of our approach. The following experiments are executed on a 2015 MacBook and compiled with LLVM/`clang-703.0.31` with optimization `-O2` turned on unless otherwise stated. We detail in the following several implementation choices. Each of the following experiments runs for 120 seconds.

a) Pre-processing: We pre-process measurements prior to statistical processing. We discard measurements that are larger than the p percentile, for various[2] values of p. The rationale here is to test a restricted distribution range, especially the lower cycle count tail. The upper tail may be more influenced by data-independent noise. This (heuristic) process gives good empirical results (makes detection faster); nevertheless we also process measurements without pre-processing.

b) Statistical test: We use an online Welch's t-test with Welford's variance computation method for improved numeric stability [Knu81, §4.2.2.A].

B. Memory comparison

Our first case study is a memory comparison routine. This task appears in many cryptographic contexts that require constant time execution. Two examples of such contexts are password verification and message authentication tag verification.

a) Variable-time implementation: As a smoke test, we implement a straightforward 16-byte memory comparison function based on `memcmp` and test its timing variability. The function compares an input string against a "secret" fixed string s.

Our first test harness is as follows. The first class for the leakage detection test considers uniformly distributed random 16-byte strings ("random class"); the second class fixes the input to s ("fix class"). (This assumes a white-box evaluator that has access to implementation internals.)

In Figure 1 we plot the cumulative distribution function (cdf) for both timing distributions corresponding to the fix and random classes. We can see that a single execution of the function takes less than 100 cycles. More importantly, we see that the distributions for the two classes are clearly different, that is, there is timing leakage about the input data.

The results of the statistical test used to discern if the two distributions are different are plotted in Figure 2. We plot the evolution of the t-statistic (absolute value) as the number of measurements increases. The different lines indicate various values of pre-processing parameter p from Section III-A. A t value larger than 4.5 provides strong statistical evidence that the distributions are different[3]. This threshold divides the plot into two regions: red background (above the 4.5 threshold) and green background (below). We can see that all lines surpass the 4.5 threshold for any number of measurements (they belong to the red background, and some of them achieve the whopping

[1] In the original leakage detection paper [CKN00] the authors propose to interleave the sequence of classes during the evaluation. We extend this suggestion and randomize this sequence to prevent interference due to any concurrent process that may be executing in parallel.

[2] The exact values for p follow an inverse exponential trend, see https://github.com/oreparaz/dudect/blob/master/src/fixture.c#L57 for the concrete specification.

[3] The t statistic tests for equality of means; since measurements are pre-processed here it indicates that the distributions are somehow different.

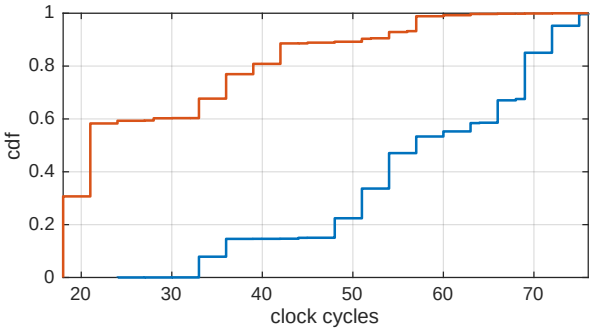

Fig. 1: Timing distributions (cdfs) for `memcmp`-based variable time memory comparison, for two different input classes. Timing leakage is present.

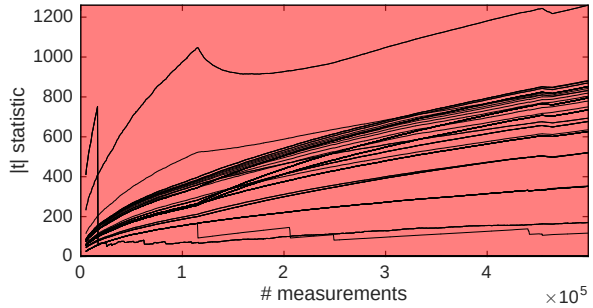

Fig. 2: Evolution of the $|t|$ statistic as the number of traces increases (same implementation of Figure 1). Different lines correspond to different pre-processing parameters. Clearly surpassing the 4.5 threshold.

value of $|t| = 1000$). Some lines (i.e. some threshold values for p) grow faster, but almost all make the test fail. The figure shows that timing leakage is detectable even when few thousand measurements are available, as expected. We can also observe the asymptotic behavior of the t statistic: it grows as $\sqrt{N}$ where N is the number of samples.

Now we perform a slight variation on the test fixture. Namely, instead of assuming that the evaluator knows the secret s (which is internal to the implementation), we are relaxing the evaluator capabilities and assume black-box testing (i.e., the evaluator does not know s). Thus, the value for the fixed class is set to 0, and not s as in the previous case. In this setting, as Figure 3 shows, the two timing distributions are much closer to each other, but they are still different. The statistical tests still reject the null hypothesis that both distributions are the same, as Figure 4 shows, albeit they need more measurements to get statistically significant results. The t statistic values are much lower than in the previous case but still significant. Some pre-processing strategies did not lead to faster detection, as it can be seen from Figure 4.

b) Constant-time implementation: We also analyze an implementation of memory comparison that is supposed to run in constant time. (This function performs the comparison by logical operations, and does not abort early on mismatch.) In Figure 5 we can see that both empirical distributions seem to be equal. The statistical test of Figure 6 indicates that both

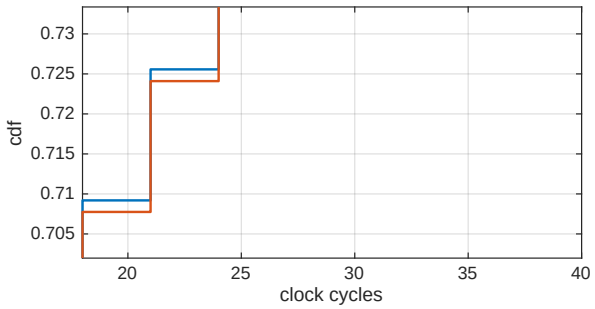

Fig. 3: Similar to Figure 1, but assuming a black-box evaluator. Timing leakage is present, but harder to detect. (Zoomed picture.)

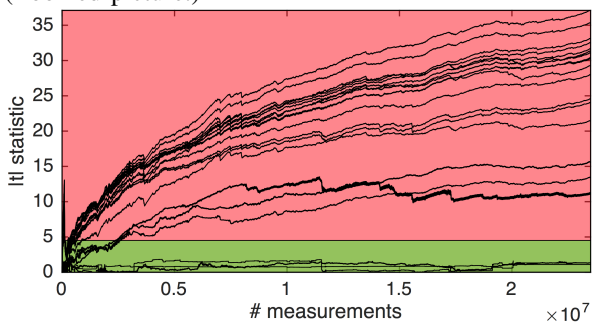

Fig. 4: Similar to Figure 2, but assuming a black-box evaluator. Timing leakage is detectable, although a bit harder than in Figure 2.

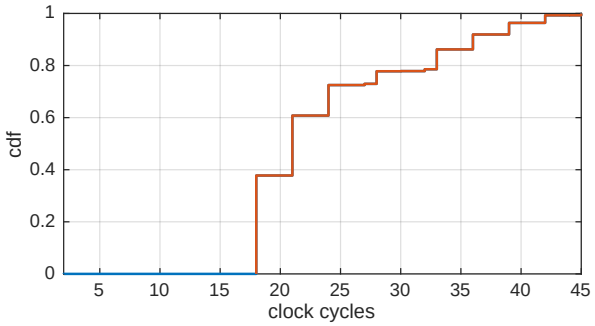

Fig. 5: Similar to Fig. 1, but for a constant time memory comparison function. There are two overlapped cdfs. No apparent timing leakage.

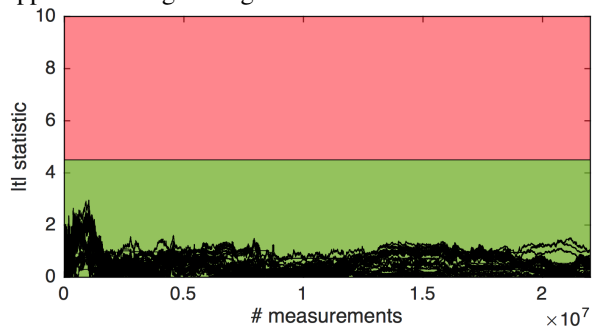

Fig. 6: Similar to Figure 2, but for a constant time memory comparison function. No test rejects the null hypothesis, thus, no timing leakage detected.

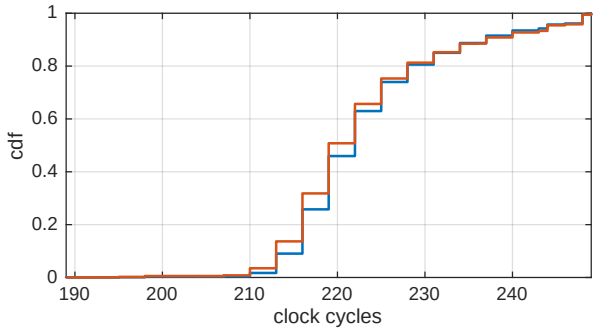

Fig. 7: Timing distributions for the T-tables AES implementation. The two classes induce clearly different distributions.

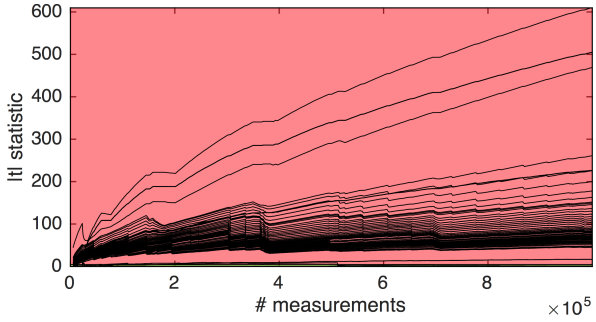

Fig. 8: Evolution of the t statistic for the T-tables AES implementation. Clear leaks are detected very quickly, all pre-processing parameter choices for p lead to leakage detection.

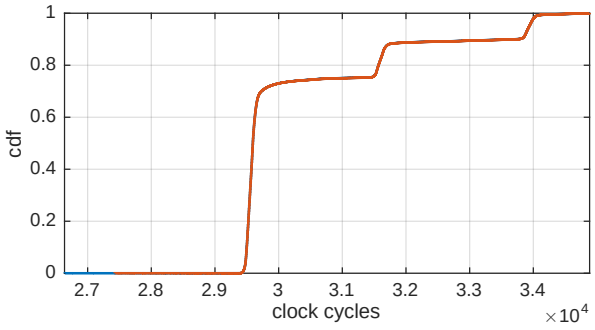

Fig. 9: Timing distributions for the bitsliced AES implementation. There are two overlapped cfs. No apparent timing leakage.

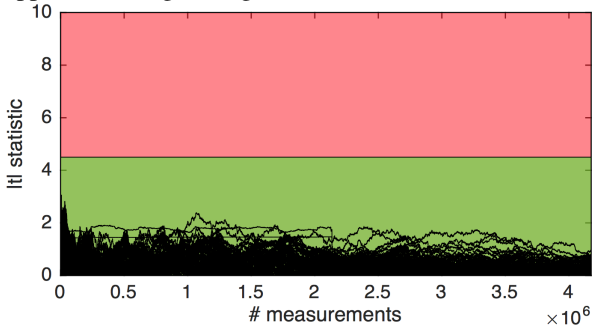

Fig. 10: Evolution of the t statistic for the bitsliced AES implementation. No leakage detected.

distributions are indeed indistinguishable up to 20 million measurements.

C. AES

We also test several AES128 implementations to confirm the validity of our approach.

a) T-tables implementation: We first test the classic T-tables AES implementation for 32-bit processors. We take the reference code `rijndael-alg-fst.c` by Rijmen, Bosselaers and Barreto and plug it in our tool. The fix class corresponds to a (randomly chosen) input plaintext, the key is chosen at random but kept fix within all executions. The key schedule is not included in the cycle count measurements. In Figure 7 we can see quite some timing variability depending on the input data. This is expected, as the implementation is known to be vulnerable to timing attacks [Ber05]. In Figure 8 we can see that indeed the statistical tests reject the null hypothesis with as few as a couple thousand measurements.

b) Bitsliced: We also test a purported constant-time implementation based on bitslicing. We take the code by Käsper and Schwabe [KS09] included in the `libsodium` library.[4] As in the previous case, the key is fixed and the key schedule is not included in the cycle count measurements. In Figure 9 we can see that indeed different input data seem to yield the same timing distribution[5]. In Figure 10 we can observe that the statistical test fails to reject the null hypothesis of both distributions being equal, up to 4 million measurements. (In this case, we also left the test running overnight until almost 1 billion measurements with identical results.)

c) Vector permutations: We also tested the AES implementation by Hamburg [Ham09]. This implementation features SSSE3-specific instructions such as `pshufb` vector permute and is written in assembly. We run it on an Intel Xeon "Westmere" processor. We left it running until we acquired 1 billion measurements. No leakage was detected.

D. Curve25519 on an ARM7TDMI

This subsection is a case study of applying `dudect` on an embedded platform. Our target platform is a 32-bit Atmel's AT91SAM7S ARM7TDMI processor. We focus on a Curve25519 elliptic-curve scalar multiplication function purported to be constant time on x86 processors.

For this experiment, we divide the codebase of `dudect` into two parts. Most of step 1 from Section II runs on the target ARM7TDMI platform and the rest runs on the host computer. (In particular, statistics are run on the host computer.) The execution time itself is measured by the host computer, which communicates with the ARM7TDMI via GPIO pins. Note that other approaches are possible, as explained in Section II-A.

[4]We are leaving aside the mode of operation and testing the AES128 primitive in isolation.

[5]Note that this implementation packs several plaintexts into a single execution, so absolute cycle counts from Figure 9 are not comparable with those from Figure 7.

We cross-compile stock `curve25519-donna` with `arm-elf-gcc` version 4.1.1 and optimization flags `-O2` to produce code running on the target platform.

The test harness is as follows. The first class considers uniformly distributed random input points; the second class fixes the input to all-zeros. In Figure 11 we plot the cdf for the resulting distributions. We can see that both distributions are indeed different, that is, the execution time depends on the input. The variance for the fixed class (orange cdf) is noticeably smaller. Figure 12 shows the results of the t-test. It clearly detects leakage from a few hundred measurements. Thus, this implementation is deemed variable-time.

The timing variability likely comes from the multiplication instructions. ARM7TDMI processor cores feature a variable-time multiplier, as documented by ARM [ARM04, §6.6]. More concretely, the hardware multiplication circuit early terminates if some bits of the operands are 0. This can justify that this executable `curve25519-donna` is not constant time. The same behavior arises in other processor families (such as the ARM Cortex-M3 family). Großschädl and others have presented SPA attacks based on this behavior [GOPT09].

Interestingly, the same source code is considered to be constant time for x86 architectures, and this fact can even be "certified" [ABB+16]. We would like to stress that such "certification" is bound to a specific processor behavior model and relies on the accuracy of such model. As we have seen, even the same code can show different behavior under different toolchain, processor, architecture or other parameters.

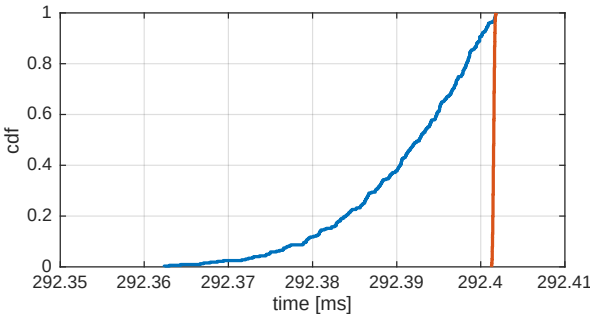

Fig. 11: Timing distributions for the Curve25519 implementation running on an ARM7TDMI. There is apparent timing leakage.

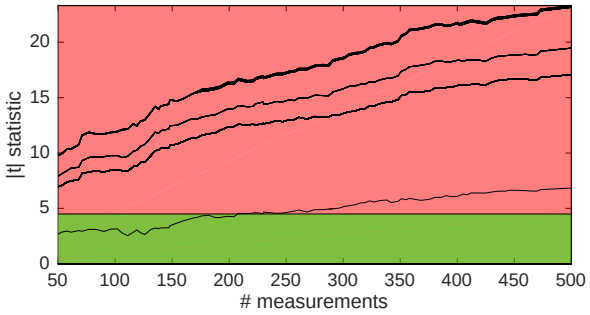

Fig. 12: Evolution of the t statistic for the Curve25519 implementation running on an ARM7TDMI. Leakage detected.

IV. Discussion

a) Tools required: Our approach is very simple, and does not require any exotic tools to run: just a C compiler and a method to measure the execution time. Thus, it can be integrated into deployed building systems relatively easily.

b) Computational effort: The computational effort is normally dominated by the execution of the cryptographic primitive to be tested. The statistical processing is very lightweight in terms of computation, and negligible in terms of memory. Memory requirements stay constant independently of the number of measurements taken (thanks to online algorithms).

c) Hardware modeling: A positive aspect of our approach is that we do not require to model anyhow the underlying hardware. We rely on actual measurements, so we have to place fewer assumptions on the hardware than other approaches. This kind of assumptions (for example: that a `MUL` instruction takes constant time, or that a conditional move such as `CMOV` takes constant time) are often the result of empirical observation, and are not documented by the hardware manufacturer. As such, they are subject to change (for instance, if the user acquires a new CPU model, or a micro-code update is installed). In that case, we would need to re-run the evaluation, but we would not have to reverse-engineer the modifications performed to the micro-code (which can be a very time-consuming task).

d) Leakage modeling: Some variants of leakage detection, such as non-specific fix-vs-random, can detect a wide range of leakages, without assuming much on the leakage behavior. This is the approach we implemented. Thus, we do not have to place assumptions on how the implementation may leak, or which subcomponent may cause the timing leakage. There are other tests such as fix-vs-fix that may be less generic but more efficient, as Durvaux and Standaert note [DS16].

e) How many measurements to take?: In the previous section, we saw that in some cases, the statistical test rejects the null hypothesis only starting from a certain number of samples. That is, if a leakage detection test fails to reject the null hypothesis based on N samples, we cannot conclude anything about the expected behavior when using $N + 1$ measurements.

A very similar situation appears in DPA attacks: typically they require a minimum number of measurements to work. In that case, the designer sets normally a target number of measurements ("security level") and claims that the implementation is secure to side-channel attacks *up to* the targeted security level. We conceive a similar approach here: when a leakage detection test deems an implementation time constant, it does so up to a certain number of measurements. It is the evaluator's responsibility to set a realistic and meaningful value for the number of measurements. If more evidence becomes available at a later time (for example, more measurements are available), the results of the evaluation may change (in a similar way DPA evaluations are carried out).

This is because leakage detection tests do not output a hard, categorical, yes/no assessment on the timing variability, but only statistical evidence of timing leakage up to a certain number of measurements.

In addition, leakage detection tests also output the magnitude of the leakage. This can be useful. Timing leaks that are extremely hard to detect (for example, only detectable when using billions of measurements) may be impossible to exploit if the application enforces a limit on the number of executions for the cryptographic operation, so one may deem as usable implementations that exhibit tiny leakages.

f) False positives: Presence of leakage is a necessary condition, but not sufficient, for a timing attack vulnerability. As such, we could find settings that exhibit timing leakage, yet it is hard (or impossible) to exploit. Our tool does not blindly construct a timing attack on the implementation, but rather outputs an indication that it might be susceptible to a timing attack.

g) Crafting good test vectors: As we saw in Section III-B, specially crafted input values lead to larger leakage that is more easily detected. One can typically craft such input values if the implementation details are known. Obviously, the choice of these "special" values highly depends on the algorithm and implementation being tested. For applications to RSA, see the technical report from CRI and Riscure [JRW].

h) Comparison with other approaches: It is very hard to compare with other approaches, since the underlying principles are so different. An important difference with respect to other approaches is that the output of our tool is not a categorical decision, but rather statistical evidence. The complexity of our tool compares favorably with other solutions (which rely on more exotic languages, for example).

V. Conclusion

We described in this paper a methodology to test for timing leakage in software implementations. Our solution is conceptually simple, scales well and has modest speed/memory requirements. Our methodology is based on established tools (leakage detection) from the area of hardware side-channels. We demonstrated the effectiveness of our approach with several software implementations. As future work we consider the problem of assisting the generation of input test vectors. Techniques from fuzz testing may help in this process.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by the Research Council KU Leuven C16/15/058, by the Hercules Foundation AKUL/11/19 and through the Horizon 2020 research and innovation programme under Cathedral ERC Advanced Grant 695305. O. R. was partially funded by an FWO fellowship.

References

[ABB+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 16*, pages 53–70. USENIX Association, 2016.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE SP*, pages 526–540. IEEE Computer Society, 2013.

[ARM04] ARM Limited. ARM7TDMI Technical Reference Manual. Revision r4p1. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf, 2004.

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX*. USENIX Association, 2003.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[Ber14] Daniel J. Bernstein. Some small suggestions for the intel instruction set. https://blog.cr.yp.to/20140517-insns.html, 2014.

[CDG+13] Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, and Pankaj Rohatgi. Test Vector Leakage Assessment (TVLA) methodology in practice. ICMC, 2013.

[CG09] Christophe Clavier and Kris Gaj, editors. *CHES 2009*, volume 5747 of *LNCS*. Springer, 2009.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[CKN00] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache. Statistics and secret leakage. In *Financial Cryptography*, volume 1962 of *LNCS*, pages 157–173. Springer, 2000.

[CNK04] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. Statistics and secret leakage. *ACM Trans. Embedded Comput. Syst.*, 3(3):492–508, 2004.

[Cry16] Timing Attack Counter Measure AES #146. https://github.com/weidai11/cryptopp/issues/146, 2016.

[DS16] François Durvaux and François-Xavier Standaert. From improved leakage detection to the detection of points of interests in leakage traces. In *EUROCRYPT*, volume 9665 of *LNCS*, pages 240–262. Springer, 2016.

[GBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really are Constant-Time. In *CCS, 2016*, pages 1639–1650. ACM, 2016.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2011.

[GOPT09] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC 2009*, volume 5984 of *LNCS*, pages 176–192. Springer, 2009.

[Ham09] Mike Hamburg. Accelerating AES with vector permute instructions. In Clavier and Gaj [CG09], pages 18–32.

[JRW] Josh Jaffe, Pankaj Rohatgi, and Marc Witteman. Efficient sidechannel testing for public key algorithms: RSA case study.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Clavier and Gaj [CG09], pages 1–17.

[Lan10] Adam Langley. ImperialViolet: Checking that functions are constant time with Valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010.

[RQaPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. CC 2016, pages 110–120. ACM, 2016.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In *CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, 2015.

[WOM11] Carolyn Whitnall, Elisabeth Oswald, and Luke Mather. An Exploration of the Kolmogorov-Smirnov Test as a Competitor to Mutual Information Analysis. In *CARDIS*, volume 7079 of *LNCS*, pages 234–251. Springer, 2011.