Toolchain for Timing Leakage Analysis of NIST Lightweight Crypto Candidates

Adam Blatchley Hansen* Eske Hoy Nielsen Morten Eskildsen

September 14, 2020

Abstract. With recent advances in IoT technology and lightweight devices, an ever increasing number of highly constrained systems now communicate over networks. Modern cryptographic algorithms are often poorly suited to the limitations of such devices, which has led to NIST publishing a call for algorithms to be considered for new lightweight standards.

As such devices are often deployed into adversarial environments, and may not have the luxury of large caches, or hardware support for cryptographic primitives, proper side channel resistance is an important property for any new standard. We have collected a set of side channel analysis tools, and used them to evaluate all 32 candidates in the second round of the standardization process.

We provide the results of running our toolchain on all reference implementations, and show some of the timing leakages and design patterns we discovered, and discuss the strengths and weaknesses of the various tools.

We have compiled our toolchain into an easy to use Docker image targeting the competition API, which we have made available for candidates to use for development purposes for the rest of the competition. Our pipeline is available at GitHub[1].

*blatchley@cs.au.dk

[1] `https://github.com/blatchley/Timing-Analysis-Pipeline`

1 Introduction

This work documents our toolchain for conducting timing-leakage based analysis on software implementations of cryptographic algorithms. Specifically, we focus on candidates in the NIST Lightweight Cryptography Standardization Process (LWC) [CSR20b], as candidates for this process are required to submit a C implementation of a cryptographic primitive with a common header. In addition to cryptographic-security requirements, the submission guidelines state the following for side-channel security [CSR18, Section 3.5]:

> *The implementations of the AEAD algorithms and the optional hash function algorithms should lend themselves to countermeasures against various side-channel attacks, including timing attacks, simple and differential power analysis (SPA/DPA), and simple and differential electromagnetic analysis (SEMA/DEMA).*

We analyse all the submissions for constant time variations, Utilizing various static and dynamic analysis tools such as *dudect* [Rep17], *FlowTracker* and *ctgrind* [Lan10], we show that several of the implementations contain variable time code. We note that these are reference implementations, and are not yet required to be constant time, but may be in the future.

In this report we provide our complete toolchain in an easy to use Docker-image, we show examples of timing leakages and design patterns detected by our pipeline in various implementations, and we reflect on the strengths and weaknesses of the tools we have used. As the standardization process moves forward the focus shifts to more side-channel resistant implementations, we hope our efforts in selecting and implementing these tools with easy to use wrapper scripts will allow authors to more easily integrate timing leakage testing into their development process.

1.1 Side Channel Security

Timing leakage which is dependent on secret data can be devastating to the security of a protocol. The variation does not have to be very big, as the attacker might be able to remove noise by rerunning the process and averaging over the results. If done a sufficient amount of times, it might be able to clearly identify the differences. [Por19]

Our combination of static, dynamic and fuzzing based tools helps detect many different sources of potential timing leakage.

Variable Time Instructions. Depending on the underlying hardware and architecture some instructions may take shorter time depending on the input. This is true for the notorious `DIV` assembly instruction on all AMD and Intel architectures, where smaller divisors yields faster execution. Similarly for many other instructions like `MUL`, `UMULL`, `FCOS` etc. [Sch16]. Even shifts can be variable time on some architectures [Por19].

Branches and Conditional Jumps Conditional jumps/branching on secrets is inherently bad. Even if the exact same amount of work happens in both branches, branch prediction might be utilized to leak the secret causing the branching [AKS06]. As such, branching on secret values should be avoided altogether.

Lookup-based S-boxes Another example of potentially variable time behavior found in some implementations of modern encryption schemes like AES is the use of substitution-boxes (S-boxes) implemented using lookup tables. They inject a non-linear operation into the cipher, and are carefully designed to thwart modern techniques such as such as linear and differential cryptanalysis.

A full lookup table does not necessarily fit into the fastest CPU cache (L1/L2 cache) registers, so depending on the index, it might need to load the data which is much slower. If the index is based upon a secret value / key, then this delay can be used to deduce information about the secret value [Tez19].

Furthermore, several studies have shown that real world timing attacks on S-boxes are indeed possible. This has typically been done by analyzing AES implementations in OpenSSL [Ber05,CRMM19]. These results include demonstrations of cache-level timing attacks carried out over a network, as well as examples where even when the full S-box is in the cache, it can be evicted or otherwise retrieved in variable time. Ashokkumar et al. targeted an S-box measuring 256 bytes, which corresponded to 4 cache lines of each 64 bytes. They examine version `1.0.2p` of OpenSSL and the attacker scenario requires them to be using the same CPU core as the victim process. They use a *cache access*-driven attack, where they just need to know which cache line was accessed. As there are four cache lines, an access reveals two bits of the key used for indexing. The remaining bits refer to the column of the cache line. They utilize a spy program with 200 threads and constantly flushing/evicting the cache, thus resulting in cache misses. This cause the encryption/decryption to be interrupted around 160 times, hence giving the spy program the possibility to inspect the cache access. By knowing around 50 blocks of plaintext for encryption or the ciphertext for decryption and then observing cache access in round 2, they are able to deduce the key with a success rate of 90%.

A key takeaway we see here is that cache-timing attacks on lookup based S-boxes are a very real security concern, as there exist realistic security models where the adversary can have very fine grained knowledge as to the contents of the cache during execution.

1.2 Related Work

At the time of the standardization process for AES, side-channel analysis was not as well understood as it is today. There were demands stating that implementations should be side-channel resistant, but NIST themselves commented that a table lookup implementation of AES was not vulnerable. However, as discussed above, it was later proved by e.g. Bernstein in [Ber05] that it is indeed vulnerable. Bernstein used cache-based attacks to conduct his attack which has since inspired many similar attacks and techniques.

There have been some larger surveys of recent developments in lightweight cryptographic algorithms [EKP+07], however these have focused more on benchmarks and design discussion, rather than the side-channel security of actual implementations.

An approach of reproducing timing behaviour execution can be found in [Che14]. Here the authors construct a time-deterministic replay mechanism for replaying the execution of a real program. They end up being 1.85% within the timing of a real execution. This includes studying cache state, preemptions, processor scheduling etc. The paper look into detecting covert side-channels, but the technique is interesting as it might be possible to detect variable time behaviour execution of a program.

In [CSY16] the authors aim at *detecting* timing attacks. Here they rely on performance counters provided by the underlying micro architecture and kernel-support for the detection. They show that it is indeed feasible to execute the cache-based attacks, and that it requires solid understanding for detecting exploitation from other processes. Hence, it is better to prevent the attacks in the first place.

2 Toolchain

In order to be able to detect the various kinds of variable behaviour described in section 1.1 we need different approaches. Thus, our toolchain consists of several tools. Namely *FlowTracker*, *dudect* and *ctgrind*. Although they all perform variable time analysis, they use different techniques for detecting it. In some areas they overlap while they complement in others.

2.1 dudect

In the paper "*dudect: Dude is my code constant time?*" [RBV17] the authors construct a very simple tool to measure the execution time of a whole program. It does so by measuring the amount of CPU cycles executed during program execution. This yields a very fine-grained timer. However, like all other applications it is affected by the OS, exact workload at the moment of execution and other sources of noise. To combat these external sources *dudect* runs several million times and performs a statistical test on the data after removing outliers. As *dudect* is running a statistical analysis and not a static analysis it does not have any assumptions about the underlying hardware or timing of microcode instructions. However, it cannot tell us exactly where any non-constant time execution happens, as it treats the program as a black box.

A strong argument for using this tool is the simplicity. To observe different behaviour one would supply different input. *dudect* takes care of generating and inputting random values to be used with a black box approach. If knowledge about special cases are known, then it can be easily incorporated. *dudect* uses the concept of *input classes*. These correspond to different types of input and special cases, such as a range of all-zeros or some similar edge-case values. The

source code for *dudect* has to be included when compiling, but it is merely 300 lines of C code.

Based upon all the different executions, *dudect* will continuously for every one million execution output whether or not it believes the application is constant time or not. It can obviously never conclude that the application is constant time, but it can conclude that it is not.

2.2 FlowTracker

FlowTracker uses static analysis on the source code to detect timing vulnerabilities [RQaPA16]. It does so by integrating into the LLVM compiler. During code generation FlowTracker injects an additional compilation pass whichs works on the program's intermediate representation (IR). By analysing the Static Single Assignment (SSA) form it can produce a sparse graph of the information flow in the target program. Generally two types of time-based information leaks are detected.

1. **Secret data affecting program execution**. E.g. when code blocks are executed only if some condition involving secret data is fulfilled. This is similar to section 1.1 about branches and conditional jumps.
2. **Memory indexed by secret data**. Again, similar to section 1.1 where table lookups, S-boxes and alike can leak.

FlowTracker aims at running as late as possible in the compilation process. The argument for this is that even though code might yield a constant time implementation in the current compiler version, later compiler versions might result in variable time behaviour. By running after the main compilation phases, new compiler optimizations will be detected by FlowTracker. However, FlowTracker does not inspect the final architecture-dependent instructions, which could end up causing variable time execution even if the IR had no indication of it.

A limitation of FlowTracker's design is that any dynamically shared libraries are not considered in the analysis. Simple yet wrong utilizations of C-functions like `memcmp()` can cause variable time behaviour. However, as this function resides in a dynamically linked library, FlowTracker does not directly detect improper usage of it.

Further limitations of Flowtracker is that it does not analyse C code directly but instead analyses LLVM, in many cases this is an advantage as we analyse code that is closer to the final form. In our case this is a slight disadvantage since NIST specified that GCC 5.4.0 is the target compiler for this competition. GCC does not use LLVM and therefore we had to stick with clang. The disadvantage here is that different compilers might reverse countermeasures and introduce branches instead, which could circumvent efforts to make code constant time.

2.3 Ctgrind

ctgrind uses dynamic analysis to evaluate timing vulnerabilities in compiled code [Lan10]. It is very similar to TIMECOP, which also uses *valgrind* to detect timing dependence on secret data, which has recently been added to the

SUPERCOP platform. *ctgrind* assumes that code is being run on processors with constant time instructions and thereby only focus on timing vulnerabilities where branches taken or memory accessed depends on secret inputs. It does so by keeping track of which bits in memory and which CPU registers contain secret data while executing the program.

Memcheck in *valgrind* already does a very similar thing to this, it keeps track of memory addresses and registers that contain uninitialized data and where it is used in branching or memory access. Therefore *ctgrind* uses a patched version of *valgrind* where the memcheck tool has been modified to not track uninitialized data but secret data, *valgrind* knows what secret data is by intercepting calls to `ct_poison` and `ct_unpoison` functions defined in a *ctgrind* shared library.

The main disadvantage of using *ctgrind* is that it is a dynamic analysis and therefore it will only find vulnerabilities on lines of code that are executed with the given input. Since the tool is not coverage-guided there might be inputs that cover more code than others, and we cannot reasonably check all valid inputs. Therefore *ctgrind* cannot promise to find all branch and memory access vulnerabilities in a program, however as long as *valgrind* doesn't have bugs, *ctgrind* will find all branch and memory access vulnerabilities in the code covered by the given input.

Advantages of *ctgrind* incluide its detailed output and that it does not rely on timing. *ctgrind* will report all problems of code covered along with line numbers making it easy to identify where the problems are in the code. Since execution time is not considered by *ctgrind* it will report any problem no matter how big or small a timing difference that code introduces. Other advantages of *ctgrind* is the speed, due to cryptographic functions often having large code coverage no matter the input, therefore only few executions of the function are typically necessary for experimental evaluations.

3 Evaluation of Implementations

3.1 Method

We ran our pipeline on all the 32 candidates[CSR20b] and their different implementations. Among the implementations are a reference implementation, which is the overall implementation and not tailored for any specific architecture, as well as multiple implementations targeting different security levels such as 128- and 256-bit keys. Some of the candidates had implementations for ARM which we have skipped due to our own hardware constrains. In total we had 89 reference implementations, and 115 compilable implementations if non-reference implementations are included.

3.2 Findings

Table 1 shows the candidates for which our tools flagged one or more of the reference implementations as having potential side-channel leakage. Note that

just because a candidate was flagged in this table does not necessarily mean that there exists a viable side-channel attack on the implementation. In the following section we will look at some common patterns which caused implementations to be flagged, before taking a deeper dive on some specific implementations in section 4.

In all, *dudect* found timing leakages in some or all of the implementations for 8 candidates. *ctgrind* flagged 14 candidates, including all of the implementations flagged by *dudect*, and *FlowTracker* flagged 11 candidates, including 5 which were not flagged by *dudect* or *ctgrind*.

We also note that three of the candidates, Gimli, Grain-128 and Subterranean 2.0, submitted implementations which did not exactly follow the API specified by NIST. As such, these required us to perform minor tweaks for our tooling to work. These minor tweaks included renaming files and including the header file specifying the AEAD API. Because the header file only contains function definitions and no code, this cannot influence the results of the tools.

3.3 Detections

Just because an implementation was flagged by one of our tools does not necessarily mean it contains side-channel vulnerabilities. In order to evaluate the potential sources of leakage, we reviewed the source code for each of the flagged candidates to try and identify why each candidate was flagged, and whether the detected behaviour constitutes a security risk.

Among the flagged implementations, we found several groups of candidates which were being flagged for very similar pieces of code relating to indexing into S-boxes, as well as several candidates which used similar reference block cipher implementations.

Finally, two of the candidates which were detected by our initial *dudect* scans, DryGascon and Comet, had their own interesting timing leakages. These are discussed in Section 4.

3.4 Constant-Time S-box lookups

S-boxes are a key component of modern substitution-permutation ciphers, and are often represented in code as table lookups. As discussed in Section 1.1, This can be an issue e.g. when you access a table using an index which is based on secret data. If an adversary could measure which data is loaded into the cache at the point when the S-box is used, it could leak information about the secret data. An example of code naively updating state using an S-box can be seen in Figure 1.

Performing a full cryptanalysis for each set of S-boxes is outside the scope of this project, however we note some important differences in the types of S-boxes used by the underlying block ciphers in Section 4.3.

MixFeed and SAEAES candidates, as well as some of the implementations of ESTATE all use variations of the AES blockcipher as an underlying primitive. These were all flagged by both *dudect* and *ctgrind*.

Candidate	**dudect**	**ctgrind**	**FlowTracker**	**Notes**
ACE	○●	○●		
ASCON	○●	○●	○●	
COMET			○●	
DryGASCON	◑●			
Elephant	◑●	◑●	○●	ctgrind finds more than dudect
ESTATE	◑●			
ForkAE	◑●			
GIFT-COFB	○●	○●	○●	
Gimli	○●	○●		NIST format not followed
Grain-128AEAD	○●	○●	○●	NIST format not followed
HYENA	○●		○●	
ISAP	○●	○●	○●	
KNOT	○●	○●		
LOTUS	○●			
mixFeed				
ORANGE				
Oribatida	○●	○●	○●	
PHOTON-Beetle	○●		○●	Also provided bitsliced asm files
Pyjamask	○●	○●	○●	
Romulus	○●		○●	
SAEAES			○●	
Saturnin	○●	○●	○●	
SKINNY	○●		○●	
SPARKLE	○●	○●	○●	
SPIX	○●	○●		
SpoC	○●	○●		
Spook	○●	○●	○●	
Subterranean 2.0	○●	○●	○●	NIST format not followed
SUNDAE-GIFT	○●	○●	○●	
TinyJambu	○●	○●	○●	
WAGE	○●		○●	
Xoodyak	○●	○●	○●	

Table 1. Reference implementations and the tools that flagged them as being variable time. Marker ● refers to issues in all implementations, ◑● translates to only a subset of the implementations being flagged. Finally, ○● refers to no implementations being flagged.

```
const unsigned char sbox[16] = {12,6,9,0,1,10,2,11,3,8,5,13,4,14,7,15};
//.../
void SubCell(unsigned char state[4][4]){
    int i,j;
    for(i = 0; i < 4; i++)
        for(j = 0; j <  4; j++)
            state[i][j] = sbox[state[i][j]];
}
```

Fig. 1. Substitution step in the ForkAE implementation, using a 4 bit S-box

ForkAE, Romulus, LOTUS and HYENA candidates, as well as the non-AES ESTATE implementations were also all flagged by *ctgrind* due to S-box indexing, with one of the ForkAE implementations also triggering *dudect* These implementations used reference implementations of the GIFT and SKINNY (ForkAE, Romulus) tweakable block ciphers.

The remaining candidates detected by *ctgrind*, were all again flagged for various indexing into lookup-based S-boxes. We also note that beside the reference implementation, PHOTON-Beetle also provided a bitsliced ASM implementation which fixed the leakages.

3.5 FlowTracker

The *FlowTracker* results did confirm most of the *dudect* and *ctgrind* results, however it also flagged 5 candidates that neither *dudect* or *ctgrind* flagged. SPIX, SpoC, ACE, KNOT and Gimli were all flagged only by *FlowTracker*. Both ACE, SPIX and SpoC have been flagged with very similar lines of code that seem to be false positives. An example of the flagged code can be seen in Figure 2. This is a false positive since there is no branching and memory access is not indexed by secret data. Instead we have a for loop that always runs a constant number of iterations with memory look up on constant indexes. Therefore there is no variation in running time in the flagged code.

We were unable to see if KNOT and Gimli were also false-positives as it was less obvious what those code snippets did, and in general it can be hard to verify whether a result is a false positive or not.

```
const unsigned char rate_bytes256[8] = {8,9,10,11,24,25,26,27};
(...)
for ( i = 0; i < 8; i++ )
    state[rate_bytes256[i]]^=k[i];
```

Fig. 2. One of the SPIX lines flagged by FlowTracker

4 Discussion

In this section look at two candidates which displayed timing leakages other than lookup based S-boxes, namely the DryGascon and COMET implementations, as well as taking a closer look at the problem of indexing into S-Boxes using secret data.

We decided to look deeper into DryGascon as this was the candidate that showed the highest timing dependency on secret data in the *dudect* report. We also found DryGascon interesting as *dudect* only observed timing dependency in the 256-bit implementation and not in the 128-bit implementation despite having almost identical codebase. COMET was also interesting as it showed high timing variation and all implementations were flagged.

4.1 DryGASCON

DryGASCON [Rio19] showed some quite clear timing problems. On the 256 bit implementations *dudect* returned almost immediately concluding that it was definitely not constant time. Testing involved 1.46 million runs.

We identified a timing leakage based on the input key, where small keys would be repeated then scrambled using the `CoreRound` function, with this process repeated a number of times dependent on the contents of the key. We reached out to the author who confirmed this was a source of timing leakage, but believed that it would not be exploitable in practice due to how little information it leaks. For more details on the flagged code, see Appendix A.

4.2 COMET

COMET was one of the non AES candidates that was most heavily flagged. Both *dudect* and *ctgrind* reported all 4 implementations as variable time, while *FlowTracker* did not report problems in any of the implementations.

dudect marked all four implementations as "definitely not constant time" within 5-20 million executions. Based on this we investigated further with *ctgrind* which showed the same single line of offending code in all four implementation, with an additional 15 lines being flagged in the AES implementations due to S-boxes.

```
if(Z_[p-1] & 0x80){        /*10000000*/
        Z[0] ^= 0x1B;      /*00011011*/
}
```

Fig. 3. Variable time code in COMET found by *ctgrind*.

Figure 3 shows the offending line of code found using *ctgrind*. The problem is that there is a conditional statement branching on the state variable of their block cipher, which is initialised using the secret key and the nonce.

For more details about the COMET timing leakage, as well as a bitsliced patch we developed which fixed the timing leakage, see Appendix B

We reached out to the authors about what level of side channel security they claimed. They responded, highlighting the fact that the code they provided on the NIST website was only reference code, and was only intended to help readers understand the specification and generate KAT's (Known Answer Tests,) and was not designed to be constant time or to optimise it's runtime. They also noted that if COMET moves into the next round, they will address these concerns.

4.3 Underlying Block-Ciphers

The implementations leverage a number of reference block ciphers, from AES to SKINNY and GIFT.

AES is an interesting choice, because while it is a widely accepted and well analysed block cipher, it is also notoriously difficult to efficiently implement in constant time or in low-memory environments without a large falloff in performance.[Por18][Ber05] This issue has been partially solved by the introduction of native AES-NI opcodes present in recent x86 CPU, allowing for fast, small, constant time AES implementations. However, specifically in this competition the entire purpose is to find cryptographic schemes for lightweight and IoT devices, which includes low memory or power limited environments, and environments which likely won't support these new x86 opcodes. We also note that the potential of timing issues in AES is not merely academic, as multiple timing attacks have been demonstrated on real world implementations of AES.

Conversely, SKINNY and GIFT represent newer block ciphers, developed specifically for lightweight environments.They use a range of techniques to improve over AES, including the option for 64 bit block sizes, (with 4 bit S-boxes,) a very lightweight key scheduler, sparser diffusion layers, with the S-boxes and permutation layers being designed such that they can be implemented using only a very low number of bitwise AND/NOR/XOR gates. As such, we expect candidates using these ciphers to be more easily converted into efficient constant time implementations.

5 Pipeline Implementation

The goal of this project was initially to analyse NIST Lightweight Crypto Standardization Process candidates and provide an easy to use codebase for hooking into various constant time analysis tools. *dudect*, *ctgrind* and *FlowTracker* are all tools that come from research projects and thus have not been maintained since their original release. While implementing the pipeline it quickly became apparent that a compilation of tools for installing locally would not be very useful to the crypto community on its own as some of these tools are old, badly documented, relied on outdated dependencies, had missing download links and were hard to install. We therefore decided to provide the pipeline along with a Docker image with all tools installed and ready to use.

The entire Docker image is open-sourced and we also provide a ready built image that can be downloaded and used within minutes.
`https://github.com/blatchley/Timing-Analysis-Pipeline`

5.1 Technical Details

We decided to base the Docker image on Ubuntu 16.04 and use GCC 5.4.0 as our main compiler. We choose this in accordance with the NIST Lightweight Crypto Specification [CSR18], as this is the platform candidates will be evaluted on. The NIST AEAS API specifies both a `crypto_aead_encrypt` and a `crypto_aead_decrypt` function. Our pipeline supports analysing both functions for variable time dependence on the key. The pipeline provides a settings file where the different tools and analysis can be configured on some parameters such as function to analyse, max analyse time, message size and sample size.

dudect

dudect is provided as in the original release as it does not rely on any other tools. It is compiled with GCC 5.4.0. For interfacing with the tool we provide C code that sets up nonce, authenticated data, plaintext, ciphertext and messages for the candidate encrypt/decrypt function. These variables are constructed at random at startup in accordance with the `api.h` file. This file is part of the NIST API format and specifies length of different input variables. After variable setup, the code specifies input classes for *dudect* to generate keys from. Since the pipeline needs to work for many crypto candidates we cannot assume anything about the candidate and must treat it as a black box. Therefore the two input classes are selected as uniformly random input and fixed value input. This means that half the selected keys will be a fixed value while the other half will be uniformly randomly chosen. The number of executions in each *dudect* iteration can be configured by the user along with a max allowed time for analysis. Standard is 1,000,000 executions per iteration.

dudect will run until it is certain enough that the code is variable time, therefore it might run forever if a timeout is not set. The tool provides very limited output and will only output the number of iterations run, statistical values for each iteration, and a decision of either "maybe constant time", "probably not constant time" or "definitely not constant time".

ctgrind

ctgrind was released 10 years ago and relies on valgrind 3.5.0, as of writing the newest version is 3.16.1. In the pipeline *ctgrind* is provided in a patched version that provides support for valgrind 3.16.1. Using 3.16.1 over 3.5.0 gives better accuracy as valgrind has had many improvements and bugfixes over the past 10 years, and is still being maintained.

For *ctgrind* we also provide C code for integrating the candidates code with *ctgrind*, here we again generate input variables in accordance with the api.h file provided and generate a uniformly random key. The position of the key in memory is then marked as secret using the *ctgrind* poison and unpoison functions provided by the *ctgrind* shared library. After poisoning the key memory addresses we execute the encrypt/decrypt function and unpoison the key after that. Since

ctgrind is a dynamic analysis we repeat this process with uniformly random data a number of times to increase the chance of getting full code coverage by the dynamic analysis. The number of random executions can be configured by the user.

ctgrind provides a more detailed output and will log each time that secret data was used to determine a branch or used to index into memory. In the output *ctgrind* will give you the total number of times this happened during the execution and a stacktrace of each unique context this happened in. Thus it is easy to find the exact line numbers in code where variable time code happens.

FlowTracker

FlowTracker relies on a patched version of LLVM/clang 3.7.0 and is provided with LLVM/clang 3.7.1 in the Docker image. We were unable to provide *FlowTracker* with the up to date LLVM/clang 10.0.1 due to large build system and codebase changes incompatible with *FlowTracker*.

Since *FlowTracker* is a static analysis and not a dynamic tool we do not need much custom code to integrate the candidate code with the tool. To make the tool analyse their code we need to be able to compile their code to LLVM - we use clang for this. Additionally we need to provide XML files for both encrypt and decrypt files specifying names of the functions to analyse, secret data and input variables.

FlowTracker provides detailed output consisting of a full graph of the program analysed along with subgraphs of the code parts that give rise to variable time problems, this makes it easy to identify the lines of code where there might be problems.

Scripts In addition to the tools we provide our pipeline script that will load settings, compile the candidate code with all three tools and execute them. The Docker image provided is setup so that when the image is used the script will be executed automatically. The pipeline script reads the source code from a source directory on host mounted to the Docker container and will similarly write output to an additional mounted output directory. This allows running multiple candidates or just a single implementation if wanted. In addition to the run script we also provide a script for downloading and unpacking all candidates from the NIST website.

5.2 Pipeline Discussion

This pipeline mostly serves as a wrapper for easily running code on many tools, thus the overall quality of the report we give is dependent on the quality of the tools. As discussed in Section 6 the results should not be definitive answers or guaranties. None of the tools are perfect and they should therefore be viewed as tools to aid in finding figuring out much timing dependency there is on secret data and where in the code it comes from.

The pipeline does have its limitations, currently the input code has to follow NISTs AEAD scheme specification, thus we do not support analysing arbitrary functions. Additionally we only support analysing timing dependency on the

key. Some AEAD schemes use secret nonces, however NIST have specified that in the Standardization Process only public nonces are allowed.

Due to the NIST specification we only support implementations written in C, though in future it should be possible to allow custom compile option and thus be able to analyse anything that compiles to assembly with *dudect* and *ctgrind*.

The upsides of using Docker is that we can provide an easy to use build process that we are sure works for everyone no matter what system they run on, or how far into the future they are. We can also provide a ready built image enabling users to start analysing code in few minutes. The downside of using Docker is speed, running inside a container can be slightly slower, and decouples from the actual machine it is running on. This is not always great as the instruction set and machine can have an effect on code being variable time. However both *ctgrind* and *FlowTracker* are not impacted by this as they do not care about cache and instruction set. For *dudect* we cannot completely exclude the possibility that running inside Docker could affect the results, however since *dudect* compares relative running time on different input and performs statistical tests there should be no significant difference between running on host and inside Docker.

For authors not familiar with docker our GitHub repository contains instructions on how to execute in the context of Docker, as it is isolated from the host operation system.

6 Evaluation of Tools

Referring to table 1 we saw that all the tools managed to identify some variable time implementations. However, the quality of the findings varied.

dudect is very simple to setup and run in the black box approach. Actual runtime of an instruction is part of the measurement instead of assuming instructions are constant time. The blackbox nature of the tool also removes any dependencies on the implementation language, and can handle hand crafted assembly just as well as C or C++ code. However, *dudect* did not identify as many things as the other tools, and it was clear from several of the candidates that it was only able to detect the variable time behaviour in some of the implementations even though the other implementations also contained the same timing leakage. An example of this was the DryGASCON 128-bit vs 256-bit implementations. In general, just because *dudect* has not found anything yet does not mean it will not in the future, as is is bounded by the amount of time we allow it to run.

After identifying that an implementation is definitely not constant time, the nature of *dudect* prevents it from saying exactly where the constant behaviour originates from. Thus, *dudect* is fine for identification of a problem, but not good at pinpointing it. The greatest strength might be the identification of the severity. If *dudect* is able to detect variable time behaviour, then it is not a false-positive and something worth looking more into.

ctgrind uses the dynamic approach mentioned earlier, but while *dudect* needed many invocations to detect the variable time behaviour, *ctgrind* only needed a few. This method succeeds at correctly identifying and pinpointing the exact lines in code which caused the problems, making it much easier to find the source of the leakage. We did not see any direct false-positive identifications.

When including noise and false-positives *ctgrind* seemed to yield the best results of the three tools utilized. As *ctgrind* have not had any official development in 10 years, its setup can be a little complex (see section 5.1). This was one of the motivations for us to create the docker container. Finally, as it is based on Valgrind, it is also bound to any coding errors in Valgrind.

We also note that with the recent introduction of TIMECOP to the SUPERCOP platform, the inclusion of *ctgrind* in the pipeline gives an easy way for developers to easily and quickly predict how TIMECOP will respond to their code, without having to enter it into the SUPERCOP framework.

FlowTracker uses pure static analysis. It did confirm most of the findings from both *dudect* and *ctgrind*. Additionally it also flagged at least 3 false-positives, see Figure 2. It seems like the engine behind the analysis contains some patterns not corresponding to real variable time problems, such as the example shown in section 3.5.

Contrary to *ctgrind*, *FlowTracker* does not have to find inputs which trigger every possible branch of the code to get full coverage. In the Lightweight Cryptography Project this does not seem to be a problem, as there are generally few branches, as all inputs and keys are valid. Especially when compared to, for example, the Post-Quantum Standardization Process, where many candidates have much more complex requirements for the input format and validation, which inevitably yields more branching [CSR20a]. Thus, for the purposes of the LWC Standardization process, we assume both *FlowTracker* and *ctgrind* achieve near full code coverage.

All together we benefited the most from *ctgrind* during our analysis. The results were more precise with less false-positives and pinpointed the location in the code. *dudect* was good at detecting the severe variable time parts with no false-positives, but did not identify near as much as the other two tools.

7 Conclusion

We have created a powerful automated pipeline that makes it significantly easier to start analysing for non-constant time behaviour. Instead of taking hours to setup and compile these tools by ones self, or having to push code to online platforms like SUPERCOP, they are now packed in a ready-to-use Docker container.

We evaluated the quality of the included tools *FlowTracker*, *dudect* and *ctgrind* in the context of the NIST Lightweight Crypto Standardization Process by running them on round 2 reference implementations. Here they found multiple variable time sections in several of the candidates.

Comparing the different tools revealed *ctgrind* to be the most precise in our context, though we expect *FlowTracker* to perform relatively better when the context involves code and implementations with more branching. It is not sufficient to analyse source code for variable time behaviour using just one of the tools. Each of them has their pros and cons, but most importantly they use different approaches for the identification. This resulting in different findings and understanding of severity.

7.1 Acknowledgements

This report is based off the results of a 5-week project for the Language Based Security Graduate course at Aarhus University.

We would like to thank Associate Professor Diego F. Aranha of Aarhus University for his guidance on modern side-channel security, and technical assistance with initial compilation of some of the more fragile tooling.

References

AKS06. Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, pages 225–242, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

Aum19. Jean-Philippe Aumasson. *Cryptocoding*, 2019.

Ber05. Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

Che14. Chen. Detecting covert timing channels with time-deterministic replay. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 541–554, USA, 2014. USENIX Association.

CRMM19. Ashokkumar C., Bholanath Roy, Bhargav Sri Venkatesh Mandarapu, and Bernard Menezes. "s-box" implementation of aes is not side channel resistant. *Journal of Hardware and Systems Security*, 12 2019.

CSR18. NIST CSRC. *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process*, 2018.

CSR20a. NIST CSRC. *Post-Quantum Cryptography*, 2020. Accessed on May 28, 2020.

CSR20b. NIST CSRC. *Lightweight Cryptography*, May 6, 2020. Accessed on May 21, 2020.

CSY16. Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49, 09 2016.

EKP+07. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design Test of Computers*, 24(6):522–533, 2007.

GJN19. Shay Gueron, Ashwin Jha, and Mridul Nandi. Comet: Counter mode encryption with authentication tag. 2019.

Lan10. Adam Langley. *Checking that functions are constant time with Valgrind*, 2010.

Por18. Thomas Pornin. Bearss:, constant time crypto. Technical report, 2018. Accessed on May 27, 2020.
Por19. Thomas Pornin. *Why Constant-Time Crypto?*, 2019. Accessed on May 25, 2020.
RBV17. 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.
Rep17. Oscar Reparaz. *dudect: dude, is my code constant time?*, 2017.
Rio19. Sébastien Riou. *DryGASCON*, 2019.
RQaPA16. Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 110–120, New York, NY, USA, 2016. Association for Computing Machinery.
Sch16. Peter Schwabe. *Timing Attacks and Countermeasures*, June 10, 2016.
Tez19. Cihangir Tezcan. Distinguishers for reduced round ascon, drygascon, and shamash permutations. 2019.

A DryGascon

We decided to look at DryGASCON [Rio19] as it involved some quite clear timing problems. On the 256-bit implementations *dudect* returned almost immediately concluding that it was definitely not constant time. Testing involved 1.46 million runs.

We identified a timing leakage based on the input key, where small keys would be repeated then scrambled using the `CoreRound` function, with this process repeated a number of times dependent on the contents of the key.

On the 128-bit reference implementation we let *dudect* run for 1 hour with more than 375 million executions only with the inconclusive result "maybe constant time", while both *ctgrind* and *FlowTracker* correctly flagged all implementations as being variable time with respect to the key. We suspect this was due to the timing difference was not big enough with the smaller key size.

DryGascon uses the ASCON construction, however we noted that it was not the S-boxes being flagged, as those were already implemented in constant time using bitslicing. In both the 128- and 256-bit versions DryGASCON allows different key sizes, this is processed in the DRYSPONGE_set_key function. Depending on the size, different actions are taking to expand the key. For the small key size, the key it is repeated and scrambled.

The DrySponge constructions needs as input a state (c, x) where c is 72 bytes and x is 16 bytes in the 256-bit version. To ensure an unique encryption each pair of 4 bytes in x must be unique. If x contains a repeated sequence of 4 bytes, then the encryption algorithm might fail according to the author. Hence, to ensure it works, the input is repeatedly scrambled for as long as there is a repeated sequence in x. See how this is done in Figure 4.

From Figure 4 it can be seen, that if we have a key which after a single call to `CoreRound` have a repeated sequence, then it will start looping and the timing will increase. Upon comparing it 1 million times with a all random key

it on average took 900 CPU cycles for the initiation + set_key execution - not the remaining encryption. With an all zero key, it took 1400 CPU cycles. This is definitely possibly to measure and distinguish.

```
for(unsigned int i=0;i<DRYSPONGE_CAPACITYSIZE;i++){
    ctx->c[i] = key[i%DRYSPONGE_KEYSIZE];
}

// ... SNIPPET ...

DRYSPONGE_CoreRound(ctx,0);

unsigned int modified=1;
while(modified){
    modified=0;
    for(unsigned int i=0;i<DRYSPONGE_XSIZE32-1;i++){
        for(unsigned int j=i+1;j<DRYSPONGE_XSIZE32;j++){
            uint32_t ci,cj;
            DRYSPONGE_load32(&ci,ctx->c+i*sizeof(uint32_t));
            DRYSPONGE_load32(&cj,ctx->c+j*sizeof(uint32_t));
            if(ci==cj){
                DRYSPONGE_CoreRound(ctx,0);
                modified=1;
                break;
            }
        }
        if(modified) break;
    }
}
memcpy(ctx->x,ctx->c,DRYSPONGE_XSIZE);
memcpy(ctx->c,key,DRYSPONGE_XSIZE);
```

Fig. 4. Small keys are modified according to size. At first key is repeated, then scrambled in the CoreRound function and finally we reiterate and scramble as long as any words are identical

Upon contacting the author of DryGASCON, Sebastien Riou, he confirms the timing leakage due to the comparison for equality of the words for x but does not think it is exploitable in practice because it leaks too little information.

One way to exploit this timing behaviour would be if an adversary controlled a word of the key. This way, he could probe different encryptions and measure the time taking to detect if words in the key were similar. Then it would require him to find all the collisions for the words. However, similarity still has to be taken after the first `CoreRound`, which makes the inferring a bit harder. Furthermore, this represents a very powerful attacker model is quite powerful and not very likely.

B COMET

COMET which stands for "Counter Mode Encryption with authentication Tag" is an authenticated encryption with associated data, it is based on a special mode of operation for block ciphers [GJN19]. They provide 4 different implementations. Two of the implementations are based on the CHAM block cipher family, one is based on the Speck block chipher family while the last is based on AES.

dudect marked all four implementations as "definitely not constant time" within 5-20 million executions. Based on this we investigated further with *ctgrind* which showed the same single line of offending code in all four implementation, with an additional 15 lines of AES related problems in the AES based implementation. The AES related problems found will not be discussed further here but are the same as those discussed in Sections 3.4 and 4.3. *FlowTracker* showed no problems in COMET, however this is probably due to COMET implementations using a lot of C's *memcpy* function which *FlowTracker* is not great at modeling.

```
if(Z_[p-1] & 0x80){        /*10000000*/
        Z[0] ^= 0x1B;      /*00011011*/
}
```

Fig. 5. Variable time code in COMET found by ctgrind

```
u8 a = Z[0]^0x1B;
u8 b = Z[0];
u8 bit = Z_[p-1] & 0x80;
    u8 mask = (bit | -bit) >> (sizeof(u8) * CHAR_BIT - 1);
    u8 ret = mask & (b^a);
Z[0] = ret ^ b;
```

Fig. 6. Constant time version of Figure 3.

Figure 5 shows the offending line of code found using *ctgrind*. The problem is that they have a if statement branching on the state variable of their block cipher, thus the program execution time will vary depending on the content of the state variable. This means that if we can somehow learn whether this line was executed we learn 1 bit of the state. *ctgrind* reported this line from 4 different context meaning that there is a potential leak of 4 bits, if those 4 bits can be related to the key in some way an attack could potentially reduce the search space to $\frac{1}{16}$ of the original search space.

Digging deeper into the code we found that the offending line of code is executed in the setup phase before the real encryption starts, and that the state is

initialized as ($Z = 0^{|k|}||key \oplus nonce$) and since the nonce is publicly available an attacker might be able to leak up to 4 bits of the key. Other stuff does happen between initialization and the offending line of code, so an attacker would have to be able to reverse that to leak part of the key.

We implemented a constant time version of COMET by substituting the conditional statement with the code seen in figure 6, using bitslicing[Aum19]. The idea of the code is to replace the conditional statement with something that wont get translated into jump instructions on the CPU. We do this by always always computing `Z[0]` $\oplus$ `0x1B` and then use constant time bitwise operations to assign `Z[0]` if `Z_[p-1] & 0x80` is nonzero. We tested the new implementation with *dudect* , *ctgrind* and *FlowTracker* and they all reported that the code was constant time. To be sure that this fix would not increase the running time too much we validated correctness and execution time of the if statement and the new code. We found that both pieces of code produced the exact same output and that the old code on average used 59,7 clock cycles, while the new code used 50,1 clock cycles.

The specification of COMET did not clearly specify their security claim for side-channel security so we contacted the author to verify if they claim side-channel security and notify them about the patch we created. They responded, highlighting the fact that the code they provided on the NIST website was only reference code, and was only intended to help readers understand the spec and generate KAT's, and was not designed to be constant time or to optimise it's runtime. They also noted that if COMET moves into the next round, they will address these concerns.