Extended Abstract: Constant-Time Analysis for Well-Known Cryptography Libraries

Xiaohan Fu, Shuyi Ni, Siyuan Yu, Zirui Wang, Siwei Liu
{x5fu,sni,s7yu,ziw007,sil013}@eng.ucsd.edu
Computer Science & Engineering, University of California, San Diego

Abstract

This paper presents a toolset for assessing whether a cryptographic function is constant-time on two given (distinct) input sets. The toolset, based on dudect [17], extends new support to languages other than C or C++, including Python 3, Golang, and JavaScript. We present a comprehensive test suite covering common cryptographic primitives and providing finely-chosen inputs that can cause non-constant-time behavior of certain implementations. We then evaluate our approach by applying this test suite to a set of libraries with our toolset and observing the violations that are detected. Our preliminary results suggest that language runtime features such as JIT may break the constant-time requirement.

Keywords: Timing attack, Constant-time implementations

1 Introduction

Timing attacks on cryptosystems have long been researched since 1996 [14]. Various attacks, such as [4], and defenses, such as [12], over widely used cryptography algorithms have been developed and applied to well-known libraries, such as OpenSSL. Tools [1–3, 7, 8, 15, 17, 18] have been developed for analyzing the time-constancy of a crypto-function implemented in C or C++. The time-constancy of cryptographic libraries implemented in other languages remains an understudied area. Consequently, applications and websites based on these libraries may suffer from potential security vulnerabilities.

In this paper, we aim to detect whether popular cryptographic libraries in Golang, JavaScript and Python 3 have constant-time implementations. Our contributions are:

1. A constant-time assessment tool for Golang, JavaScript and Python 3 evolved from dudect[17].
2. A test suite against common cryptography primitives.
3. Presentation and discussion of the results of evaluating the official *Crypto* library (and its extension) of Golang, *PyCrypto*, *Cryptography*, and *PyCryptodome* of Python 3, and the official *Crypto* library of Node.js.

2 Tool Design

Our tool is evolved from dudect [17], designed to assess whether a C function runs in constant time or not. Dudect takes two inputs and runs the function many times for each input to see if the running time for these two inputs shows a statistically significant difference. A difference indicates that there might be a timing exploit and the tested function is very likely to be not constant-time. Note that a detected leakage only shows that the measured running time may not be constant. There is no guarantee that such a leakage is sufficient to conclude the actual presence of a working exploit or that the passed functions/libraries are definitely constant-time.

We modified dudect to make it more statistically reliable, test-extensible, and user-friendly, with changes including:

1. A provision for choosing whether to perform the cryptography function's state initialization every time for each measurement, or once for the whole measurement. This is now customizable by the user, while dudect only supports one state initialization for the whole run. 2. An estimate of the adequate sample size as a threshold for Welch's test on each sample, using the method described in [11]. It compares the difference in size and variance between two populations instead of an arbitrarily set threshold as used in dudect. 3. A more user-friendly API to facilitate writing test suites.

2.1 Workflow

In general, our approach was to measure the execution time of a cryptography function against two different classes of inputs or states. Then we check statistically if these two classes have different timing distributions.

Step 1: Class Definition. First, the user specifies the characteristics of the two classes to be compared by defining two functions: one for initialization and one for inputs generation. The first function initializes the state for the target function and returns a closure function which does the actual computation based on the state initialized. The tool only measures the execution time of the returned closure function to avoid disturbances from key generation and object instantiation. The second function defines two classes of inputs (bound to their class id) as arrays in advance. These inputs are to be fed into the computing function.

Step 2: Measurement. The tool takes in the above two functions, executes the computing function with the prepared inputs one by one, and records the execution time. The highest-resolution timer available on the platform is used to measure the execution time. For Golang with GOOS=x86, we are using the cycle counters in TSC registers. For Node.js, we are using the native performance API. For Python 3, we use the `perf_counter` function provided by the time library.

Step 3: Data Pre-processing. Since timing distributions might be positively skewed, it can be helpful to crop the full data against certain percentile levels. We compute 100 levels of percentiles, from 50% to 99.9995%, and obtain 100 samples in addition to the original one. We also apply higher-order pre-processing, namely centered product [6], to imitate higher-order DPA attacks as claimed by dudect.

Step 4: Statistical Test. The last step is to apply a statistical test on samples of large enough sizes to see if the mean of the two populations within one sample are fundamentally different. Welch's test, which is used to test the null hypothesis that two populations have equal means, is very suitable here as the two populations are of unequal variance and unequal sample size[19]. The test will output a t-value, representing the confidence to reject the null hypothesis (of equal means). As suggested by [10], a t-value larger than 4.5 can probably reject the Null hypothesis; a t-value larger than 100, gives very strong evidence to reject the Null hypothesis.

3 Test Vectors

We designed a general test suite which covers most commonly available cryptographic primitives to apply to our target libraries. The general approach is to vary certain secret inputs with the other inputs fixed to see if the varied input affects the overall execution time. Specially crafted inputs [9, 10] are also added to the test suite. For brevity, the full test suite is listed in the appendix.

4 Results and Analysis

We apply our test suite over each primitive for three trials and output the trial that yields the highest t-value. For brevity, we list only the results of primitives having large t-values i.e. >10 in Table 1; full results are listed in Appendix B, C, and D for Golang, Javascript, and Python respectively.

Table 1. Results of Failing Primitives in Golang and JS.

Language	Primitive	Test	t-value
Golang	RSA OAEP	3	519
JavaScript	RSA PKCS#1v1.5	2	166.93
JavaScript	DSA	2	465.60

4.1 Golang

All the other tested primitives show no violation of time-constancy except RSA OAEP in test-3 (constant vs. varying random key pairs). To verify this, we added a special test: we randomly generated 100 key pairs, and fixed class 0 and class 1 to use the keys in each respective pair to encrypt a randomly generated fixed plaintext. Measurement of three trials revealed that among the 100 randomly generated key pairs, 36%, 42%, and 46% of them showed definitive non-equal execution time. This gives further evidence that different keys may lead to different execution times, which demonstrates that the RSA OAEP implementation here is not constant-time.

Looking into the source code of RSA OAEP encryption, we find that the left-padding process is dependent on secret inputs and conducted only when the computed ciphertext is smaller than the size of the key. Another non-constant-time point lies in the modular exponentiation algorithm implemented in `big.Int` package which is used in the core computations of RSA. The code deploys the square and multiply algorithm [13] which is known to be vulnerable under timing attack with Montgomery reduction [16] or sliding window conditionally. We note that the authors declare this piece of code to be not constant-time in the comments.

4.2 JavaScript

As shown in Table 1, we observe that test-2 (constant key and varying special messages) for RSA and DSA signature reports non-constant-time behavior. This is interesting because this package is just a wrapper of the OpenSSL implementations in c which are shown to be likely to be constant-time by dudect [5]. We observer a trend of decreasing t-values in increasing rounds of measurement after looking deeper into this. We suspect that this is very likely to be caused by the V8 JIT interpreter, which may break the constant-time property of the underlying OpenSSL implementations.

4.3 Python3

Many of the tested primitives show t-value >10, including Chacha20 and Salsa20 which are designed to be constant-time. (We omit results of Python in Table 1 for brevity.) We also observe that multiple trials of the same test on Python output t-values with huge variance even after turning off the garbage collection. As this is not found in Golang and JavaScript, we strongly suspect that this is caused by certain language runtime behavior of CPython.

5 Conclusions and Future Work

We create a tool, evolved from *dudect*, in Python3, Golang and JavaScript to evaluate constant-time behavior of functions. We show the effectiveness of this tool by catching some violations with a specifically-designed test suite.

Future work will extend these preliminary investigations by 1. expanding the test vectors with more implementation specific inputs as discussed in [9, 10] 2. taking hardware optimization into consideration in measurements 3. delving into the impacts of runtime/JIT of Python and JS on constant-time implementations.

Acknowledgments

We are particularly grateful to Deian Stefan, Gabe Fierro, Peter Zelchenko, Dezhi Hong, Liran Xiao and Yihao Liu for providing us feedback, directions and help.

References

[1] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 40–53, New York, NY, USA, 2000. Association for Computing Machinery.

[2] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[3] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, July 2013.

[4] Daniel J. Bernstein. Cache-timing attacks on aes. 2005.

[5] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: A dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[7] Laboratório de Compiladores. FLOW TRACKER – A Flow Analysis to Discover Side Channels. http://cuda.dcc.ufmg.br/flowtracker/example.html.

[8] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C., August 2013. USENIX Association.

[9] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance. 2011.

[10] Josh Jaffe and Pankaj Rohatgi. Efficient sidechannel testing for public key algorithms: Rsa case study 2. introduction. 2011.

[11] Show-Li Jan and Gwowen Shieh. Optimal sample sizes for welch's test under various allocation and cost considerations. *Behavior Research Methods*, 43(4):1014–1022, 2011.

[12] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 1–17, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

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

[14] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, page 104–113, Berlin, Heidelberg, 1996. Springer-Verlag.

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

[16] Peter L. Montgomery. Modular multiplication without trial division. 1985.

[17] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1697–1702, 2017.

[18] TrustInSoft. TIS-CT. https://trust-in-soft.com/tis-ct/.

[19] B. L. WELCH. THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. *Biometrika*, 34(1-2):28–35, 01 1947.

A Full Test Vectors

Note for each primitive, test-i will represent the test on set-0 vs. set-i i.e. set-0 is the baseline case in the comparison.

A.1 Symmetric Cipher

The test vectors designed for symmetric ciphers are listed in Table 2. Typical examples of symmetric ciphers are chacha20, salsa20, AES in various modes.

Table 2. Test suites for Symmetric Ciphers.

Set	Note
0. constant key, plaintext/ciphertext, nonce/iv	baseline case, all arguments randomly prepicked
1. constant plaintext, nonce/iv (same as set 0), varying random key	keys should be randomly picked in advance, must have large population
2. constant plaintext, nonce/iv (same as set 0), const special key: 0, 1, 2, 3	can use smaller population for each special key, but more trials is need to test against different plaintexts
3. constant key, nonce/iv, varying plaintext	
4. constant key, nonce/iv, constant special plaintext: 0, 1, and plaintext whose ciphertext is 0, 1	
5. constant key, plaintext, varying nonce/iv	
6. constant key, plaintext, constant special nonce/iv: 0, 1	

At the very beginning of each test trial, we draw all arguments for the cipher randomly from the whole range e.g. $[0, 2^128 - 1]$ for set-0 as the baseline case. In set-1, 3, and 5, we reuse the other arguments the same as fixed set-0, and draw a number of messages, nonces, and keys respectively from the corresponding ranges uniformly. The number of random numbers drawn here is chosen as a factor of the total measurement number e.g. 1/3 of the total. Note the total number of measurements here must be large enough. The rationale behind test-1, test-3, and test-5 are straightforward: we want to see if random values vs. fixed value for each input of symmetric cipher would reveal any obvious violations. [10] points out that special ciphertext as small integers e.g. 0 and 1 may have impacts on the encryption of AES and RSA which leads to our test-2. We reuse the other arguments, and pick the plaintext to be those encrypted to 0/1 and 0/1 themselves. Test-6 (set-6) and test-4 follows a similar logic here. As we have only a limited number of special inputs in these tests, the total measurements can be relatively small. There exists more specific test vectors designed particularly for certain primitives available like discussed on AES in [9] which could be added into test-2 and test-4.

A.2 Asymmetric Cipher

The test vectors designed for symmetric ciphers are listed in Table 3. Typical examples here are RSA OAEP and RSA PKCS#1v1.5. The test design rationale majorly comes from [10].

Table 3. Test suites for Asymmetric Ciphers.

Set	Note
0. constant key, plaintext/ciphertext	baseline parameter prepicked
1. constant key, various plaintext	
2. constant key, speical plaintext: 0, 1, and plaintexts whose ciphertext is 0, 1	
3. constant plaintext, various key	keys should be randomly picked in advance, must use large population

A.3 Signature

The test vectors designed for signatures are listed in Table 4. Typical examples are RSA PKCS#1v1.5, RSA PSS, DSA and ECDSA. The design logic for test-1 and test-2 is self-contained.

Table 4. Test suites for Signatures.

Set	Note
0. constant key, plaintext/ciphertext	baseline parameter prepicked
1. constant key, various plaintext	
2. constant plaintext, various key	keys should be randomly picked in advance, must use large population (e.g. 1billion population, 10k keys)

A.4 Hash Function

The test vectors designed for hash functions are listed in Table 5. Typical examples are SHA2 families and SHA3 families. The design logic for test-1 is trivial.

Table 5. Test suites for Hash functions.

Set	Note
0. constant data	baseline, data randomly prepicked (at various length level)
1. varying data	

A.5 MAC

The test vectors designed for MAC are listed in Table 6. Typical examples are HMAC and Poly1305. The design logic for test-1 is straightforward.

Note for HMAC, the hash functions chosen should be consistent with the tested hash functions such that the varying data part is already covered in tests for hash functions. (Might need varying data test for Poly1305.)

B Full Results for Golang *Crypto*

The full test results for Golang are presented in Table 7, 8, 9, 10 respectively for symmetric ciphers, asymmetric ciphers, signature and hash/mac functions.

Table 6. Test suites for HMAC functions.

Set	Note
0. constant key	
1. varying key	keys should be randomly picked in advance, must use large population (e.g. 1billion population, 10k keys)

Table 7. Results for Symmetric Ciphers in Golang *Crypto*.

Test	Salsa20	AES-CBC	AES-CFB	AES-OFB	AES-CTR	AES-GCM
1	3.56	1.46	1.50	2.48	1.48	1.85
2	0.63	1.77	1.67	2.87	1.63	2.02
3	1.63	1.36	2.00	1.01	1.40	1.68
4	2.67	2.29	2.08	1.97	2.55	2.40
5	2.68	2.46	3.23	1.80	3.22	1.89
6	4.03	3.23	2.77	1.80	3.96	1.92

Table 8. Results for Asymmetric Ciphers in Golang *Crypto*.

Test	1	2	3
RSA OAEP	1.83	2.18	519.59

Table 9. Results for Signature in Golang *Crypto*.

Test	ECDSA-NISTP256	ECDSA-NISTP384
1	2.07	1.67
2	3.60	2.40

Table 10. Results for Hash Functions and MAC in Golang *Crypto*.

Test	SHA256	SHA3-256	HMAC-SHA256	HMAC-SHA3-256	Poly1305
1	1.91	2.06	N.A.	N.A.	N.A.
1	N.A.	N.A.	2.26	1.75	2.28

C Full Results for JavaScript *Crypto*

The full test results for JavaScript are presented in Table 11, 12, 13, 14 respectively for symmetric ciphers, signature, hash functions and mac functions.

Table 11. Results for Symmetric Ciphers in JavaScript *Crypto*.

Test	AES-CBC	AES-CFB	AES-CTR	AES-GCM	AES-OFB	Chacha20
1	2.36	3.18	2.23	2.93	2.64	2.82
2	2.07	3.21	2.48	2.07	2.33	2.75
3	2.42	2.54	2.25	1.60	1.92	2.42
4	3.21	4.20	3.60	4.80	3.72	2.14
5	3.60	3.57	2.20	2.68	5.54	3.57
6	3.51	3.38	3.51	3.36	3.53	4.19

Table 12. Results for Signatures in JavaScript *Crypto*.

Test	RSA PKCS#1v1.5	DSA	ECDSA
1	1.17	1.56	2.14
2	166.93	465.60	3.37

Table 13. Results for Hash Functions in JavaScript *Crypto*.

Test	SHA256	SHA3-256
1	1.91	1.98

Table 14. Results for HMAC in JavaScript *Crypto*.

Test	HMAC-SHA256	HMAC-SHA3-256
1	2.34	2.65

D Full Results for Python3 *PyCrypto*, *Cyptography*, and *PyCryptodome*

The full test results for Python3 are presented in Table 15 for all three tested libraries.

Table 15. Results for Python3 *PyCrypto*, *Cryptography*, and *PyCryptodome*.

Test	0	1	2	3	4	5	6
Cryptography-AES-CBC	10.0	18.4	10.8	6.2	10.5	10.2	15.8
PyCrypto-AES-CBC	6.0	13.9	7.5	2.1	3.8	21.4	53.2
PyCryptodome-AES-CBC	4.0	17.1	9.7	5.9	5.7	17.6	5.8
Cryptography-AES-CFB	10.3	29.1	10.6	8.5	9.6	25.9	10.8
PyCrypto-AES-CFB	4.2	10.8	4.4	7.6	3.1	19.8	16.2
PyCryptodome-AES-CFB	6.9	23.9	9.7	6.9	5.3	29.1	8.0
Cryptography-AES-OFB	5.5	39.2	6.1	7.8	8.1	48.4	17.6
PyCrypto-AES-OFB	6.9	27.1	36.4	5.2	7.5	25.0	8.1
PyCryptodome-AES-OFB	5.1	11.2	5.9	4.2	6.4	14.1	7.8
PyCryptodome-AES-CTR	6.8	12.1	6.5	8.4	7.9	11.4	6.2
PyCryptodome-AES-CCM	7.1	6.1	6.8	6.0	6.1	8.9	7.1
PyCryptodome-AES-EAX	3.0	6.5	15.1	1.9	2.9	11.1	2.9
Cryptography-AES-GCM	11.3	32.7	20.4	16.0	17.5	43.9	15.1
PyCryptodome-AES-GCM	3.2	4.3	7.7	7.7	9.2	11.2	8.5
PyCryptodome-AES-OCB	7.6	21.2	7.9	7.9	7.8	34.4	8.4
PyCryptodome-ChaCha20	14.3	20.7	23.7	16.9	19.6	22.1	18.4
Cryptography-ChaCha20	5.5	25.5	24.5	8.3	9.8	29.3	124.2
PyCryptodome-ChaCha20	13.8	23.4	16.4	14.8	17.7	20.4	19.6
PyCryptodome-Salsa20	12.4	24.6	8.7	2.3	9.7	25.1	19.4
Cryptography-RSA	2.0	1.9	1.9	56.5			
PyCrypto-RSA	1.9	3.8	2.2	111.2			
Cryptography-DSA	2.0	2.8	2.1	193.3			
Cryptography-ECDSA	1.8	2.5	1.4	3.0			
Cryptography-SHA256	7.7	6.9	7.0				
PyCrypto-SHA256	3.1	3.7	3.3				
PyCryptodome-SHA256	3.6	3.3	2.6				
Cryptography-SHA3-256	3.6	3.7	4.4				
PyCryptodome-SHA3-256	5.0	2.2	3.6				
Cryptography-HMAC	5.8	18.9	14.6				
PyCrypto-HMAC	5.5	12.7	9.5				
PyCryptodome-HMAC	4.2	10.9	3.6				
Cryptography-POLY1305	3.3	11.2	10.6				
PyCryptodome-POLY1305	17.5	13.9	16.4				