Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

A compiler-based technique for automated analysis and protection against side-channel attacks

Relatore: Prof. Alessandro BARENGHI
Correlatore: Prof. Gerardo PELOSI

Tesi di Laurea di:
Stefano SANFILIPPO Matr. 815996

Anno Accademico 2015–2016

To the one person who got me through these dire years.

Author's thanks

I would like to thank my advisor Gerardo Pelosi, and my coadvisor Alessandro Barenghi for their invaluable help in understanding and researching the topic presented in this work. However, their support was definitely not limited to my thesis. Over the last few years, they have guided me through the elaborate landscapes of computer science. They made passionate about cryptography, compilers, and more in general about scientific research. And for this, I am deeply grateful to them.

Michele Scandale, who in the meantime got his PhD from Politecnico di Milano, deserves the biggest thank you one could piece together, for putting up with my incessant questions and issues while diving into the guts of LLVM, one of the most fascinating and complex achievement of software and compiler engineering.

Finally, I would like to thank all the other people whose direct or indirect input helped me navigate around all the theoretical and practical obstacles and, of course, to those who made me smile during the inevitable bad days.

Prefazione

In questa tesi introduco un metodo per valutare la sicurezza di un'implementazione software di un cifrario simmetrico contro varie tipologie di attacchi a side-channel e predisporre appropriate contromisure senza intervento umano.

Gli attacchi a side-channel sfruttano debolezze nella struttura interna e, più comunemente, nell'implementazione di un algoritmo di cifratura. Lo strumento di analisi qui descritto è in grado di localizzare queste debolezze a livello di singole istruzioni ed applicare contromisure laddove necessarie, consentendo di proteggere l'implementazione in esame da una vasta gamma di attacchi. L'applicazione selettiva delle contromisure consente inoltre di contenere l'inevitabile calo di performance senza compromettere la solidità del prodotto finale.

Lo strumento di analisi è altresì in grado di produrre visualizzazioni delle proprietà di sicurezza a livello di singolo bit, offrendo dunque una accurata rappresentazione delle caratteristiche dei vari algoritmi presi in considerazione. I grafici prodotti hanno confermato alcune debolezze ben note, come nel caso di Misty1, e messo in evidenza proprietà interessanti, come nel caso di Serpent.

Sono stati analizzati tutti gli algoritmi nel portfolio ISO/IEC 18033-3, inclusi AES (Rjindael), DES/TDEA e Camellia, assieme ad altri meno noti ma comunque interessanti come XTEA, Speck e Simon – tutti algoritmi che puntano alle piattaforme meno potenti, come quelle embedded che si stanno sempre più diffondendo nelle applicazioni Internet of Things.

Assieme alla descrizione dello strumento di analisi e dei metodi di protezione adottati, ho anche composto una vista generale sulle tipologie note di attacchi a side-channel, sulla loro formalizzazione e sulle relative contromisure, valutando le opzioni disponibili in ciascuno scenario considerato.

Preface

In thesis I introduce a method to analyse and improve the security of symmetric ciphers software implementations against various classes of side-channel attacks. These are attacks that exploit weaknesses in the internal structure and, more commonly, in the implementation of the algorithms. A large fraction of these weaknesses can be revealed and automatically eliminated by my solution. The analysis can locate the weak spots at single instruction level, and selectively apply countermeasures, resulting in a minimum run time overhead while preserving the soundness of the results.

The tool here presented is also able to produce heatmap visualisations of the security properties at bit level, offering a solid insight into the properties of the examined algorithms. These visualisations confirmed some well-known flaws, such as in the case of Misty1, and revealed interesting properties of others – like in the case of Serpent, which has been found to have very good diffusion properties, resulting in highly efficient protected implementations.

All the ciphers in the ISO/IEC 18033-3 portfolio have been implemented, including AES (Rijndael), DES/TDEA and Camellia, and then some other less common choices that cater to low-capacity devices, e.g. in the realm of Internet of Things, such as XTEA, Speck and Simon. Alongside with the description of the analysis and countermeasure application algorithms and tools, I also provide a survey of the main side-channel attack strategies and relative countermeasures, evaluating the best options in each scenario that was considered.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

In the era of the Internet and global communications, embedded cryptographic devices are pervasive in many aspects of our daily life, from electronic payments and ticket cards, to all sorts of increasingly smarter devices, like gas meters or automotive computers. However different and seemingly unrelated, all devices have a common ground: they are designed to encrypt and authenticate communications using a key stored in their internal memory, without ever revealing it.

This is the kind of application where an attacker may successfully impersonate the device only by knowing the key, and if this key is well buried inside a tamper-proof device, there is no hope of posing as the legitimate user. Embedded cryptography surfaced in the 1990s, and since its inception it has greatly improved the security and reliability of crucial technologies. The technology improvement was for everybody, not only for some top-secret military project: thanks to it, phone SIMs and chip-based credit cards are now (at least, should be) impossible to clone.

Unfortunately, as someone once said, attacks only get better. And never a person was more correct: by the end of the century, an entirely new field was born, *side-channel attacks*. This innovative family of techniques exploits pieces of information involuntarily given away by a computing device, such as fluctuations in the power consumption or electromagnetic emissions during the computation. This secondary, unintended, flow of data was aptly named the side-channel.

Combining laboratory measurements with increasingly more refined statistical tools allowed to recover the key in many practical instances, and even to reverse-engineer hardware encryption devices whose internal structure was classified, as it happened for NSA's own Skipjack. This is a proof of how powerful the technique is.

The introduction of side-channel attacks was so successful because it took an entire discipline by surprise. Until that moment, the security of a cipher was evaluated in a purely theoretical framework, which only considered the algorithm and not the machine that implemented it. The fact that a piece of hardware was to execute an implementation of the algorithm was seen as a practical detail. As we found out, these seemingly boring implementation details soon became the source of many issues. When side-channel attacks are involved, what is being calculated is not everything: *how* a certain value is obtained becomes crucial. The order of the instructions, their timing, the specific way an elementary operation is implemented, all potentially leak information.

Countermeasures soon started to be developed. Those, paired with new design criteria that finally took the implementation into account, together with the constant improvements in the manufacturing technology, contributed to thwart the most common attacks. At the beginning of the 2000s, a theoretical framework was published that allowed to formally evaluate the soundness and privacy of a certain *implementation* against the most prominent class of attacks at the time. This was a huge step forward, which allowed to establish formal security proofs for the countermeasures, and to compare the various alternatives under a unified model. Some of these countermeasures relied on specially designed logic styles to implement the digital circuit, while many others simply involved modifications to the microarchitectural and architectural levels of the device. This last category proved very successful because it allowed to reuse widely deployed implementation techniques, and can be extended to software implementations. While some of these countermeasures exploited the properties of a specific cipher, others contributed generic building blocks, allowing to secure any encryption algorithm.

Until recently, these countermeasures had to be applied manually, by inspecting either the circuit network or the software implementation and identifying the weak points. The latter case meant reading the assembly code and replacing some of them with secure transformations, taking into account the ordering of the operations and data dependencies. Evidently this operation has to be repeated for each single software implementation and platform. As the size of the design space grows, both in terms of available ciphers and available platforms, the widespread application of manual methods quickly become unpractical. In this scenario, the analysis and transformation framework of modern optimising compilers can be successfully employed to automate the analysis, identify vulnerable instructions and apply appropriate countermeasures.

The field of automated analysis and protection is still very young, with the first published studies dating back to the early 2010s, and my work aims to add an hopefully non-zero contribution to it. Automated solutions offers great flexibility when it comes to progressive improvements of the code, since they eliminate most of the burden that previously laid on the software author or auditor. In addition to that, countermeasures may be handled with algorithms that guarantee the soundness of the result, no matter what the complexity of the codebase is. The precision of the analysis also allows to limit the countermeasures to where they are effectively needed, reducing the performance hit inevitably introduced by the transformations.

The content of this thesis is articulated in three chapters, plus an appendix. In the first one, I introduce and detail the specific family of attacks my work aims at preventing, the underlying theoretical model and the countermeasures I have adopted. The second chapter will instead discuss the inner workings of the solution I studied and the details of the tools I developed for this purpose. In order to demonstrate the feasibility of the techniques I describe, I analysed and protected the implementations of several standardised and some novel symmetric block ciphers. The third chapter presents and discuss the results of the time and code size benchmarks I conducted under two popular embedded architectures, ARMv7 and MIPS32, and different security parameters.

The analysis tool is also able to produce visualisations of the internal properties of the various algorithms analysed, which proved valuable for exploring the design space of symmetric ciphers and offered meaningful insights into the distinctive features of different internal structures, giving further substance to interpret the benchmark results.

Chapter 1

State of the Art

This chapter contains a survey of the theoretical framework underlying my work. The presentation is organised around five main topics, which cover the attack model, its formalisation and one specific family of countermeasures whose choice will be motivated further. An additional section will provide details about the analysis technique which is at the core of my tool.

The first two sections examine side-channel attacks, a broad range of techniques which comprises several attack vectors and analysis methods, that can nevertheless be grouped into two high-level categories: *passive* -- when an attacker limits himself to observe properties of the circuit or program execution, and *active* -- where the attacker manipulates the computation and evaluates the effects. The third section formalises said attacks using the theoretical framework introduced by Ishai, Sahai and Wagner, in which a proof of security can be given for circuits and software routines. The fourth section will provide insights about masking countermeasure against side-channel attacks, a particularly flexible and well studied one. The fifth and last section will detail the dataflow analysis framework.

1.1 Passive side-channel attacks

A symmetric cipher (or any cryptographic algorithm, for what matters) can be seen as a black box, with some inputs, a *plaintext* and a *key*, and some outputs, the *ciphertext* in our case. When operating the cipher, the information flows from the input to the output, forming a channel that we will call the *main channel*. In the last few decades, various design techniques have been introduced, making it

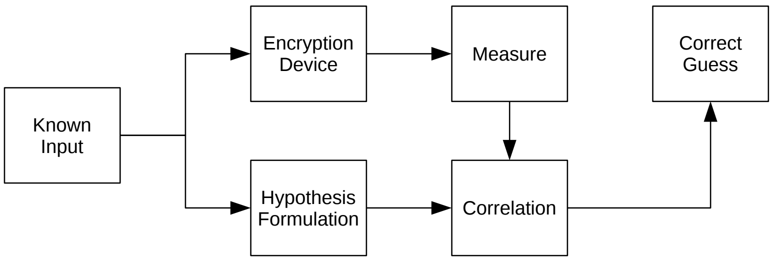


Figure 1.1: Phases of a power side-channel attack

impossible for the attacker to extract information about the input from the output, i.e. by observing the main channel.

All these techniques are based on the assumption that this main channel is the only path information flows through. In practice, though, ciphers have hardware or software implementations which leak additional information about the computation being performed. For instance, the power consumption of a specific gate could be correlated to a particular form of the operands, or the time a certain operation takes might have a dependency on the input. This extra flow is called the *side channel*. All the attacks described in this section process measurements taken from the device, but do not interfere in any way with the computation, hence the “passive” term. This is in contrast with “active” techniques, which disturb the circuit and gather information from its reaction to the faults introduced. Most of the attacks described in the following sections require physical access to the computing device, although some remote side-channel attacks have been reported in literature, e.g. timing attacks against web services implementing cryptographic ciphers [13, 16, 20].

There is a long record in literature of attacks that exploit different side channels, ranging from time, to power consumption [48,49], electromagnetic [4], photon [74] and sound emissions [34]. Power analysis is a common one, since measurements can be taken in a minimally invasive way with standard laboratory equipment.

1.1.1 Power analysis

At a macroscopic level, a microprocessor is composed of functional units, each of them handling a specific task. The dispatch of a memory load instead of e.g. an addition, will activate different units at different times, with an effect on the power consumption of the device. At the microscopic level, each semiconductor logic gate in the microprocessor is built of transistors. Because of their physical structure, a current flows through them during each state transition, determining again a variation in the power consumption of the device. A theoretical modelisation of the power consumption of a computing device can be found in [19].

Although these variations are minimal and not directly observable, they are detectable using statistical techniques. In both case, the power consumption of the device over time can be linked to the operations being performed, and for this reason it can be exploited as a side-channel. Even if these classes of attacks were first targeted at power measurements and still retains the fact in their name, the same methodologies described have been successfully applied to electric, magnetic, acoustic and thermal probes. The variants based on electro-magnetic emissions are very common as well, and have been dubbed SEMA *(simple electro-magnetic analysis)* and DEMA *(differential electro-magnetic analysis)* [4,33,70].

The power analysis attacks described in this section assume the implementation is a white box, as the internal structure of the circuit must be known to some extent in order to formulate a prediction model for their power consumption under given hypothesis. Two common models for power variations are the Hamming weight of an intermediate result, which accounts for the energy needed to set the wires carrying the value from an initial state all-up or all-down, and the Hamming distance between two different values, which models the behaviour of a switching memory element, e.g. a D-latch.

The logical phases of a power attack are shown in Figure 1.1. Starting from a known input, the target device is attached to an oscilloscope or similar, which records a power profile during the execution. In parallel, the attacker makes hypothesis on the internal state and uses a theoretical model to produce the expected value for the quantity under measure. The oscilloscope data is then compared with this expected value, leading to either confirming or discarding the set of hypothesis. By trial and error, the attacker gains knowledge about the internal state of the device. The precise nature of the input values, hypothesis and comparison methodology depends on the specific attack, as it will be detailed in the rest of this section.

Collecting traces The first step in any power analysis is collecting one or more *traces* from the device. A trace is a sequence of measurements over time, in this case of power consumption while the cipher runs. These measurement could be taken, for instance, by placing a resistor on the ground line and measuring the voltage drop at its terminals. To increase the precision of the measures, a stabilised bench power supply might also be used in place of the one embedded in the circuit, or the decoupling capacitors might be removed. A sinusoidal clock might also be used to reduce the interference of high-frequency harmonics [49]. Sometimes, having the device operate at the edge of its operational envelope might also be useful to decrease the effectiveness of attack countermeasures, e.g. by a controlled lowering of the input voltage, as described in [49].

Simple power analysis (SPA) Simple power analysis involves a direct interpretation of the power consumption measurements collected during cryptographic operations [48]. SPA techniques exploits major variations in the power consumption that are apparent from the traces, and therefore are not able to work around noisy captures. However, the problem of noise can be reduced by techniques as averaging multiple traces instant by instant and performing SPA on the result.

The central point in SPA is making inferring aspects of the inner working of the circuit being probed by looking at the trace. For instance, a circuit might perform some memory read/write operations at the end of each round of a symmetric cipher implementation, producing a pattern which is different from the one of the round computation. The number of rounds could reveal the exact algorithm implemented by the circuit. Sometimes, the pattern can directly reveal the secret input, for instance if the computation flow depends on parts of it. This happens in the case of naive square-and-multiply RSA implementations, where the encryption operation loops over each bit in the private key (exponent), performing a different operation in case of 0 and 1.

A more refined SPA technique is trace pair analysis [49], based on comparing multiple traces to identify differences, which could identify the points where the computation takes different paths, depending e.g. on the input. In general, the source of SPA leaks are visible deviations in the computation, caused by conditional branches and operations with variable timing depending on the input. Even in the case of fixed-time operations and branch-less code, differences in a microprocessor microcode might be revealed by the analysis. More high-level events such as context switches and interrupts might be exploited as well, although they tend to yield poorer results, because of their inconsistent timing.

Other SPA-based techniques include collision attacks, which involves determining pairs of inputs which result in the same internal state, and algebraic attacks, which may lower the number of bits required to perform a successful key extraction, such as the lattice-based attacks, such as the one against DSA [53, 63] described by Howgrave-Graham et al. in [40].

Although fairly simple, SPA is practical only in the presence of significant dependencies of the control flow on the input, which cause some instructions to be skipped. This condition seldom occurs in real-world implementations of block ciphers, and are often hidden by noise. In these cases, and when there is no apparent way to interpret discrepancies in the traces, DPA techniques might be more appealing, thanks to their noise-canceling properties and a more relaxed guessing model. Other than stand alone, an SPA could also be used as a support for DPA, by pinpointing the time instants in which the process of interest takes place, e.g. marked by a sudden increase in power consumption, thus helping the attacker to narrow the time window to be analysed.

Differential power analysis (DPA) DPA is a statistical method for analysing sets of power measurements in order to identify data-dependent correlations, as stated in [48]. The basic method involves splitting the set of samples for a given instant in time into subsets, then computing the averages of the subsets. If the partition is not correlated to the value being computed, then each subset is just a random sampling from the full distribution, and the differences between averages will tend to zero as the number of traces grows. Instead, if there is a correlation, the differences will approach a non-zero value. DPA can expose even tiny correlations when enough traces are collected, under the assumption that the noise and sampling errors have zero-average distributions that cancel out. However, improvements in the data collection can greatly reduce the time needed to recover the secret by improving the input signal quality, i.e. making the signal-to-noise ratio better.

This analysis is supposed to be performed at a “point of greatest leakage” in time. In practice, determining this point is difficult or impossible without prior analysis. To remove this requirement, the classification and averaging process is performed at various time offsets within the traces. In the case of a binary partition, the DPA result can be plotted so that the X axis is the time offset in the traces, and Y axis the difference in the averages of the two distributions at that point in time. The graph will be mostly flat, but whenever a correlation with the data being processed in that moment exists, a non-zero value will appear, i.e. a spike. More refined statistical tests can also be adopted instead of a difference-of-averages, leading to good results with fewer traces, for instance in the case of CPA introduced by Brier et al. [15], which exploits the Pearson correlation factor.

The function used to file a certain sample under one of the subsets is called a *selection function*, and it determines the information revealed by a DPA. Typically, a selection function is a binary function, thus determining a binary partition, based on a guess about the value of an intermediate value in the computation. More complex multi-bit hypothesis also enable non-binary selection functions, which may speed up the process [49]. The guess can be as simple as the value single bit, e.g. one output of a S-box, or any complex prediction. In any case, good selection functions are crucial in performing a successful DPA, and constitute the “smart” part of any DPA attack.

A DPA works by collecting traces over multiple executions of the same computation. In order to synchronise these traces , the measuring equipment could be triggered by a signal that happens at a well-known point in time, e.g. on the line that gives a “start encrypt” command, if available. Based on the expected characteristics of the traces, analog filters and special probes could be used to increase the quality of the capture. Additionally, a post-processing may be implemented to perform tasks like noise reduction, feature highlighting, or temporal realignment of the collected traces to counterbalance device clock drifts. It could be as easy as finding the time shift d that minimises the differences between $T_0\left[t\right]$ e $T_1\left[t+d\right]$ for two traces T_0 and T_1. In particular, some DPA countermeasures work by misaligning traces through non-uniform time skews in the form of random no-ops, clock skips and non-synchronised clocks. In any case, it should be noted that perfect alignment is not required, although it can significantly speed up the attack.

Once the traces have been collected and prepared, the attack works in three steps:

1. **Prediction and selection function generation** different selection functions are prepared based on different guesses about the same variable, and the set of samples partitioned accordingly.

2. **Testing** usually, the average of each subset is computed. A compression function might be applied to isolate the meaningful points and reduce the computational weight of this step, which already accounts for most of the time spent in the process, together with the capture time.

3. **Evaluation** the results of the previous step are composed and analysed to determine the most likely guess. The process is usually done by visual inspection of an operator. Automated tools have been developed as well that suggest the most likely candidates.

These steps may also be iterated, for instances in attacks against multi-round block ciphers, where a successful key recovery requires attacking the first round and recover the first subkey, then using the results to attack the second round and so on.

Correlation Power Attack (CPA) *Correlation power attacks* (CPA) are a variant of the DPA, proposed by Brier et al. [15], that works by maximising the correlation between the observed distribution of power consumption at a fixed time instants over the collected traces, and the predicted distribution under a certain key hypothesis. Usually, the Pearson correlation test is employed and the attack proceeds by formulating hypothesis and checking which yields the maximum correlation coefficient.

Higher-order DPA (HODPA) The attacks described above rely on the traces collected from a single point in the circuit, for instance the ground line. It is entirely possible to combine the traces collected from t different points in time or space during the computation, using a generalisation of DPA known as *higher order DPA* (HODPA). The order of a HODPA t is equal to the number of probes applied to the circuit, and we speak of t^{th}-order DPA.

A classic application of the HODPA is to bypass masking countermeasures, where a value is split into a set of shares, such that individual probing of them give no usable measurement, but the set of traces for probes $A, B, \ldots$ will show correlation to their $\oplus$-sum. In general, the number of traces necessary to perform a successful HODPA grows exponentially in the order of the attack, meaning that those countermeasures that force an attacker to use higher orders are an effective way to hinder practical attacks by increasing the trace capture time over the threshold of feasibility [75].

1.1.2 Countermeasures

SPA can be easily thwarted in most practical instances, as it suffices to avoid conditional branches based on input-dependent variables and use constant-time operations and algorithms. These conditions naturally occur for most if not all the modern symmetric ciphers which are treated in this discussion, and easily reduce the source of SPA leakage to the point where it is overcome by noise. Preventing a DPA is much more challenging because of its noise-cancelling properties, and a lot of different countermeasures have been studied since the inception of this class of attacks.

The first set of DPA countermeasures include hardware design techniques that minimises data-dependent leakage or decreases the signal-to-noise ratio of the circuit to the point where collecting enough traces to reveal an actionable correlation becomes unpractical [50]. Noise might be introduced in the measurements by adding circuits that perform calculations non correlated with the values of interest to an attacker, or by inserting random variations in timing and execution orders with methods, such as drifting or jittery clock, random no-ops , dummy operations, or random branching between different implementations as used in [60]. These counter measures are not very effective, and signal processing can often eliminate the noise, for instance by detecting a pattern in the injected dummy operations [19]. However, a combinations of them can help thwart most basic attacks by increasing the time needed to collect traces, as demonstrated by Schramm et al. in [75], and requiring (slow) human assistance in detecting potential noise patterns.

A more refined approach consists in splitting the inputs and combining them with random values in a way that require challenging HODPA attacks, like blinding, which introduces random values in computation exploiting known mathematical properties of the algorithm or masking, a countermeasure based on encoding each input into multiple, randomised shares that are processed independently and recombined only at the end.

1.2 Active side-channel attacks

Passive side channels attacks as described in Section 1.1 rely on simple observation (probing) of the circuit, followed by post-processing and analysis. Another class of side-channel attacks exists, which is based on deliberately manipulating a computing device, e.g. by flipping a bit in a register or cutting a wire, and observing the effects or lack of thereof. These are *active* side-channel attacks, also known as *fault attacks*. Various methodologies of active attacks exist, which differ heavily based on the functionality implemented by the circuit under attack, e.g. for symmetric and asymmetric cipher implementations. In the following sections I will privilege the discussion of attacks against symmetric ciphers.

Common fault injections techniques are variations in power supply at the boundaries of the operational envelopes, variations in the external clock to induce a desynchronisation with e.g. memory, variations in temperature to affect the operation, use of light sources or ionising radiations to trigger photoelectric effects that result in a bit flip, high-precision laboratory equipment to cut wires or insert new pads. The resulting fault may be either transient or permanent. In the former case, usually a current is induced in the device that is falsely interpreted as an internal signal. As the ionisation source is removed, the chip goes back to its normal behaviour. On the contrary, a permanent fault introduces a non-reversible modification in the structure of the circuit, such as the destruction of a gate, short-circuiting of two wires or bypassing of a memory element.

Irradiation techniques require the silicon to be exposed, which makes the tampering evident and reduces the chances to be able to reuse the device outside the laboratory setup. This is even more true if the irradiation is used to produce permanent effects. The process reduction in the manufacturing of microchips means that targeting a single gate is becoming harder and harder, and the laboratory equipment is extremely expensive, ranging from thousands to millions of euros. However, such equipment offers very precise control over the faults being induced, which might sensibly speed up an attack, or enable new ones.

Setup-time violations, which include clock and input voltage manipulations, are much cheaper but less localised, as they target the chip as a whole. Their effect on the computation varies from skipping instructions to preventing correct loads and stores. These fluctuations can be triggered at specific instants during the computation, but it should be noted that, with microprocessors becoming faster and faster, the rise time of high-precision power supplies is usually not sufficient to isolate single clock cycles. Thermal variations are also used, with the peculiar ability to disable loads and stores separately. In fact, the operational envelope for the two operations differ, therefore a range of temperatures can be found such that one operation is malfunctioning with a certain probability, but the other is working properly, as mentioned in [49].

1.2.1 Differential fault analysis (DFA)

A differential fault analysis attack against a symmetric cipher works by reconstructing the internal status of a round by observing the differences between pairs of faulty-correct outputs from the same input. Once the relevant portion of the key schedule has been discovered, the attack then moves up, by inverting the last round (whose key is known) and targeting the last but one, up until the whole key has been recovered or the first round has been broken. Because of the diffusion properties of modern ciphers, faults are typically induced close to or in the last round, with the goal of minimising the complexity of the relation with the bits potentially affected. Doing otherwise would diffuse the effect on all bits, providing no useful information to the attacker.

A famous DFA attack is the Piret attack [66] against AES-128, which injects single byte faults in the internal state and can cut down the bruteforcing effort to 2^{40} with a single faulty-correct pair. Additional pairs often narrow down the candidates to few, if not a single one. The attack can be adapted to AES-192 and AES-256, although it requires an additional guessing effort. Provided that faults can be injected with the precision required, the attack is feasible in some minutes of computation on a common desktop with around 200 or 300 faulty-correct pairs. The Shamir attack against DES [12] is similar to the Piret attack, although he also showed that it can be applied to a black-box cipher (as Skipjack was in 1996) to reverse-engineer its internal structure.

1.2.2 Safe-error attacks

Safe-error attacks (SFA) [44] exploit the ability to inject errors that do not alter the result of a computation, on the contrary of a DFA. Because of the inherent structure of block ciphers, any single-bit flip will rapidly diffuse to the entire ciphertext, which means that safe-errors might only happen when a bit is forced to the value it would already have in normal conditions. Therefore, an attacker proceeds by forcing a certain value on a given bit, and observing variations in the output, or the lack thereof. No change means that the bit was set to the correct value, anything else means the guess was incorrect, which for a single bit reveals the value anyway, as the complement of what was forced.

Depending on where the fault is injected, this class of attacks is divided in two families: attacks to the computation (called C-SFA) [82], which target the ALU, and attack to the memory (M-SFA) [81], targeted at memory or registers. The latter is less common because of the higher degree of control needed over the fault location and timing [47].

1.2.3 Countermeasures

Countermeasures against malicious faults are essentially the same employed against accidental faults in high reliability equipment, as shown for instance in [44, 47, 82].

The usual approach is based on redundancy, either spatial, temporal or encoding redundancy. In the case of spatial redundancy, more circuits are employed to compute the same operation multiple times, and a voting mechanism is used to detect discrepancies, e.g. by accepting a result only if at least 2 out of 3 units yield the same. Temporal redundancy works in a similar fashion, but multiple results are sequentially calculated by the same device. Finally, coding redundancy employs error detection or correction codes, such as the Reed-Solomon [80] or Manchester [32] encodings. Once a malfunction has been detected, the circuit can emit random output to prevent the leakage, or enter an emergency mode.

Temporal redundancy is the cheapest, as it does not require special hardware, it just accepts the overhead of re-executing the same computation, either hardware or software. Fault injection is able to fault the majority of the values by inducing multiple times the same failure in the same point in the case of 2-out-of-2 schemes, while 2-out-of-3 and higher are usually more difficult to attack. Fault injection is still possible, but increasingly challenging in that an higher and higher temporal and spatial precision is required to inject a faulty majority, for instance 2 on 3 equally faulty values.

Besides temporal duplication, a symmetric cipher implementation might be protected without additional hardware requirements by decrypting the ciphertext and checking with the original plaintext. This is possible for a lot of ciphers that use the same primitive for both operations, and quite difficult to overcome, as injecting another fault in the decryption routine that results in a correct plaintext is extremely challenging.

A low-impact hardware duplication technique involves Dual Data Rate (DDR) logic [56], which perform an operation both on the rising and falling edges of the clock, resulting in a doubling of the computation. The rising and falling results are stored separately and then compared. This technique has been applied with success to AES implementation, although no results are known on more complex routines, such as asymmetric ciphers.

Error correction are usually optimised for hardware implementations, and the large amount of bitwise operations make software implementations slow, which is why they are usually employed in special circuitry. In both cases, the main challenge is preserving a valid encoding through the non-linear operations of a symmetric cipher, with the usual approach involving a parity code whose value is cached for each possible output of the S-box. Parity codes are not extremely reliable, though, for instance they won't detect an even number flip errors.

In addition to redundancy, hardware watchdogs can be used to detect sudden variations in clock or input voltage, triggering a similar emergency mode. Capacitors and self-sustaining oscillators can also be embedded in the device to remove the dependency on external devices which are very easy to tamper, as the quartz oscillator or the filtering capacitors. These hardware countermeasures prevent most of the low-cost attacks, but imply an increase in the price, and therefore are usually restricted to high security applications.

Finally, the circuit can be somewhat protected from irradiation by adding a reflective coating over the chip, or packages whose opening irreparably breaks the contained chip. An even more sophisticated countermeasure implies running a thin mesh of conductors over the die and running a periodic signal through them. FIBs are highly likely to break the mesh, interrupting the anti-tamper signal and triggering again an emergency mode.

1.3 Masking

Masking is a form of secret-sharing and it is the at the core of the ISW secure transformation discussed in Section 1.4.1. It consists in splitting each value into a collection of n values, also called *shares*, such that at least $m \leq n$ of them are required for reconstructing the original value, and no information (at all!) about the original value can be gathered by observing less than m of them. Usually $m = n$, i.e. all of the shares are required for reconstructing the masked value. A masking scheme is said to be of order n if each value is split in $n+1$ shares.

Goubin and Patarin first mentioned this method in [37, 38], and it was almost contemporarily analysed by Chari et al., who showed the soundness of first-order masking and how it can be extended to higher-orders as in a secret-sharing scheme in [19]. In particular they showed that in a realistic leakage model, the number of acquisitions necessary to recover the key grows exponentially in the masking order. Ishai et al. expanded the work of Chari by proposing the ISW framework for provably secure masking. Rivain and Prouff extended the ISW analysis in [73], proving that the information obtained by observing the entire leakage of an execution can be made negligible in the masking order. All in all, the masking order t can be considered a sound security parameter for the circuit under analysis.

Although masking is generally slower than other countermeasures, it carries a formal proof of security and can be used to secure any algorithm, two properties that make it very appealing. In particular, the ISW proof shows that a masking of order $m = 2t + 1$ is effective against any passive side-channel attack of order at most t. The complexity of carrying on such an attack grows exponentially in its order, and attacks above the third order are considered unfeasible.

In general, a masking scheme defines a way of transforming a value into a collection of shares (encoding) and, conversely, how to compose shares to retrieve the masked value (decoding). The masking scheme at the base of the ISW countermeasure described in Section 1.4.1 is usually called *boolean*, as opposed to another common scheme, *arithmetic masking*, which employs a modular sum (usually modulo the word size) to the same purpose, instead of a xor-sum. These two schemes are described in Sections 1.3.1 and 1.3.2, and together cover all the masking applications I am aware of. The existence of two different encoding scheme is motivated by efficiency, as it will be explained in the following sections.

These schemes have been first envisioned for hardware devices and will often refer to hardware terminology, talking about e.g. gates or latches. However, the same techniques have since been shown to be feasible in software as well, as first described by Messerges et al. in [60] and Rivain and Prouff, who adapted the ISW framework to software implementations in [73]. Additional considerations might be necessary to avoid introducing new sources of leakage, e.g. consecutively writing two shares of the same value in the same register leak sensitive information about the Hamming distance, which could be used to reveal the unmasked value.

1.3.1 Boolean masking

A boolean mask of order n is a tuple of $n+1$ shares $x_0, x_1, \ldots, x_n$ such that $\mathbf{x} = \bigoplus_{i=0}^{n} x_i$ (*boolean invariant*), where $\oplus$ is the bitwise exclusive-or and $\mathbf{x}$ is the masked value. Given a value to mask $\mathbf{x}$, a n-th order boolean masking can be composed by selecting n uniformly distributed independent random values $x_1, x_2, \ldots, x_n$ and imposing $x_0 = \mathbf{x} \oplus (\bigoplus_{i=1}^{n} x_i)$. This is exactly the masking transformation used in the ISW stateless transformer. Masked operations have been developed in this scheme for most boolean linear and non-linear operations [42, 71], for table look-ups [18, 19, 21, 60, 72, 73, 75] and more recently for modular addition and subtractions [23, 24, 35, 45].

1.3.2 Arithmetic masking

An arithmetic masking of order n is a tuple of $n+1$ shares $x_o, x_1, \dots, x_n$ such that $\mathbf{x} = \sum_{i=0}^{n} x_i \mod 2^k$ (*arithmetic invariant*), where k is usually the word size of the machine being used and $\mathbf{x}$ is the masked value. We proceed in the same way of boolean masking for producing a a n-th order arithmetic masking, by selecting n uniformly distributed independent random values $x_1, x_2, \dots, x_n$ and imposing $x_0 = \mathbf{x} - (\sum_{i=1}^{n} x_i) \mod 2^k$.

This scheme is convenient for masking arithmetic operations, such as modular additions/subtractions, and multiplications, because in those cases the arithmetic invariant is more efficiently maintained than the boolean one. However, most boolean masked operations don't have equivalents in the arithmetic masking scheme, and conversion algorithms are necessary to switch between the two, as detailed in Section 1.3.5 and later in Section 6 . Because of this conversion overhead, arithmetic masking is not advantageous, unless multiple arithmetic operations are performed in a row. It is worth noticing that this is not the case for any of the ciphers that will be considered in our contribution, with XTEA [78, 79] being the closest to having two consecutive additions.

1.3.3 Masked operations

Masked operations manipulate groups of shares in a certain encoding, producing another tuple of shares in the same encoding, such that the resulting set of shares can be decoded to the result of the non-masked operation applied to the decoded input shares. Masked operations on the same encoding can be chained, as shown in the case of the ISW transformer, allowing the construction of arbitrarily complex computations. In particular, all modern symmetric ciphers can be seen as deterministic boolean functions of their inputs (key and plaintext), and therefore can be protected in our attack model by chaining basic masked operations.

Algorithm 1.3.1: Rivain-Prouff MASKREFRESH

```
Input: $x_0, \dots, x_n$ k-bit shares s.t. $x = \bigoplus_{i=0}^{n} x_i$
Output: $y_0, \dots, y_n$ such that $y = \bigoplus_{i=0}^{n} y_i$

1 $x_0 \leftarrow y_0$
2 for $i \leftarrow 1 \dots n$:
3     $t \leftarrow \{0,1\}^k$
4     $y_0 \leftarrow y_0 \oplus t$
5     $y_i \leftarrow x_i \oplus t$
6 return $y_0, \dots, y_n$
```

1.3.4 Boolean masked operations

This section describes the known boolean masking schemes for all the operations employed in the symmetric ciphers I will consider in the next chapters. For some of them, such as NOT and AND, an established solution exists. For others, e.g. modular addition, various approaches have been devised over time, sometimes proven insecure and then patched. In these cases there is no established solution, and one has been devised based on various consideration of performance and provable security.

Negation A masked NOT gate is obtained using the method shown in Section 1.4.1, by observing that $\neg x = x \oplus 1$, from which follows that:

$$x = x \oplus 1 = (x_0 \oplus x_1 \oplus \cdots \oplus x_n) \oplus 1 = (x_0 \oplus 1) \oplus \cdots \oplus x_n = \neg x_0 \oplus \cdots \oplus x_n$$

Therefore, it's enough to negate only one of the shares. In the formula above x_0 is negated, but it could be any of the x_i.

XOR and other $\oplus$-affine operations A boolean masked XOR exploits the $\oplus$-linearity of the boolean masking by applying the operation to each pair of corresponding shares, i.e. $\mathbf{x} \oplus \mathbf{y} = (x_0 \oplus y_0, \dots, x_n \oplus y_n)$. More in general, any $\oplus$-linear operation can be masked by applying it to each pair of corresponding shares singularly. It should be noted that this operation combines two sets of shares without introducing new randomness. Although not necessary, a refresh of the result might help to increase the security in practical attacks, by avoiding long chains of operations which depend on few random bits. The mask refresh algorithm has been proposed by Rivan and Prouff in [73] and is reported in Section 1.3.1. The random values t used in the algorithm have the same bit size of the input shares x_i.

Bitwise non-linear operations (`AND` and `OR`) OR is reduced to a `AND` using De Morgan's Law: $\mathbf{x} \vee \mathbf{y} = \neg(\neg\mathbf{x} \wedge \neg\mathbf{y})$, and the masked negation described in this section. The masked `AND` is computed using the ISW construction shown in Section 10.

Shift, rotation, extension and truncation Shifts and rotations by a constant can be performed by shifting each share of the input singularly. In the case of shifts and rotations by a variable, the operation $a \ll b$ essentially amounts to the multiplication $a \cdot 2^b$, and it is computed as such. There is no algorithm to my knowledge that can perform this last operation more efficiently.

Extensions and truncations are assumed to be by a constant amount, since variable length numeric types are usually not a thing, and certainly not one in hardware. In the case of extensions are truncations by a constant, each share can be manipulated singularly.

In both all cases, the resulting shares will maintain the same distribution of values in the input, and no mask refresh needs to be be applied, as we are not mixing multiple sets of shares.

Table lookups A first approach to masked table lookups has been the *randomised table* (or, *table recomputation*) approach, proposed by Chari et al. in [19] and adopted by Messerges in [60]. Given a first-order boolean masked index (x_0, x_1) to be looked up in a table T and a random value r, a randomised table T' is built in memory such that $T'[a] = T[a \oplus x_1] \oplus r$ for each possible input to the table T. One can verify that the pair of shares $(r, T'[x_0])$ is a sound and private boolean masking of the looked-up value. The first share is a random value with all the desired properties, by hypothesis. The second share is obtained as the $\oplus$-sum of a value of the table T, on which no assumption can be made, and a perfectly random value r, obtaining another perfectly random share. It should be stressed that the whole table has to be calculated, not just the desired cell. Otherwise, an attacker will see the unmasked value as the index of the lookup, since $a \oplus x_1|_{a=x_0} = x_0 \oplus x_1 = x$.

The first higher-order masking scheme for which no attack has been devised to the best of my knowledge was introduced by Rivain and Prouff in [73], who exploited the fact that the Rijndael lookup table can be written as an exponentiation over $\mathbb{F}_{2^8}$ and generalised the ISW masking of the `AND` operation, which can be seen as a multiplication on $\mathbb{F}_2$. This scheme was later expanded to generic lookup tables by Carlet et al. in [18] by using Lagrange interpolation over a field $\mathbb{F}_{2^k}$, where k is the size of the lookup index, (the input size for the S-box). They proposed various optimised variants, with the best one requiring $\mathcal{O}\left(2^{k/2} \cdot n^2\right)$ field multiplications over $\mathbb{F}_{2^k}$, where n is the number of shares.

A second-order randomised table scheme was introduced by Schramm and Parr in [75], who generalised Chari's algorithm. However, a third-order attack against this method was shown by Coron et al. in [25], making it unsuitable for real-world use beyond the second order. Another second-order table recomputation scheme was introduced by Rivain et al. in [72], but no countermeasure existed for orders higher than the second until the one introduced by Coron in [21]. This scheme is essentially similar to the Schramm and Parr generalisation, but each row of the randomised table is masked with a different set of random shares, thus thwarting Coron's attack [25]. Additionally, masks are refreshed between every successive manipulation of the input, using the usual Rivain and Prouff procedure reported in Algorithm 1.3.1. The algorithm is proven secure in the ISW framework for stateless circuits, as introduced in Section 1.4.

Algorithm 1.3.2: Coron masking of table lookups

```
Input: $x_0, \dots, x_n$ : set of boolean $k$-bit shares
       $S$ : table with $2^k$ entries
Output: $y_0, \dots, y_n$ : set of $k'$-bit shares s.t. $\bigoplus_i y_i = S[\bigoplus_i x_i]$
Data: $T$, $T'$ : tables with $2^k$ entries, each one a set of $n+1$ $k'$-bit shares
1 for each $u \in \{0,1\}^k$:
2     $T[u] \leftarrow S[u], 0, \dots, 0$
3 for $i \leftarrow 0 \dots n$:
4     for each $u \in \{0,1\}^k$:
5         $T'[u] \leftarrow T[u \oplus x_i]$
6     for each $u \in \{0,1\}^k$:
7         $T[u] \leftarrow \textsc{MaskRefresh}(T'[u])$
8 $y_0, \dots, y_n \leftarrow \textsc{MaskRefresh}(T[x_n])$
9 return $y_0, \dots, y_n$
```

Coron et al. presented two variants of their countermeasure, a first one with time complexity $\mathcal{O}\left(2^k \cdot n^2\right)$, shown in Algorithm 1.3.2, and an optimised version having time complexity $\mathcal{O}\left(2^{k/2} \cdot n^2\right)$ with k size of the input and n number of shares -- the same of Carlet et al. approach. The spatial complexity is $\mathcal{O}(n)$ for both, thus better than the Rivain and Prouff approach and Carlet's generalization, which require $\mathcal{O}\left(n^2\right)$ additional space. However, Coron et al. suggested that a re-ordering of the operations can bring Carlet on par on space as well. They also showed that a complete masking scheme which uses their randomised table approach and the algorithms described in the previous sections for XOR, AND, OR, shifts and rotations achieve perfect security against a t-limited adversary who can move probes during the computation (i.e. in the *full model*), when using $n \geq 2t+1$ shares.

In practice, Coron's scheme is less efficient than Schramm and Parr's on Rijndael S-Boxes, but it is very interesting since it can be applied to a generic lookup table, with no assumption on the internal structure. Both of them have a significant computational overhead (often 100x or more), which might still be acceptable for masking the implementation of a symmetric cipher, where few table lookups take place, typically one per round. For my implementation, I chose Coron's countermeasure over Carlet's because of its higher flexibility, being fully parametric in the size of the input and output, while Carlet's requires a multiplication routine on the appropriate field.

Algorithm 1.3.3: AND-XOR-and-double addition

```
   Input: $x, y \in \mathbb{Z}_{2^k}$
   Output: $r = x + y \mod 2^k$
1  $A \leftarrow x, B \leftarrow y$
2  $C \leftarrow A \wedge B$
3  $A \leftarrow A \oplus B$
4  $B \leftarrow 0$
5  for $i \leftarrow 1 \dots k-1$:
6      $B \leftarrow B \wedge A$
7      $B \leftarrow B \oplus C$
8      $B \leftarrow B \ll 1$
9  $r \leftarrow A \oplus B$
10 return $r$
```

Algorithm 1.3.4: Kogge-Stone addition

```
   Input: $x, y \in \mathbb{Z}_{2^k}$
   Output: $r = x + y \mod 2^k$
1  $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2  $P \leftarrow x \oplus y$
3  $G \leftarrow x \wedge y$
4  for $i \leftarrow 1 \ldots n-1$:
5      $G \leftarrow \left(P \wedge \left(G \ll 2^{i-1}\right)\right) \oplus G$
6      $P \leftarrow P \wedge \left(P \ll 2^{i-1}\right)$
7  $G \leftarrow \left(P \wedge \left(G \ll 2^{n-1}\right)\right) \oplus G$
8  $r \leftarrow x \oplus y \oplus (G \ll 1)$
9  return $r$
```

Algorithm 1.3.5: Coron's higher order boolean masked `ADD`

```
   Input: $x_0, \ldots, x_n$ and $y_0, \ldots, y_n$ sets of $k$-bit boolean shares
   Output: $z = z_0, \ldots, z_n$ such that $\bigoplus_i z_i = \bigoplus_i x_i + \bigoplus_i y_i$
1  $w \leftarrow \textsc{SecureAnd}(x, y)$
2  $u \leftarrow \mathbf{0}$
3  $a \leftarrow x_0 \oplus y_0, \ldots, x_n \oplus y_n$
4  for $j \leftarrow 1 \ldots k-1$:
5      $ua \leftarrow \textsc{SecureAnd}(u, a)$
6      $u \leftarrow ua_0 \oplus w_0, \ldots, ua_n \oplus w_n$
7      $u \leftarrow u_0 \ll 1, \ldots, u_n \ll 1$
8  $z \leftarrow x_0 \oplus y_0 \oplus u_0, \ldots, x_n \oplus y_n \oplus u_n$
9  return $z$
```

Algorithm 1.3.6: Coron's Kogge-Stone boolean masked `ADD`

```
**Input:** $x_0, x_1$ and $y_0, y_1$ pairs of $k$-bit boolean shares
**Output:** $z_0, z_1$ s.t. $z_0 \oplus z_1 = (x_0 \oplus x_1) + (y_0 \oplus y_1)$
1  $n \leftarrow \max(\lceil \log_2 (k-1) \rceil, 1)$
2  $t \leftarrow \{0,1\}^k$
3  $u \leftarrow \{0,1\}^k$
4  $P \leftarrow x_0 \oplus y_0 \quad P \leftarrow P \oplus y_1$    ▷ $P \oplus x_1 = (x_0 \oplus x_1) \oplus (y_0 \oplus y_1)$
5  $G \leftarrow u \oplus (x_0 \wedge y_0) \quad G \leftarrow G \oplus (x_0 \wedge y_1) \quad G \leftarrow G \oplus (x_1 \wedge y_0)$
   $G \leftarrow G \oplus (x_1 \wedge y_1)$    ▷ $G \oplus u = (x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$
6  $G \leftarrow G \oplus x_1 \quad G \leftarrow G \oplus u$   // $G$ mask changes from $u$ to $x_1$
7  for $i \leftarrow 1 \ldots n-1$:
8      $H \leftarrow t \oplus \left(G \ll 2^{i-1}\right) \quad H \leftarrow H \oplus \left(x_1 \ll 2^{i-1}\right)$    ▷ $H \oplus t = (G \oplus x_1) \ll 2^{i-1}$
9      $U \leftarrow u \oplus (P \wedge H) \quad U \leftarrow U \oplus (P \wedge t) \quad U \leftarrow U \oplus (x_1 \wedge H)$
       $U \leftarrow U \oplus (x_1 \wedge t)$    ▷ $U \oplus u = (P \oplus x_1) \wedge (H \oplus t)$
10     $G \leftarrow G \oplus U \quad G \leftarrow G \oplus u$    ▷ $G \oplus x_1 = (G \oplus x_1) \oplus (U \oplus u)$
11     $H \leftarrow t \oplus \left(P \ll 2^{i-1}\right) \quad H \leftarrow H \oplus \left(x_1 \ll 2^{i-1}\right)$    ▷ $H \oplus t = (P \oplus x_1) \ll 2^{i-1}$
12     $U \leftarrow u \oplus (P \wedge H) \quad U \leftarrow U \oplus (P \wedge t) \quad U \leftarrow U \oplus (x_1 \wedge H)$
       $U \leftarrow U \oplus (x_1 \wedge t)$    ▷ $U \oplus u = (P \oplus x_1) \wedge (H \oplus t)$
13     $P \leftarrow P \oplus x_1 \quad P \leftarrow P \oplus u$   // $P$ mask changes from $u$ to $x_1$
14 $H \leftarrow t \oplus \left(G \ll 2^{n-1}\right) \quad H \leftarrow H \oplus \left(x_1 \ll 2^{n-1}\right)$    ▷ $H \oplus t = (G \oplus x_1) \ll 2^{n-1}$
15 $U \leftarrow u \oplus (P \wedge H) \quad U \leftarrow U \oplus (P \wedge t) \quad U \leftarrow U \oplus (x_1 \wedge H)$
   $U \leftarrow U \oplus (x_1 \wedge t)$    ▷ $U \oplus u = (P \oplus x_1) \wedge (H \oplus t)$
16 $G \leftarrow G \oplus U \quad G \leftarrow G \oplus u$
17 $z_0 \leftarrow x_0 \oplus y_0 \quad z_0 \leftarrow z_0 \oplus x_1$
18 $z_0 \leftarrow z_0 \oplus (G \ll 1) \quad z_0 \leftarrow z_0 \oplus (x_1 \ll 1)$
19 $z_1 \leftarrow y_1$
20 return $z_0, z_1$
```

Algorithm 1.3.7: Karroumi-Joye boolean masked `ADD`

```
Input: $x_0, x_1$ and $y_0, y_1$ pairs of $k$-bit boolean shares
Output: $z_0, z_1$ s.t. $z_0 \oplus z_1 = (x_0 \oplus x_1) + (y_0 \oplus y_1)$
$C \leftarrow \{0,1\}^k$
$T \leftarrow x_0 \wedge y_0 \quad \Omega \leftarrow C \oplus T$
$T \leftarrow x_0 \wedge y_1 \quad \Omega \leftarrow \Omega \oplus T$
$T \leftarrow x_1 \wedge y_0 \quad \Omega \leftarrow \Omega \oplus T$
$T \leftarrow x_1 \wedge y_1 \quad \Omega \leftarrow \Omega \oplus T$
$B \leftarrow \Omega \ll 1 \quad C \leftarrow C \ll 1$
$z_0, z_1 \leftarrow x_0 \oplus y_0, x_1 \oplus y_1$
$T \leftarrow C \wedge z_0 \quad \Omega \leftarrow \Omega \oplus T$
$T \leftarrow C \wedge z_1 \quad \Omega \leftarrow \Omega \oplus T$
for $i \leftarrow 2 \ldots k-1$:
    $T \leftarrow B \wedge z_0 \quad B \leftarrow B \wedge z_1$
    $B \leftarrow B \oplus \Omega$
    $B \leftarrow B \oplus T$
    $B \leftarrow B \ll 1$
$z_0 \leftarrow z_0 \oplus B$
$z_0 \leftarrow z_0 \oplus C$
return $z_0, z_1$
```

Additions and subtractions (modulo 2^k) Masked additions and subtraction are straightforward operations under arithmetic masking of any order (cfr. Section 1.3.2), since they are linear in that scheme and thus behave as XOR under boolean masking. Unfortunately, they pay the overhead of conversion from and to the boolean masking, which is used for the majority of the other operations. Such overhead could be amortised over the cost of many chained operations, but none of the symmetric ciphers I considered exhibits two additions or subtractions in sequence. The one that goes closest is XTEA [78, 79], having two additions per round, separated by a XOR.

It is therefore understandable that a "direct" approach that can operate on boolean masked shares is very appealing. This idea was first mentioned in Golić [35]. His solution was based on a masked ripple-carry adder, which gives good results in hardware, but is unfortunately too slow for software.

The first feasible software approach was introduced by Karroumi, Joye et. al in [45] and is essentially a first-order masking of the usual AND-XOR-and-double procedure shown in Algorithm 1.3.3, with a fused Goubin's arithmetic-to-boolean conversion inside (cfr. Section 1.3.5). The algorithms contains only boolean operations and the masking proceeds as shown in the previous sections for each operation involved. An in-depth description of the algorithm can be found in the same paper.

A proof of soundness and privacy is provided by exhaustion, showing that each value involved in the computation is either uniformly distributed and independent of the unmasked inputs, or distributed as the AND of two independent and uniformly distributed random values. Karroumi's algorithm requires $5k+8$ elementary operations (AND, XOR and shifts), where k is the size in bits of the input shares. The authors observe that a first-order addition under arithmetic masking, would cost $5k+17$ elementary operations, including switching from and back to boolean masking using Goubin's algorithms (cfr. Section 1.3.5). Both methods are therefore in the $\mathcal{O}(k)$ class, although a direct addition on boolean shares offers a visible speedup in practice.

Coron et al. improved over Karroumi in [24] by masking in a similar way the Kogge-Stone adder [51] shown in Algorithm 1.3.4, which computes the carry in $\mathcal{O}(\log k)$ instead of the classical $\mathcal{O}(k)$ of a ripple carry adder. Again, I refer to the paper for a description of the algorithm. Their practical results show a measurable speedup of 14% and 23% on larger word sizes ($k=32$ and $k=64$, respectively) when compared to an addition in the arithmetic scheme complemented with Goubin's conversions. The exact number of elementary operations is $28\cdot\log_2 k+4$, therefore having a much larger hidden constant than Karroumi's method, and solving the inequality $5k+8 > 28\cdot\log_2 k+4$ reveals that Coron's method is more advantageous for $k \geq 26$ bit, i.e. for 32- and 64 bit operands, a fact confirmed by their benchmarks on a 32bit AVR microcontroller.

The first higher order masked addition algorithm was proposed by Coron et al. in [23]. Their approach is essentially the same used by Karroumi et al., a higher-order boolean masking of an AND-XOR-and-double adder with a fused conversion, and has time complexity in $\mathcal{O}\left(n^2k\right)$, n being the number of shares and k the size of the operands. In the same paper in which Coron et al. presented their masked Kogge-Stone, they also suggested that the same generalisation could be applied to it reaching time complexity in $\mathcal{O}\left(n^2\cdot\log k\right)$, although they didn't expand on this point.

All the three methods mentioned above have been adopted in my implementation and selected based on which yields the best performance, using the criteria summarised in Table 1.1. The algorithms implemented are reported in Algorithm

1.3.7 for Karroumi-Joye, Algorithm 1.3.6 for Coron's Kogge-Stone and Algorithm 1.3.5 for Coron's higher order method.

Algorithm 1.3.8: Deriving secure subtraction from addition

```
Input: $x_0, x_1$ and $y_0, y_1$ boolean masked shares.
Output: $z_0, z_1$ such that $z_0 \oplus z_1 = (x_0 \oplus x_1) - (y_0 \oplus y_1)$
1 $s_0, s_1 \leftarrow$ SecureAddition$((\neg x_0, x_1), (y_0, y_1))$
2 $z_0 \leftarrow \neg s_0$
3 $z_1 \leftarrow s_1$
4 return $z_0, z_1$
```

Finally, it's worth mentioning that the algorithms for adding boolean-masked shares can be used to compute a subtraction as well, by exploiting the identities $\neg x = x \oplus 1$ and $-x = \neg x + 1$ in the way shown in Algorithm 1.3.8. This technique was first mentioned by Karroumi et al. in [45]

1.3.5 Conversions between masking schemes

As already mentioned, masked additions and subtractions are linear operations in an arithmetic masking scheme (cfr. Section 1.3.2) and therefore easily computed at any order, in the same way a XOR is in the boolean scheme (Section 6). Unfortunately, most operations have a natural or more efficient boolean masking, making a switching algorithm between the two a necessity.

As already discussed in Section 17, a few masked addition algorithms exist that can directly operate on boolean shares, and they are usually more efficient than addition in the arithmetic scheme in the context of symmetric ciphers, as it will be discussed in the following sections. In fact, these "direct" algorithms fuse together the operations of switching and adding, improving performance in most practical instances.

Table 1.1: Masked addition methods selected for the implementation.

		Masking order	
		$n = 1$	$n \geq 2$
Op. size	$k = 8, 16$	Karroumi	Coron's higher order
	$k = 32, 64$	Coron's Kogge-Stone	

None of the algorithms discussed in the following sections has been implemented in my solution, but they are mentioned anyway for completeness of this discussion.

Arithmetic to boolean Goubin first introduced a first-order arithmetic to boolean conversion with complexity $\mathcal{O}(k)$ in [36]. His algorithm is based on the following theorem and its corollary, proven in the same paper:

Theorem 1. *If we denote $x' = (A + r) \oplus r$, we also have $x' = A \oplus u_{k-1}$, where k is the size of the input and u_{k-1} is obtained from the following recursion formula:*

$$\begin{cases} u_0 & = 0 \\ u_{i+1} & = 2\left[u_i \wedge (A \oplus r) \oplus (A \wedge r)\right] \end{cases}$$

Corollary 2. *For any value γ, if we denote $x' = (A + r) \oplus r$, we also have $x' = A \oplus 2\gamma \oplus t_{k-1}$, where t_{k-1} is obtained from the following recursion formula:*

$$\begin{cases} t_0 & = 2\gamma \\ t_{i+1} & = 2\left[t_i \wedge (A \oplus r) \oplus \omega\right] \end{cases}$$

Both recurrences only contain XOR, AND and shifts, which can be easily masked using the algorithms presented in Section 1.3. Goubin formula was later improved by Joye in [45], who showed a rearrangement of the operations which increases performance, still remaining in the $\mathcal{O}(k)$ class, though.

A different first-order approach based on precomputed tables was devised by Coron and Tchulkine in [26], then improved by Neiße and Pulkus in [65]. Debraize [29] worked on the Coron-Tchulkine scheme, correcting a bug and improving performance for machines with small registers, again with complexity in $\mathcal{O}(k)$.

Coron et al. devised in [23] the first higher-order mask switching proven secure in the ISW framework. Given a set of arithmetic shares $A_1, \dots, A_n$, their transformation is based on splitting each A_i into n boolean shares $x_{i,1}, \dots, x_{i,n}$ such that $A_i = \bigoplus_{j=1}^{n} x_{i,j}$. These sets of shares are then summed using a secure addition algorithm (cfr. Section 17), resulting in $n-1$ operations at a cost in $\mathcal{O}\left(n^2 k\right)$ each, thus an overall complexity of $\mathcal{O}\left(n^3 k\right)$ -- or $\mathcal{O}\left(n^3 \log k\right)$ if Coron's Kogge-Stone addition is used, as they later suggested in [24]. They also proposed an optimised variant based on a recursive split of the set of shares that allows to save a number of operations in $\mathcal{O}(n)$, bringing the complexity down to $\mathcal{O}\left(n^2k\right)$ -- or $\mathcal{O}\left(n^2 \log k\right)$ in the Kogge-Stone case.

Given that this method employs a number of secure additions on boolean shares internally, one can easily realise that the conversion overhead is repaid only if n or more masked additions are chained, a condition that never happens in the case of the symmetric ciphers examined.

Algorithm 1.3.9: Goubin boolean to arithmetic switching

```
**Input:** $x, r$ $k$-bit boolean shares
**Output:** $A, r$ such that $x \oplus r = A + r$
1 $\Gamma \leftarrow \{0,1\}^k$
2 $T \leftarrow x \oplus \Gamma$
3 $T \leftarrow T - \Gamma$
4 $T \leftarrow T \oplus x$
5 $\Gamma \leftarrow \Gamma \oplus r$
6 $A \leftarrow x \oplus \Gamma$
7 $A \leftarrow A - \Gamma$
8 $A \leftarrow A \oplus T$
```

Algorithm 1.3.10: Coron higher-order boolean to arithmetic switching

```
**Input:** $x_1, \dots, x_n$ set of $k$-bit boolean shares
**Output:** $A_1, \dots, A_n$ such that $\sum_i A_i = \bigoplus_i x_i$
1 **for** $i \leftarrow 1..n-1$: $A_i \leftarrow \{0,1\}^k$
2 **for** $i \leftarrow 1..n-1$: $A'_i \leftarrow -A_i$
3 $A'_n \leftarrow 0$
4 $y_1, \dots, y_n \leftarrow \textsc{ArithmeticToBoolean}(A'_1, \dots, A'_n)$
5 $z_1, \dots, z_n \leftarrow \textsc{SecureAddition}((x_1, \dots, x_n), (y_1, \dots, y_n))$
6 $A_n \leftarrow \textsc{XorFold}(z_1, \dots, z_n)$
```

Boolean to arithmetic The first boolean to arithmetic conversion was proposed by Messerges in [60] and was based on unmasking the pair of shares (x_1, x_2) either to x or $\neg x$, based on a random bit, before applying a boolean mask. His reasoning was that a first-order attacker is not able to distinguish whether x or its negation is being processed, thus giving each bit a uniform distribution. However, a second-order attack against this method is well feasible, as it was first described by Coron and Goubin in [22], leaving this method broken.

Goubin later devised another first-order boolean masking scheme conversion, again in [36], based on the corollary of the following theorem:

Theorem 3. *Given $I = \{0, 1, \ldots, 2^k - 1\}$ and the function $\Phi_x : I \to I$ defined as $\Phi_x(r) = (x \oplus r) - r \mod 2^K$, the following holds:*

$$\Phi_x(x) = x' \oplus \bigoplus_{i=1}^{k-1} \left[\left(\bigwedge_{j=1}^{i-1} \left(2^j \wedge \neg x\right) \right) \wedge \left(2^i \wedge x\right) \wedge \left(2^i \wedge r\right) \right]$$

Corollary 4. $\Phi_x(r) = (x \oplus r) - r \mod 2^K$ *is affine over* $\mathbb{F}_2$.

His solution, shown in Algorithm 1.3.9, runs in constant time in the size of the input and is proven secure by exhaustion.

Coron et al. proposed the first higher-order switching algorithm in [23], which is shown in Algorithm 1.3.10, where *XorFold* is a $\oplus$-sum of the shares, preceded by the RP mask refreshing procedure reported in Algorithm 1.3.1. Internally, it exploits the transformation in the opposite sense which was presented in the same paper and discussed in Section 1.3.5. This construction was proven secure in the ISW framework against a t-limited attacker, where $n \geq 2t + 1$. The same considerations of the arithmetic to boolean transformation apply though, i.e. this construction is not advantageous over a direct addition on boolean shares, unless n or more arithmetic operations are chained.

1.4 The Ishai-Sahai-Wagner framework

A paper by Ishai, Sahai and Wagner [42] first introduced a theoretical framework for transforming generic circuits into circuits protected against a set of t probes in a provably secure way. A a similar construction had been already researched by Chari et al. [19] and Goubin and Patarin [37, 38]. However, the ISW framework is outstanding because it introduces a practical transformation that has been implemented with a reasonable performance overhead, albeit suffering from some limitations that will be explained in this section. In addition to that, the ISW framework is completely generic, as it does not exploit any specific property of the circuit being protected.

Proving the security against a first-order passive attack is straightforward, as it suffices to show that each internal variable has a uniform distribution. Such process is extended to higher orders by considering each pair of internal variables, by induction on the number. In alternative to exhaustive proof, the ISW framework proposes a simulation-based approach, in which the security of a construction can be proved by showing that any set of t of observations made by the attacker can be perfectly simulated without knowing the input variables.

Besides the proof, they also showed a practical construction that can perfectly secure any boolean circuit with $|C|$ gates by transforming it into a circuit with $\mathcal{O}\left(|C| \cdot t^2\right)$ gates. This construction is based on the fact that a boolean circuit can always be expressed as a combination of AND and NOT gates (the so-called NAND logic) and proceeds first by showing a provably secure transformation for these two logic gates, and then how the transformations can be chained to cover the entire input circuit. The construction is then extended to prove the same properties for stateful circuits, i.e. circuits augmented with memory cells.

Two models of attack are introduced in the ISW framework. The *restricted* model, in which the attacker can place at most t probes, but cannot move them during the execution, and the *full* model, where the attacker can move the probes, e.g. every few clock cycles. Security in the full model implies security in the restricted model, and for this reason only the former will be discussed in the following sections. It should be noted that security in the restricted model might be achieved with a lower overhead, although it would be arguably less secure, as the full model has been proven to be well feasible [52].

It is important to keep in mind that the computation must be secured in a way such that no information is leaked in any intermediate step, since there is no restriction on where the adversary can place its t probes, with the only exceptions detailed in Section 1.4.1. This means that not only the I/O function of each circuit described must be implemented as described, but also that the *exact* order of the operations matters. No simplifications should be attempted and most importantly, unless the circuit is hand-coded, care must be placed to avoid that any optimisation layer reorders gates in a way that thwarts the security of the transformation.

1.4.1 The stateless transformer

A deterministic, stateless circuit C can be seen as a directed acyclic graph whose vertices are boolean gates and edges are wires that connect gates. The size of the circuit $|C|$ is defined as the number of gates (or nodes) in the circuit (graph). We can also talk about width and depth of the circuit, referring to the usual concept of width and depth of the associated graph. Based on this definition, Ishai et. al

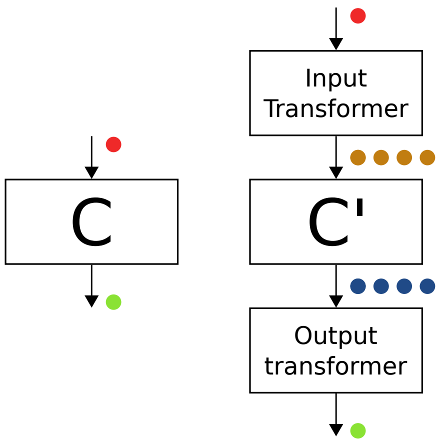


Figure 1.2: Circuit C and its ISW transformed C', with transformers.

introduced in [42] the randomised circuit as a deterministic stateless circuit, augmented with *random-bit gates*, which assumes an uniformly distributed random binary value for each invocation. Random-bit gates are nodes with no inputs and exactly one output. This construction is summarised in Figure 1.2.

The existence of two randomised circuits I and O, respectively the randomised input encoder and output decoder, is assumed. These special circuits cannot be probed by an adversarial and do not depend on the function implemented by the transformed circuit. Even if this hypothesis if very strong, Ishai et al. observe that it might still possible to achieve a reasonable approximation in real-world application by implementing them in tamper-proof logic that can be designed once and incorporated in all secured circuits.

Definition 5. Given I and O, let T be an efficiently computable deterministic mapping from a stateless circuit C to another stateless circuit C'. (T, I, O) is a *stateless transformer* if it has the following two properties:

soundness $O \circ C' \circ I$ and C are equivalent.

privacy any observation made by an adversary that can probe at most t wires of $O \circ C' \circ I$ can be perfectly simulated without knowledge of any wire in the circuit.

Theorem 6. *There exists a perfectly t-private stateless transformer (T, I, O) such that T maps any stateless circuit C of size n and depth d to a randomised stateless circuit of size $\mathcal{O}\left(nt^2\right)$ and depth $\mathcal{O}\left(d \log t\right)$.*

The demonstration given is constructive, and proceeds by showing two circuits $I$and O and a transformation T which respects the aforementioned assumptions. The strategy used is effectively a secret sharing scheme (see [76]), precisely a $m+1$ out of $m+1$ scheme, where $m = 2t$ and t is the number of probes the circuit should be secured against.

Input encoder I The encoder maps a single binary input x into $m+1$ binary outputs. The first m outputs are random bits $r_1, \dots, r_m$, chosen using m random-bit gates, while the last output is given as $r_{m+1} = x \oplus (\bigoplus_{i=1}^{m} r_i)$.

Output decoder O The output decoder is the symmetric of the input encoder I, and produces a single output y out of $m+1$ inputs $r_1, \dots, r_{m+1}$ as $y = \bigoplus_{i=1}^{m+1} r_i$.

Transformer T Assuming without loss of generality that the circuit is composed of NOT and AND gates only, the following transformations are applied:

NOT $m+1$ input shares $x_1, \dots, x_{m+1}$ are transformed in $m+1$ output shares $y_1, \dots, y_{m+1}$ such that shares $x_i = y_i$ for each $i = 1 \dots m$, and $x_{m+1} = \neg y_{m+1}$.

Algorithm 1.4.1: ISW AND masking

```
Input: $a_0, \dots, a_n$ and $b_0, \dots, b_n$ : sets of $k$-bit boolean shares.
Output: $c_0, \dots, c_n$ : boolean shares s.t. $\bigoplus_i c_i = (\bigoplus_i a_i) \wedge (\bigoplus_i b_i)$
1  for $i \leftarrow 0 \dots n$:
2      for $j \leftarrow i+1 \dots n$:
3          $z_{i,j} \leftarrow \{0,1\}^k$
4          $z_{j,i} \leftarrow (z_{i,j} \oplus a_i b_j) \oplus a_j b_i$
5  for $i \leftarrow 0 \dots n$:
6      $c_i \leftarrow a_i b_i$
7      for $j \leftarrow 0 \dots s$ :
8          if $i \neq j$:
9              $c_i \leftarrow c_i \oplus z_{i,j}$
10     return $c_0, \dots, c_s$
```

AND Given two encoded inputs $a_1, \dots, a_{m+1}$ and $b_1, \dots, b_{m+1}$, we observe that $a = \sum_{i=1}^{m+1} a_i \mod 2$ and $b = \sum_{i=1}^{m+1} b_i \mod 2$. Therefore, $c = a \wedge b = a \cdot b \mod 2 = \sum_{i=1}^{m+1} a_i \cdot b_i$. This product must be computed in a way that no intermediate value reveals information to the adversary. For this purpose, random values $z_{i,j}$ are calculated for each $1 \leq i < j \leq m+1$. Then, $z_{j,i} = (z_{i,j} \oplus a_i b_j) \oplus a_j b_i$ are computed. Finally, the output shares $c_1, \ldots, c_{m+1}$ are computed as $c_i = a_i b_i \oplus \left(\bigoplus_{j \neq i} z_{i,j}\right)$. The process is described in Algorithm 1.4.1 and can be visualised in form of a matrix $[z_{i,j}]$, where the lower part is composed of random values, the upper part is computed as just described, and the diagonal is left undefined.

Proving the property of *soundness* as given in Definition 5 is straightforward, while proving the *privacy* of this construction requires some additional considerations, which are explained in Section 1.4.2.

1.4.2 Proof of privacy of the ISW stateless transformer

Algorithm 1.4.2: ISW simulation

```
Input: $q_h$ wires queried by the attacker
Output: $[z_{i,j}]$ transformation matrix

// Step 1: build $I$ set of the selected indices
1  $I \leftarrow \emptyset$
2  for $h \leftarrow 1 \ldots |I|$ :
3      if $q_h$ is $a_i$, $b_i$, $a_i b_i$ or $z_{i,j}$ $(i \neq j)$ or a $\oplus$-sum of those:
4          $I \leftarrow I \cup \{i\}$
5      else:  // $w_h$ in form $a_i b_j$ or $z_{i,j} \oplus a_i b_j$ (for $i \neq j$)
6          $I \leftarrow I \cup \{i, j\}$

// Step 2: build the$[z_{i,j}]$ matrix
7  for $i \leftarrow 1 \ldots m+1$, $j \leftarrow 1 \ldots m+1$:
8      if $i \notin I$:
9          $z_{i,j} \leftarrow$ undefined
10     elif $i \in I$ and $j \notin I$:
           // If $i < j$ this exactly what happens in $C'$
11         //
12         if $j > i$ we exploit the fact that $z_{i,j}$ will never be
            used in the computation for any $q_h$
13         $z_{i,j} \leftarrow \{0, 1\}$
14     else:  //
15     if $i \in I$ and $j \in I$
16         $z_{i,j} \leftarrow \{0, 1\}$
17         $z_{i,j} \leftarrow z_{j,i} \oplus a_i b_j \oplus a_j b_i$    //
18         We have $a_i$, $b_i$, $a_j$ and $b_j$.
19         return $[z_{i,j}]$
```

The proof of privacy of the ISW transformer (T, I, O) proceeds by showing that the observations made by a t-limited adversary can be perfectly simulated without knowing the input values of the C. A probing attack can be imagined as the process of providing an answer to any t queries of the attacker, in the form "is *this* wire high or low?" or any equivalent. When I say "simulation", I refer to providing answers to such queries without actually probing the circuit C'. A perfect simulation is one such that that the (statistical) distribution of the answers is identical to what the attacker would obtain by actually probing C'. If one can provide a perfect simulation, it means that the attacker is not able to extract any information about the inputs of the circuit, no matter which observation it may make.

Single NOT gate All the non-negated shares $a_1, \dots, a_m$ keep the same distribution of the input, i.e. independent and uniformly distributed. The only negated share a_{m+1} is independent and uniformly distributed as well, since each of its bits is flipped starting from a value which is independent and uniformly distributed.

Single AND gate Let C' be the transformed circuit of a single AND gate, with inputs $a_1, \dots, a_{m+1}, b_1, \dots, b_{m+1}$ and outputs $c_1, \dots, c_{m+1}$. Given the set $w_1, \dots, w_{m+1}$ of wires the attacker is probing, we will show that, fixed the inputs a and b to the original circuit, the joint distribution of the values assigned to the wires w_h can be perfectly simulated with the only knowledge of at most m shares of a_i and m of b_i. We call I the set of the indices i corresponding to the shares we need to perform the simulation and denote the sets of selected shares as $a|_I$ and $b|_I$.

In a true evaluation of C', the inputs a_i and b_i have by construction the property that any m shares have the distribution of uniform, independent random bits. Therefore, the values of $a|_I$ and $b|_I$ can be perfectly simulated by picking uniformly distributed, independent random values, if $|I| \leq m$. Given that $a|_I \cup b|_I$ is the only input to our simulation, and that it can be perfectly simulated without probing C', showing a procedure to build I and a way to perform the simulation will prove the privacy of the ISW stateless transformer. The ISW simulation algorithm works by iteratively generating I, and it is shown in Algorithm 1.4.2. The algorithm is split in two phases:

1. The set I is built out of the queries q_h submitted by the attacker.
2. A matrix $[z_{i,j}]$ is built from I and q_h

The $[z_{i,j}]$ matrix is used as described in Section 10, and it has the property that all the entries required for calculating the answer to the q_h queries have been assigned, for any set of q_h. It useful to observe that since $|I| \leq m$ and each iteration of the first step adds at most 2 indices, the cardinality of I can be at most $m = 2t$, with $t$order of the attack. Hence, the $m = 2t$ constant used throughout our description.

We are able to produce a perfect simulation for any m wire in C' without knowledge of the inputs to C. $\square$

Generalisation The same algorithm can be generalised by examining each transformed gate and computing the respective set I. Since at most m wires can be probed in the whole circuit, $|I|$ is still bounded by m. Then, a simulation is run for each gate, proceeding from the inputs to the outputs of the circuit. It can be observed that all the intermediate inputs required for the simulation can be obtained by running the simulation on the preceding gates.

Randomness economy The transformed gates described in Section 10 requires $\Theta\left(t^2\right)$ independent random values, which make the construction very demanding in the case of large circuits. Ishai et al. leave open the problem whether the randomness could be reused under certain hypothesis, suggesting that it could be possible to use the same approach to reducing the complexity in MPC protocols via limited independence following the approach in [17].

1.4.3 Extension to stateful circuits

Once privacy has been achieved in the stateless model, the same construction can be extended to the stateful case. For doing so, the existence of a stateless transformer (see Section 1.4.1) T_C is assumed that satisfies the *re-randomized outputs property*, i.e. the fact that the encoding of each bit in the output of the transformed stateless circuit is t-independent from all other values, even given all the encodings of the input bits. The said property may be proven for the construction discussed in the previous sections.

A stateful transformer $T = (T_C, T_S)$ is obtained by augmenting T_C with a transformer T_S for memory cells, as follows. Given $E_t(x)$ the encoding used in T_C, each cell of memory will be encoded with $E_{2t}(x)$. The doubling of the number of shares is necessary because an attacker may place t probes on a memory cell, obtaining t

measurements at the end of a clock cycle, and other t in the next one, effectively observing $2t$ values from the internal state.

The simulation proceeds by transforming a stateful circuit C into a stateless equivalent C', composed by the concatenation of a circuit Q_i for each step of the computation, having the initial state of the memory as a hidden input, and the final state as an output. An attacker is allowed to probe t wires in the stateful circuit C, which means t wires for each of the Q_i . However, by virtue of the re-randomization property, a full simulation of the entire unwound circuit can be provided by recovering only a bounded set of inputs for each Q_i. The stateless proof is then carried on C', obtaining the following result:

Theorem 7. *There exists a perfectly t-private stateful circuit transformer T which maps any stateful circuit C of size n and depth d to a randomised stateful circuit of size $\mathcal{O}\left(nt^2\right)$ and depth $\mathcal{O}\left(d\log t\right)$.*

1.5 Dataflow analysis

The dataflow analysis is a framework of reasoning for deducing properties of *run time* values in a computer program at *compile time*. It is the most common tool used by optimising compilers to guide code transformation, but it also proves useful in itself. In fact, the tool introduced in the following chapters is structured as a sequence of dataflow analysis passes.

The analysis works by pairing each node of a graph-based representation of the program with an initial value for the property to be computed, and then iteratively solving a set of simultaneous equations over them, called the *dataflow equations*. The graph in question is usually the CFG introduced in Section 1.5.1, although applications to the call graph and other graph representations exist as well. To account for the propagation of the property through loops in the graph, dataflow equations are solved at fixed point, and various approaches have been devised. The most common one is the iterative round-robin algorithm, which is introduced in Section 1.5.3.

Dataflow analysis works well for scalar values, but is less suited to reasoning about other familiar programming concepts. For instance, arrays must be either represented as a whole. No way exists in the classical framework to treat this concept with finer granularity, apart from breaking every index in a different variable. Even more challenging are pointers, which suffer from severe issues, as they

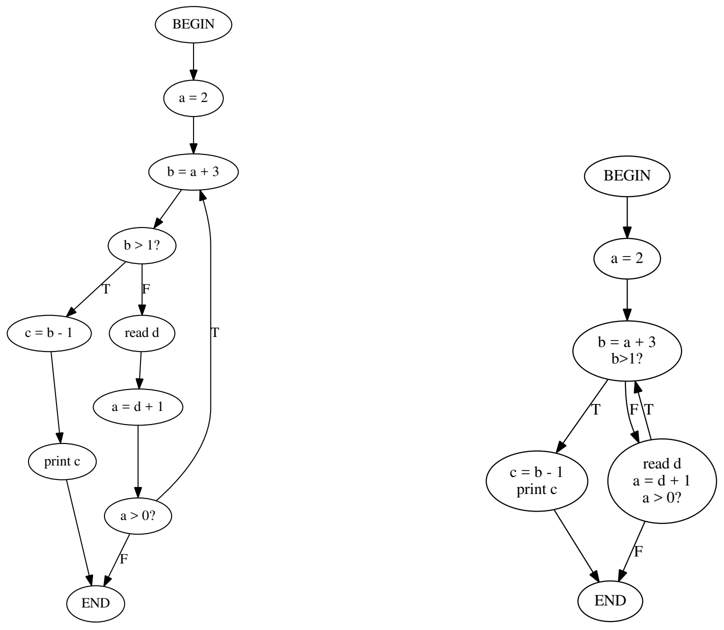


Figure 1.3: Example of expanded and collapsed CFG

could be pointing to nothing valid, and introduce even more severe ones, such as *aliasing*, i.e. the fact that two pointers might end up referring to the same memory location even through completely independent calculations.

1.5.1 The control flow graph (CFG)

The control flow graph is a representation of a program in form of a graph which was first introduced by Allen in [5], who based her work on the previous use of boolean connectivity matrices for representing control flow made by Prosser in [68]. The canonical definition is given in Ferrante et al. [31]:

Definition 8. A control flow graph is a directed graph $G = (V, E)$, with the following properties:

1. V contains at least two nodes, BEGIN and END.
2. Each node in V has at most two successors.
3. There is always at least one path from BEGIN to END.

Each node in the graph, excluding BEGIN and END, represents a single operation from the instruction set of the underlying target, either a real machine or an intermediate representation. Of the two (pseudo) nodes, BEGIN represents the entry point, and likewise END the exit point of the program. Nodes that have multiple successors are called *branch-* or *condition nodes* and, as the name implies, represent those points in a program where different execution paths might be taken based based on a condition. The classic definitions restrict the number of successors to two, but this constraint is just for theoretical simplicity and it is not strictly required. In fact, many CFG-based analysis frameworks admit nodes with more than two successors, for instance the one built into LLVM [54].

A set of nodes $B \subseteq V$ is called a *basic block* if each node has exactly one predecessor and exactly one successor, both belonging to B, with the exception of two $a, b \in B$, such that a has exactly one successor $a' \in B$, and b exactly one predecessor $b' \in B$.

It is easy to observe that the instructions in a basic block are always executed in sequence, from the first (a) to the last (b). Exploiting this fact, the control flow graph is often collapsed so that each node represents a basic block instead of a single operation: "uninteresting" internal transitions are then hidden, clearly outlining conditional arcs, those whose origin is a condition node. The collapsed graph better outlines the control flow of the program i.e. the different paths that the execution might take from BEGIN to END, as shown in Figure 1.3.

An important fact to keep in mind is that the a control flow graph offers a conservative representation of the control flow, in that it can't tell which paths from BEGIN to END might eventually be taken and which cannot because of impossible conditions. In fact, executable paths are not computable. Notwithstanding this limitation, the CFG is at the heart of many optimising compilers, because it offers a representation of the program which is very convenient for a broad range of code analyses based on the dataflow framework.

1.5.2 Dataflow equations

A generic set of dataflow equations is written in one of the two following forms:

$$
\mathit{Forward}\ \begin{cases} \mathit{Out}\left[i\right] & = T_i\left(\mathit{In}\left[i\right]\right) \\ \mathit{In}\left[i\right] & = J_{p\in \mathit{pred}[i]}\left(\mathit{Out}\left[p\right]\right) \end{cases}
$$

$$
\mathit{Backward}\ \begin{cases} \mathit{In}\left[i\right] & = T_i\left(\mathit{Out}\left[i\right]\right) \\ \mathit{Out}\left[i\right] & = J_{s\in \mathit{succ}[i]}\left(\mathit{In}\left[s\right]\right) \end{cases}
$$

Where $\mathit{pred}\left(x\right)$ is the set of nodes y such that an edge $\left(y,x\right)$ exists in the CFG and likewise is defined $\mathit{succ}\left(x\right)$. T_i is the transfer function of an instruction i, J is the *join* function (or operator), which combines the results of all the predecessors for forward equations, or successors for backwards equations. Both T_i and J are specific to the analysis.

Although not necessary, most instances of dataflow equations express the property as a set attached to each node, and use either set union or set intersection as the join operator. These two orthogonal properties allow to classify the set-based dataflow analyses as follows:

$$
\textbf{Set based dataflow equation} \begin{cases} \textbf{Flow direction} & \begin{cases} \textbf{Forward} \\ \textbf{Backward} \end{cases} \\ \\ \textbf{Set operation} & \begin{cases} \textbf{Union} \\ \textbf{Intersection} \end{cases} \end{cases}
$$

Since a CFG cannot always distinguish impossible paths from admissible ones, dataflow applied to a CFG will give answers that assume all paths might eventually be taken. This leads to a conservative solutions, which might overestimate some effects, for instance telling that a definition will be used, although no computable path to its use exists, but never leave out admissible cases. This fact is of particular interest, as it means that dataflow solutions are conservative approximations of the properties exhibited by the program.

Algorithm 1.5.1: Round-robin forward dataflow solver

```
Input: The CFG of the function, to obtain $pred(n)$ and $Instructions$
       $entry \in Instructions$ : the entry point of the function
Output: $(Reach_{in}, Reach_{out})$ : the reaching definitions property.

$Reach_{out}[entry] \leftarrow \emptyset$
for each $i \in Instructions$:   // Initialisation
    $Reach_{out}[i] = \emptyset$

do:   // Propagation
    $changed \leftarrow false$
    for each $i \in Instructions$:
        $Reach_{in}[i] \leftarrow \bigcup_{p \in pred[i]} Reach_{out}[p]$
        $temp \leftarrow Gen[i] \cup (Reach_{in}[i] - Kill[i])$
        if $Reach_{out}[i] \neq temp$:
            $Reach_{out}[i] \leftarrow temp$
            $changed \leftarrow true$
while $changed$

return $Reach_{in}, Reach_{out}$
```

Algorithm 1.5.2: Round-robin forward dataflow solver with worklist

```
Input: The CFG of the function, to obtain $pred(n)$ and $Instructions$
       $entry \in Instructions$ : the entry point of the function
Output: $(Reach_{in}, Reach_{out})$ : the reaching definitions property.

$Reach_{out}[entry] \leftarrow \emptyset$
$Worklist \leftarrow \emptyset$
for each $i \in Instructions$:   // Initialisation
    $Reach_{out}[i] = \emptyset$
    $Worklist \leftarrow Worklist \cup \{i\}$

while $Worklist \neq \emptyset$:   // Propagation
    pick and remove any $i$ from $Worklist$
    $Reach_{in}[i] \leftarrow \bigcup_{p \in pred[i]} Reach_{out}[p]$
    $temp \leftarrow Gen[i] \cup (Reach_{in}[i] - Kill[i])$
    if $Reach_{out}[i] \neq temp$:
        $Reach_{out}[i] \leftarrow temp$
        for $s \in succ(i)$:
            $Worklist \leftarrow Worklist \cap \{s\}$

return $Reach_{in}, Reach_{out}$
```

1.5.3 The dataflow solver

An example of property that can be calculated through dataflow analysis is the *reaching definitions* $(Reach_{in}, Reach_{out})$. Once defined a *definition* of a variable x as an instruction that assigns, or may assign, a value to x, the *reaching definitions* property is defined for each node $i \in V$ as the set of definitions d for which a path in the CFG exists from d to x such that d is not killed along that path. The corresponding dataflow equations are following:

Initialisation $Reach_{out}[i] = \emptyset$ for each instruction i in the CFG.

Update equation $\begin{cases} Reach_{in}[i] & = \bigcup_{p \in pred[i]} Reach_{out}[p] \\ Reach_{out}[i] & = Gen[i] \cup (Reach_{in}[i] - Kill[i]) \end{cases}$

Where $Gen[i]$ is the set of definitions generated by the instruction i and $Kill[i]$ is the set of definitions killed by i. In general, most analyses define an initialisation value for the sets, and one or more update equations which give the value of a set at iteration $i+1$, based on the sets calculated at iteration i.

This set of equations is solved at fixed point, i.e. by iteratively recalculating the property based on updated values, until the values of the properties affixed to all the graph nodes converge to a fixed value. The simplest solver is the round-robin procedure, shown in the forward variant in Algorithm 1.5.1, with the reaching definitions plugged in. This algorithm works by exploring each node in the CFG until no property set changes from the previous iteration.

An optimisation of this approach is achieved by observing that the only nodes whose $Reach_{in}$ *might* change at the next iteration are the successors of those that have changed at the current one. This improved variant works by keeping a work-list, as shown in Algorithm 1.5.2, again for the reaching definitions. For backwards analyses, the algorithm has to be slightly modified by populating the worklist with the *predecessors* of the node whose property was updated in the current iteration.

Another interesting improvement is found in Ferrante et al. [31], who introduced the *incremental dataflow*, allowing the dataflow solution to be updated incrementally as the CFG changes, for instance under the effect of other optimisations which require a recalculation of the same property.

So far no guarantee is given about the termination of the solving procedures. Turns out, the equations as written above may either add new elements to the set associated to a particular node or none at all, but they cannot remove an element.

Each property set is a subset of V, which can be assumed to be finite in practice, as V is the set of instructions in the program, which cannot be infinite in any real-world instance. Therefore, at a given iteration a property set might either be left untouched or enlarged, and this enlargement is bounded by the size of V. These two conditions lead to the guarantee of a fixed point, either because the set attached to a node stabilises, or because saturation is reached, i.e. the set is V itself and cannot grow any larger.

Kildall introduced in [46] a theoretical formalisation of dataflow based on semilattices, in which the correctness and termination properties can be formally proven for all analyses, also those which have an underlying (bounded) infinite semilattice. Correctness is especially relevant, since dataflow is often used by a compiler to prove that a certain modification of the program will preserve the semantic. Whenever the equations admit multiple solutions, these will be ordered in the Kildall framework, allowing to identify *which* one will be calculated by a certain solver algorithm.

While termination and correctness are guaranteed irrespective of the solver, the speed at which the fixed point is reached is not guaranteed at all. In fact, it depends heavily on the order in which the nodes are explored. Generally, forward analyses are faster when the CFG is explored in forward post-order, BEGIN to END, and backward analyses when the visit happens in reverse post-order on the reversed graph, i.e. the one obtained by swapping the direction of the edges in E, END to BEGIN. A “rapid condition” is given in the lattice-theoretic framework, allowing to identify the fastest order of computation for a given analysis.

Chapter 2

Vulnerability analysis

The contribution described in this thesis is an analysis tool which associates a vulnerability index to each instruction based on the various side-channel attack models described in the previous chapter, identifies the security critical instructions, produces meaningful visualisations and applies a masking countermeasure as described in Section 1.3 at any specified order. The input is high-level source code (e.g. C code) augmented with specific annotations, and the executable output is produced in a fully automated way, enabling a selective protection which is effective against attacks without needlessly hindering performance. In fact, the countermeasures have a non negligible impact on the execution time, so it is essential to apply them only when necessary, and this tool aims at providing a sound way to determine where the necessity arises in the generated code.

The process happens in two distinct stages: first a vulnerability index is calculated for each instruction and the instructions are marked as vulnerable (or not) according to a predefined criterion. Then, a masking countermeasure is applied to the instructions identified as vulnerable. Both stages are composed of multiple compiler passes whose results build one on top of the others. Doing so allows to test each component singularly, rather than the analysis as a whole, and helped building a solid engineering foundation to the project.

The tool itself is implemented as a collection of analysis and optimisation modules for the LLVM compiler framework [54]. Each module, or *pass* in the LLVM terminology, is a functor that visits the CFG (cfr. Section 1.5.1) built on the LLVM Intermediate Representation, (IR) to calculate a property or apply a graph transformation. The IR is high-level enough to retain the expressive power required to carry out the analysis, but operates with concepts at a low enough level to ensure that the generated assembly will not significantly alter the results.

The name of the passes and the high-level information flow graph among them is shown in Figure 2.1, and the inner working of each of them will be detailed in the following of this chapter, proceeding from the source code to the masked binary. Each group shown in the figure correspond to one high-level phase of the analysis. During features recognition (Section 2.3), the source code is processed to identify the features of the cipher implementation -- rounds, plaintext and key inputs, ciphertext output and S-Boxes -- and calculate the properties of interest, such as dead bits in S-Boxes and plaintext dependency depth. The, the selection phase (Section 2.4) identifies the most likely candidates for a side-channel attack, i.e. those where an attack may be mounted with the smallest complexity possible, thus representing a lower bound for the security of the implementation. A set of instructions is selected for attacks from top (forward), and another for attacks from bottom (or backwards). The vulnerability information is propagated to all the instructions in the cipher during the propagation phase (Section 2.5), which happens once for the forward set and once for the backwards, thus the apparent duplication. Once the vulnerability information has been propagated, a vulnerability index is calculated in the filtering and aggregation phase (Section 2.6) and vulnerable instructions are then identified and replaced with secure equivalents, as it will be described in Chapter 2.8.

The implementation resulted in around 9000 lines of source code, organised as a plugin for LLVM optimiser, plus the modifications to the Clang frontend necessary to support source level annotations of rounds, plaintext, key and S-Boxes. Besides the algorithms, the implementation brought a few additional difficulties which are explained in the following sections. As one might have realised, writing any nontrivial piece of code entails a series of challenges for the author to produce something which is both readable and reasonably performing. However, not all of these decisions are worth being expanded into a discussion. Only the larger design and implementation issues are presented in this section.

2.1 Notation

Throughout this chapter and the next one, I will adopt the following notation in formulas and algorithms: let A be a $n \times m$ boolean matrix (n rows and m columns),

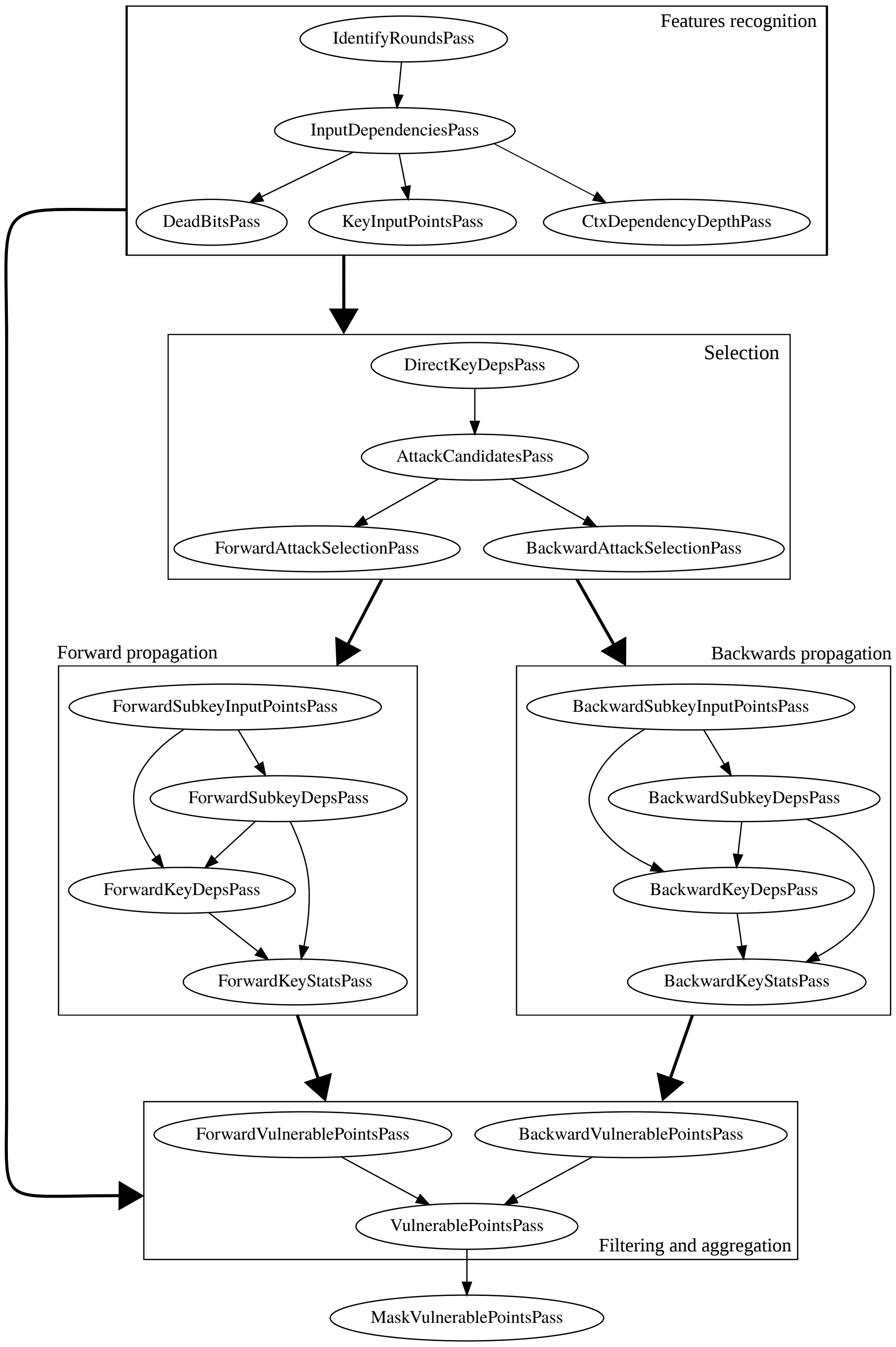


Figure 2.1: LLVM passes implemented and information flow between them.

$(A)_i$ will denote the i-th column of A, $a_{i,j}$ the element of A in row i and column j, $A \gg 1$ the matrix obtained by shifting the columns of A by one towards larger indices, discarding the one with the largest index and replacing the $0th$ with a column of zeroes. Finally, $cols(A)$ denotes the number of columns m of the matrix A.

Assuming that A and B are either matrices or vectors, $|A|_1$ denotes the number of elements of A equal to 1 and $A \vee B$ is the cell-by-cell $\vee$-sum of A and B. Additionally, $\mathbf{0}$ denotes the zero matrix or vector, having the dimensions appropriate for the context it is used in, and $\mathbf{1}$ similarly denotes the appropriately shaped matrix or vector having a 1 in each cell. Finally, given a mapping $M : K \rightarrow V$ and two values $k \in K$ and $v \in V$, $M[k]$ will denote the value v associated with the key k by M, and $M[k] \leftarrow v$ the association of k with v, possibly replacing the existing one.

2.2 Frontend

The first set of modifications have been done in the frontend and core of the compiler, which comprise all the steps that lead from the source code to the input of the analysis.

2.2.1 Loop check and unrolling

The first step of the analysis is to ensure that the procedure under analysis does not contain any loop. Eliminating loops and testing the loop-free condition from the very beginning allows the analysis to exit early in those cases that cannot be processed and, more importantly, allows to make the hypothesis that the CFG is loop-free in the following passes. Not only this removes further expensive checks, it also simplify the storage and processing of the vulnerability properties, as each instruction has a unique value attached, rather than possibly one for each iteration. Since this pass dominates all the others and CFG is not modified throughout the entire set of passes, this check will not have to be repeated.

Loops in the source code are allowed, as long as they can be eliminated by full unrolling, a code transformation which replaces the loop with a sequence of copies of the body, each with the loop inducing variables replaced by their respective values at the iteration the copy corresponds to. This transformation is only possible if the boundaries of the loop are known at compile time, i.e. if the compiler is able to calculate the value of the loop inducing variables at each iteration. However, such a requirement is not restrictive, as all the modern ciphers are designed with a known number of rounds and well defined round parameters. Full unrolling may be achieved in source code by annotating all the loops in the cipher implementation with a non-standard `#pragma unroll` or any language equivalent. In this case, the unrolling is performed by the LLVM unroll pass, scheduled for execution before the analysis.

2.2.2 Optimisation pipeline

One of the goals of the project is allowing the code to be compiled through an almost unmodified toolchain until it comes to the vulnerability analysis itself. This involves being able to support most of the normally scheduled optimisations. The solution described in this thesis enables a set of passes very near to the -O3 switch in Clang, with the exclusion of 3 which had to be disabled. These are the *slp-vectorizer,* which tries to group multiple arithmetic operations with vectorial equivalents, *loop idiom* and *memcpy*, which both try to recognise patterns in memory operations performed inside loops and replace them with intrinsics with potential hardware support -- for instance replacing `for (int i = 0; i < 5; ++i) b[i] = a[i];` with a call to `memcpy(b, a, 5*k)` (assuming that `a` and `b` do not alias). The first of the three introduces vector types in the intermediate, which the analysis cannot handle in its current status, while the other two introduce opaque intrinsics which would need special casing.

The output of the security analysis and masking is run through a final *dead code elimination* pass to remove unneeded, and the IR is passed to the code generation phase, which turns it into machine code. It is crucial that no other optimisation pass is run *after* the analysis, as it could undo the transformations or simply hinder the security of the code, which in many cases depends on the operations being performed in redundant way or a very precise order.

2.2.3 Source level markers

The analysis relies on the user marking the input values, plaintext, key and S-Boxes in the source code. In order to produce visualisations of the properties calculated by the analysis, the user may also delimit the blocks of codes correspondent to the rounds of the cipher.

The Clang C/C++/Objective C frontend to LLVM has been modified to process specific source code annotations and emit the information in the IR. Additionally, the optimisation pipeline has been modified to remove those transformations that might affect the precision of the analysis.

Cipher variables markers Key and plaintext annotations are handled using the `__attribute__` mechanism, which allows to attach pre-declared information to variables and functions. Specifically, two boolean flags have been added: `__attribute__((key))` and `__attribute__((plaintext))`. The flags are retrieved while lowering the parsed syntax tree to its intermediate representation and translated into decorator functions -- `llvm.crypto.key` and `llvm.crypto.plain`, taking the annotated variable as single parameter.

The decorator functions are defined as opaque intrinsics, that is to say implicitly declared functions whose definition is not known to the optimiser and might include side effects. The optimiser cannot prove that it would be safe to move or drop the function call and this guarantees that the annotation will not be wiped away by an optimisation pass, as it would happen if IR comments were abused as markers. The marker intrinsics will be discarded in the core of the compiler, just before code generation in the backend, and never appear in the executable.

S-Box markers Those S-Boxes represented in the source code as as arrays of constant integers, similarly marked with a `__attribute__((sbox))`. The frontend inserts a pointer to each array in a list, and this list is attached to the IR as a global metadata entry named `@llvm.crypto.sbox`. The analysis passes will then retrieve the list by name and keep track of the S-Boxes, and calculate the dead bits (see Section 2.3.2) property, which will be used to identify the portions of the code most vulnerable to the side-channel attack models considered.

Round markers The modifications to Clang also allow to annotate blocks of instructions in the source code to identify them as a rounds of the encryption routine. For the languages supported by Clang, I chose to use code blocks (`{}` enclosed sequences of statements) decorated with an ad-hoc `#pragma crypto loop`. Pragma directives are handled by Clang directly in the parser for performance reasons, and this entailed a lot of modifications in order to add a new one. However, the clean result in itself justified the effort.

The pragma directive is transformed into an flag which is attached to the associated block to indicate that it constitutes the body of a loop. When lowering the instructions contained in that block to their intermediate representation, two synthetic loops are inserted right before and after it. These loops have an empty body and a looping condition given by other two intrinsics, `llvm.crypto.begin` and `llvm.crypto.end`. The optimiser is not able to eliminate or move these loops because of potential side effects in the intrinsics, and they effectively serve as delimiters, or markers, of the round code.

These delimiters are entire basic blocks and not single instruction. At the moment, LLVM does not offer a facility to attach metadata to entire blocks, therefore they need to be re-identified by the analysis. A round marker will be defined as the inmost loop containing the aforementioned intrinsics. The inmost condition is necessary because a round might be part of the body of another loop, and the intrinsic would be part of that loop as well, which is clearly not a marker. IR instructions are attributed to a round by treating the markers as delimiters and collecting all the instructions between a begin and its corresponding end in a list, then saving the list and repeating.

The marker blocks serve no other purpose, and are eliminated once the instructions have been associated the respective rounds. In particular, the markers must be removed before passing the code to the vulnerability analysis, because it requires that the cryptographic routine is loop free, (cfr. Section 2.3.1). Eliminating the markers and stitching the code in a single block is transparent to the analysis, and saves the hassle of dealing with these special cases by hiding them during the CFG exploration.

It should be noted that the identification of rounds is not perfect, although in practice it performs well. It might happen that the last or last few instructions in a round are attributed to the next one, because the optimiser proved that they could be moved across a loop marker without changing the semantic of the program. This happens when the first or last instructions of a round only perform pure, exception-free computation over registers, with no access to memory or call to functions.

The solution would be to make the other code transformations performed by the optimiser aware of the round delimiters, and instruct them to treat round markers as non trespassable barriers, which at the current time has not been attempted. In any case, loop identification is only used for visualisation and debugging purposes, so any imprecision will not affect the soundness of the results.

Listing 2.1: Annotated toy cipher

```
#include "stdint.h"

#define rot32(x,n) (((x) << (n)) | ((x) >> (32-(n))))

inline __attribute__((always_inline))
void keyschedule(uint32_t* subkeys, const uint32_t* key) {
  subkeys[0] = key[1];
  subkeys[1] = key[0];
  subkeys[2] = key[0] ^ key[1];
  subkeys[3] = key[1] << 10 ^ 0xFCEF;
}

void crypt(uint32_t __attribute__((key)) *key,
           const uint32_t __attribute__((plain)) *input,
           uint32_t *output) {
  uint32_t subkeys[4];
  uint32_t tmp = *input;

  keyschedule(subkeys, key);

  #pragma unroll
  for (int n = 0; n < 4; ++n) {
    #pragma cipher round
    {
      tmp ^= subkeys[n];
      tmp = rot32(tmp, 2+n);
    }
  }

  *output = tmp;
}
```

Example of an annotated cipher In order to demonstrate the minimal impact of the modifications required to the source code for the analysis to run, this section reports the complete and annotated implementation of the toy cipher used as example by Agosta et al. in [3]. Besides the `__attribute__` annotations, all subroutines need to be marked for inlining and all loops for unrolling. Optionally, the round operations are enclosed in a lexical block decorated with a `#pragma`. No structural modification is necessary, and any other compiler is able to process the same, unchanged, piece of code. All the unrecognised annotations will be ignored with no error.

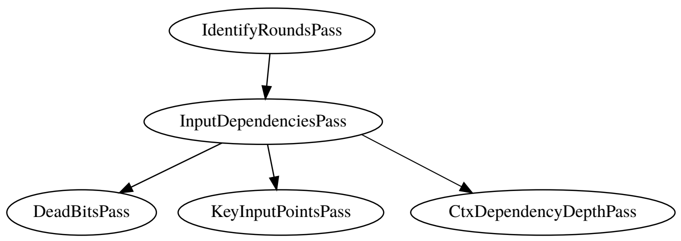


Figure 2.2: Input pre-processing passes

2.3 Features recognition

The first set of passes in the diagram is dedicated to identifying the location and the properties of the components of the encryption routine. First, `IdentifyRoundsPass` exploits the annotations in the source code to detect the rounds of the cipher and associate the instruction to the one they belong to, if any. This information is not strictly required for the analysis, however it will greatly improve the layout of the visualisations introduced in the next chapters. When the rounds have been identified, `InputDependenciesPass` (Section 2.3.1) retrieves the annotated plaintext and key inputs and calculate the dependency depth for each of them, a property which will be then used to select the attack points in Section 2.4.2. The same pass also deduces the ciphertext output point and feed them to `CtxDependencyDepthPass`, which calculates dependency depth in the reverse DFG, following a very similar method. `DeadBitsPass` (Section 2.3.2) identifies the S-Boxes in the cipher, and for each of them calculates the dead bits, another property which will be used both for attack selection and dependency propagation, the latter described in Section 2.5. Finally, `KeyInputPointsPass` (Section 2.3.3) assign the appropriate range of key bits to each key loading point, in preparation for the dependency propagation.

2.3.1 Dependency depth and loop checking

Input points identification and dependency depth Input variables are annotated in the code with special attributes that the frontend translates into intrinsics at the IR level, as detailed in 2.2.3. There are two source code attributes, one for variables containing portions of the plaintext, and another for variables containing portions of the key. In the same way, there are two correspondent intrinsics, denoted as `llvm.crypto.key` and `llvm.crypto.plain` in the algorithms here presented.

This pass scans the IR of the encryption routine, searching for marked instruc-

Algorithm 2.3.1: Input points identification routine

```
Input: $F$ : routine, as sequence of IR instructions
       (implicitly) the DFG of $F$, to get the *users* of each instruction
Output: $PlaintextDependencyDepth : Instruction \to \mathbb{N}$ map
        $KeyDependencyDepth : Instruction \to \mathbb{N}$ map
        $PlaintextInputPoints : Instruction$ set
        $KeyInputPoints : Instruction$ set

1  $PlaintextInputPoints \leftarrow \emptyset$
2  $KeyInputPoints \leftarrow \emptyset$
3  for each $i \leftarrow instructions(F)$:
4      if $i$ has form $llvm.crypto.plain(o)$:
5          for each $u \in users[o]$:
6              $PlaintextInputPoints \leftarrow PlaintextInputPoints \cup \{u\}$
7              TrackPlain $(PlaintextDependencyDepth, u, 0)$
8      else if $i$ has form $llvm.crypto.key(o)$:
9          for each $u \in users[o]$:
10             $KeyInputPoints \leftarrow KeyInputPoints \cup \{u\}$
11             TrackKey $(KeyDependencyDepth, u, 0)$
12 return $PlaintextDependencyDepth$, $KeyDependencyDepth$,
13         $PlaintextInputPoints$, $KeyInputPoints$
```

Algorithm 2.3.2: TrackPlain as used in Algorithm 2.3.1

```
Input: $PlaintextDependencyDepth : Instruction \to \mathbb{N}$ map
       $u$ : Instruction
       $depth \in \mathbb{N}$
Output: $PlaintextDependencyDepth : Instruction \to \mathbb{N}$ map

1  if $PlaintextDependencyDepth[i] \neq \perp$ :
2      return   // Already seen
3  $PlaintextDependencyDepth[i] \leftarrow depth$
4  for each $u \leftarrow users(i)$
5      TrackPlain $(PlaintextDependencyDepth, u, depth + 1)$
6  return $PlaintextDependencyDepth$
```

Algorithm 2.3.3: TRACKKEY as used in Algorithm 2.3.1

```
Input: $KeyDependencyDepth : Instruction \to \mathbb{N}$ map
       $u$ : instruction
       $depth \in \mathbb{N}$
Output: $KeyDependencyDepth : Instruction \to \mathbb{N}$ map
1  if $KeyDependencyDepth[i] \neq \perp$ :
2      return   // Already seen
3  $KeyDependencyDepth[i] \leftarrow depth$
4  if $i$ has form $store\ location, value$:   // Traverse memory bridges
5      for each $u$ having form $load\ location$:
6          TRACKKEY $(u, depth + 1)$
7  else
8      for each $u \leftarrow users(i)$
9          TRACKKEY $(KeyDependencyDepth, u, depth + 1)$
10     return $KeyDependencyDepth$
```

tions and collecting them in two sets, which will be referred to as *PlaintextInputPoints* and *KeyInputPoints*. In parallel with the construction of the two sets, a recursive exploration of the DFG is performed, starting from the nodes corresponding to the input points and following the successors of each one in the def-use chain, taking note of the depth at which each instruction is met, defined as the minimum distance in the DFG between an input point and the instruction under exam. The minimum depth is considered because it represents the best case scenario for an attacker.

The procedure that performs the exploration is shown in Algorithm 2.3.1, in particular Algorithm 2.3.2 and Algorithm 2.3.3 show the procedures used to calculate dependency depth. The DFG is walked breadth first and therefore each use will be visited first at the shallowest depth. All the subsequent visits will happen at a greater or equal depth, and therefore the search can be immediately stopped when an already explored node is seen again.

Memory bridges Some uses of the key material or the plaintext may be mediated by memory accesses, with an instruction storing its result in a memory location and another instruction retrieving it. Most of the occurrences have been eliminated in the intermediate by running a *memory to register* (mem2reg) transformation, which turns this pattern into accesses to a virtual register in the IR. However, this optimisation might not be able to eliminate all memory operations. The surviving ones were successfully handled under an extremely simplified model, which assumed that only one instruction might write to a given memory location and that the address of this location is known at compile time. A mapping is kept between the values and the memory locations they are written to, and each load from the certain location will be assigned the matrices of the correspondent store -- thus the term “bridge”. No attempt was made at handling multiple write points or, worse, pointer aliasing.

Output points deduction and dependency depth In the same way of the plaintext and key input points we just examined, the set *CiphertextOutputPoints* of instructions that output fragments of the ciphertext is constructed, and the dependency depth is calculated. Unlike input points, output points do not need explicit markers and are identified as the instructions that transitively depend both on the key and the plaintext and have no further uses in the routine. Although conceptually separate, the computation of *CiphertextOutputPoints* is performed together with the construction of $KeyDependencyDepth$ and $PlaintextDependencyDepth$ for performance reasons.

Once *CiphertextOutputPoints* is built, the map of dependency depths *CiphertextDependencyDepth* is calculated by performing a recursive exploration of the CFG, this time following the predecessors of each node. Also in this case there might be multiple paths from a ciphertext output point to a given instruction, and again the shallowest depth is recorded, since it represents the best case for an attacker. The CFG is assumed to be loop-free, because the condition has been already tested when calculating input dependencies.

2.3.2 S-Boxes identification and dead bits calculation

Dead bits are bits of the output fixed at either at zero or one throughout all the entries of a table, that might be present in optimised S-Boxes for performance reasons. These bits do not carry any useful information, hence why they are called *dead*. Because no calculation is performed, a lookup in the S-Box table is an opaque operation, and the analysis is not able to detect dead bits only by looking at the input or output of any single operation. However, dead bits in a S-Box can be identified by inspecting the whole table, under the assumption that it is encoded as a pre-computed array of integers.

Each of integer is interpreted as a boolean vector having as size the bit width of the declared cell type for the table, and each vector is filled with the base-2 rep-

Algorithm 2.3.4: Dead bits detection algorithm

```
Input: $Table$ : array of boolean vectors of size $n$, each of them being the
       binary representation of an $n$-bit sized S-Box entry.
Output: $Mask$ : boolean vector representing dead bits in $Table$.
Data: $A$, $B$ : boolean vectors of size $n$, as defined above.
1 $A \leftarrow \mathbf{1}$
2 $B \leftarrow \mathbf{0}$
3 for each $v$ in $Table$:
4     $A \leftarrow A \wedge v$
5     $B \leftarrow B \vee v$
6 $Mask \leftarrow A \vee \neg B$
7 return $Mask$
```

resentation of the associated entry in the S-Box, one bit per element. The detection algorithm operates on this transformed table and is presented in Algorithm 2.3.4. It proceeds by initialising two vectors A and B of the same size of a transformed table entry, one with all 1 and the other with all 0. The entries in the transformed table are then $\wedge$-accumulated in A and $\vee$-accumulated in B. At the end of the procedure, any bit in A which is still at 1 signals a dead bit fixed at 1, as the $\wedge$ of all the bits in that position can be 1 only if all of them are 1. Following the a similar reasoning for $\vee$, any bit in B at 0 signals a dead bit fixed at 0. The dead bit mask is computed as $Mask = A \wedge \neg B$, resulting in a 1 in each position corresponding to a dead bit, and a 0 in each position associated with a live one. Masks for each S-Box are finally stored in a $Box \to Mask$ hashmap called *DeadBits* for later use.

In those cases in which the function stored in the S-Box is computed at runtime, be it precalculated in an actual table or not, the computation is enough for the analysis to detect dead bits and it is not necessary to mark the storage array, if any. Serpent [6] is the example of an encryption algorithm having computational S-Boxes, which are transparently handled by this analysis, being just pieces of computation.

2.3.3 Key bits assignment

For each instruction in the *KeyInputPoints* set calculated as shown in Algorithm 2.3.1, this pass assigns a subrange of the key bits, or *slice*, equal to the size of the instruction result. If two variables load the same key fragment, they both get assigned the same slice, whereas two variables that load different fragments get as-

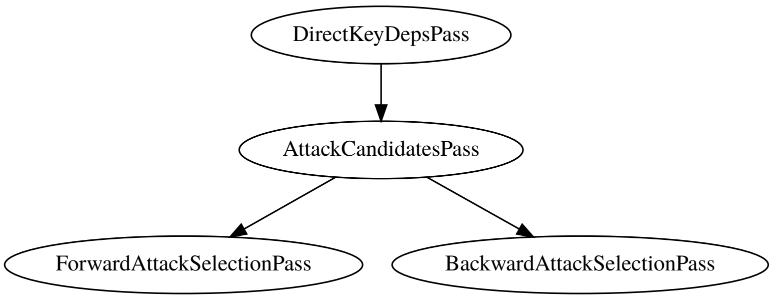


Figure 2.3: Selection passes

signed two non-overlapping slices. In those cases where the key material is stored in an array and each cell is loaded to a different variable, the index of the cell loaded is used to determine which slice to assign (the size of the slices is already given by the type of the array), otherwise the next available one is selected. At this stage, no effort is made to handle overlapping ranges, and none of the algorithms considered actually exhibits one. Nevertheless, a check has been put in place, for future expansions.

This pass also records the total number of key bits assigned, which is used to double check that all the key portions have been correctly identified and processed. The granularity of the analysis is limited by the one of a load operation, which happens at byte level. Therefore, those cases in which not all the bits in a byte are used will result in a reported key size which is rounded up as if all the bits in the byte were in use. For instance, many DES implementations take the 56bit key as an array of 8 octets, of which seven are key material and one is discarded (technically a parity bit). In those cases, the pass will report a key size of 64 bits, instead of 56.

2.4 Selection

The selection phase identifies in the code the best points to probe for mounting a side-channel attack, starting either from the top (plaintext) or the bottom (ciphertext) of the implementation. The assumption made in the analysis is that no attack can be mounted at a lower cost than any one targeting the instructions identified in this phase, meaning that any consideration on the security of this subset represent a lower bound on the security of the whole implementation.

First, `DirectKeyDepsPass` (Section 2.4.1) propagates the key dependencies from the input points to all the instructions in the code, obtaining a mapping between instructions and single bits of the key they depend on. Then, `AttackCandidatesPass` (Section 2.4.2) identifies all the instructions that meet the conditions to mount the attack, and `ForwardAttackSelectionPass` and `BackwardAttackSelectionPass` (Section 13) use the information produced by the other passes to identify the optimal subset to attack in each direction.

2.4.1 Direct dependencies on the key

Starting from the direct key dependencies calculated in Section 2.3.3, the instructions in the cipher are associated to a pair of matrices exploiting the same algorithm that will be used to propagate vulnerable (sub)key dependencies in Section 2.5. The pair of matrices is then $\vee$-summed (elementwise), and the columns of the resulting matrix are $\vee$-reduced themselves to obtain a single vector. This vector has one element for each bit in the key, set either to 1 if the instruction depends on the associated key bit, or 0 otherwise. These vectors are stored in an *Instruction* $\rightarrow$ *Vector* mapping called *KeyDeps*, that will be used for selecting attack points, as explained in Section 2.4.2.

2.4.2 Attack points selection

Attack candidates In all symmetric block ciphers, a subkey (or round key) is derived starting from the key material for each round and mixed with the data being encrypted. In many instances, the calculation that produces the subkeys, called the *key schedule*, can be inverted to recover the cryptographic key starting from one or more round keys. In other cases, the key schedule is not invertible and the key cannot be recovered, however recovering *all* the subkeys is enough to allow a successful decryption of a ciphertext.

When performing a side-channel attack on a block cipher implementation, be it active or passive, the simpler the relation is between the observed quantities (or probes) and a subkey fragment, the easier the fragment can be recovered. For this reason, the instructions that mix the round key with the cipher stream in the outermost rounds are the most likely targets of a side-channel attack. In particular, key mixing points in the first rounds will be the target of a forward attack (starting from the plaintext), while the last few rounds will be considered in a backwards attack (starting from the ciphertext). Which of the two directions is used depends on the nature of the specific attack.

Algorithm 2.4.1: Attack points selection algorithm

```
Input: $AttackCandidates$ : set of the instructions identified as vulnerable
       $Depth[v]$ : either $CandidateForwardDepth$ or
       $CandidateBackwardsDepth$, depending on the direction
       $KeyDeps[v]$ : direct key dependencies as calculated in 2.4.1.
       $KeyBitsCount$ : number of key bits to be covered, as found in 2.4.1.
Output: $AttackSet$ : set of the instructions containing the target subkey bits,
        or $\perp$ if there is no viable assignment
Data: $assigned$, $toassign$ : boolean vectors of size $KeyBitsCount$
Global: $FreePicks$ : initially empty set of triples $(i, picked, left)$ where $i$ is
        an instruction of the DFG, $picked, left$ are boolean vectors of
        length $KeyBitsCount$ representing the subkey bits assigned, and not
        assigned to $i$ as a result of an unconstrained choice

$assigned \leftarrow \mathbf{0}$
$AttackSet \leftarrow \varnothing$
for $d \leftarrow 1 \ldots \max(d | (i,d) \in Depth)$:
    $toassign \leftarrow \mathbf{0}$
    for each $v \in AttackCandidates$ s.t. $Depth[v] = d$:
        $available \leftarrow KeyDeps[v] \wedge \neg assigned$
        $Preferred \leftarrow \textsc{GreedyOptimize}(available, assigned)$
        if $|available \vee preferred|_1 > 0$:
            $toassign \leftarrow toassign \vee \textsc{CapAssigned}(available, preferred, v)$
            $AttackSet \leftarrow AttackSet \cup \{v\}$
    if $|assigned|_1 = KeyBitsCount$:
        return $AttackSet$
return $\perp$
```

The set of subkey mixing points is called *AttackCandidates* and is more formally defined as the set of the instructions that depend on a key fragment but not on the plaintext, such that at least one of their uses transitively depends on a plaintext fragment. Two mappings $Instruction \to \mathbb{N}$ are attached to the set, *CandidateForwardDepth* and *CandidateBackwardsDepth*, one carrying for each instruction in the set the distance from the nearest plaintext fragment in the CFG as calculated in Section 10, and the other the distance from the nearest ciphertext fragment in the reverse CFG as calculated in Section 10.

Attack points selection The *ForwardAttackPoints* (resp. *BackwardsAttackPoints*) set is defined as the smallest subset of *AttackCandidates* from which the entire key can be reconstructed, such that the instructions are located at the smallest

CandidateForwardDepth (resp. *CandidateBackwardsDepth*) possible, in order to maximise the likelihood of a successful attack against them. The selection algorithm was first introduced in [3] and is reported in Algorithm 2.4.1. It works by identifying the key bits each $v \in \mathit{AttackCandidates}$ depends on, and then selecting the smallest subset such that covers all the key bits and such that each partial covering is performed at the shallowest depth possible. Each key bit might be covered by more than one instruction, in which case the coverage is not optimal, but no key bit should be left uncovered.

Algorithm 2.4.1 makes use of two auxiliary procedures. GREEDYOPTIMIZE checks for the existence of an element i of *FreePicks* such that some of its left subkey bits l cannot be assigned in the current *available*, and some of its taken t can. If this condition is met, i is set to cover those l subkey bits, t is removed from coverage, *assigned* and *FreePicks* are updated accordingly, and *preferred* is recomputed considering those t bits. Instead, CAPASSIGNED compute the result as a bit vector with all the assignments in *preferred* plus $N - |preferred|_1$ bits from *available*, where N is the size in bits of the result of the instruction v.

Algorithm 2.4.2: Subkey to key matrix construction

```
Input: $KeyDeps[v]$ : direct key dependencies as calculated in 2.4.1
       $AttackPoints$ : set of selected attack points, as calculated in 2.4.2
Output: $SubkeyToKey$ : matrix having one row for key bit, one column for
        subkey bit.
1 $base \leftarrow 0$
2 for $v \in AttackPoints$:
3     $size \leftarrow size(typeof\ v)$   // Size of the result of v in bits.
4     for $i \leftarrow base \ldots base + size$-1:
5         $(SubkeyToKey)_i = (KeyDeps[v])_i$
```

Subkey to key matrix Finally, a subkey to key boolean correlation matrix is built from the selected attack points using the algorithm presented in Algorithm 2.4.2. Ranges of adjacent columns correspond to the respective bits of each selected attack point -- which are subkey bits, and each row is associated to a key bit. A cell contains 1 if the subkey bit associated with the column depends on the key bit associated with the row, 0 otherwise. This matrix will be used to reconstruct key dependencies and propagate throughout the all the instructions of interest, as described in Section 2.5.

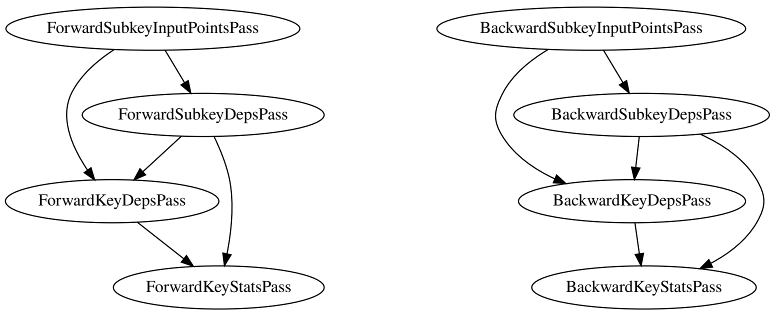


Figure 2.4: Dependency propagation passes detail.

2.5 Dependency propagation

The attack points that have been selected in the previous phase are all points where a subkey, derived from the key, is mixed to the cipher stream. In the attack scenario considered, the key will be recovered from these subkey fragments, i.e. assuming that the derivation function (or keyschedule) is invertible. If the cipher has a non-invertible keyschedule, it will be necessary to recover all the subkeys and the attack will necessarily be harder. For this reason, the invertible case represent a lower bound on the cost of an attack.

First, `ForwardSubkeyInputPointsPass` assigns a pair of vulnerability matrices (cfr. Section 2.5.1) to each key input point. This information is propagated to all the instructions in the cipher in the `ForwardSubkeyDepsPass`. Once the propagation is complete, `ForwardKeyDepsPass` calculates key dependency matrices from subkey dependency matrices using the subkey-to-key matrix built in Section 5. Finally, `ForwardKeyStatsPass` calculates various statistics about the matrices, including the vulnerability indices that will be used to identify where to apply countermeasures, as described in Section 2.6 and Section 2.8. The same flow is replicated for the backwards propagation by the equivalent `Backward` passes.

2.5.1 The vulnerability (dependency) matrices

At the core of the vulnerability analysis sits a representation first introduced by Agosta et al. in [3], the *vulnerability matrix,* or *dependency matrix*. Each variable involved in the computation is associated to a pair of matrices $n \times m$, with n number of key/subkey bits the dependencies are referred to, and m bit width of the result of the instruction. Each cell in the matrix will contain a 1 if the corresponding bit of the result depends linearly (resp. non-linearly) on the key bit assigned to that row, a 0 otherwise. The pair of matrices carry the linear and non-linear dependencies from an encryption key or round key (subkey), and will be referred to as the *linear* and *non-linear* matrices for brevity, or with the even more concise notation L and NL.

The diagonal of the L matrix associated to an instruction which first meets a piece of key material (or round key) as identified in Section 2.4.2, is initialised with the correspondent vector of assigned key bits. Dependencies are then propagated at fixed point, using the dataflow framework presented in Section 1.5 and the rules introduced in the rest of this section. These rules depend on the operation being considered and the direction in which the analysis proceeds, either forward (operands to results, plaintext to ciphertext), or backwards (results to operands, ciphertext to plaintext). Each rule here described yields a pair of contribution matrices, which are respectively $\vee$-summed to the pair associated to the result.

With the backward and forward subkey dependency matrices calculated, the correspondent subkey-to-key matrix (cfr. Section 5) is used to reconstruct a pair of key dependency matrices for each instruction analysed. This pair, called *mediated key dependency matrices*, captures which key bits each bit in a variable depends from, an information which is directly correlated to the vulnerability of each bit in the associated variable to side-channel attacks.

2.5.2 Forward propagation

Forward dependencies are propagated from operands to results, and therefore the matrices for each instruction are constructed by examining those of its operands. The calculation proceeds in the same direction of the computation, i.e. from an instruction towards its uses, the propagation rules for the classes of instructions are discussed in the following paragraphs and summarised in Figure 2.5 and Figure 2.6. These rules are the same ones used to propagate direct dependencies on the key, which are needed by the attack points selection algorithm presented in Algorithm 2.4.1.

Linear boolean operations: `XOR` *(Figure 2.5)* The linear matrix of a `XOR` is given by the $\vee$-sum (elementwise) of the linear matrices of its operands, and likewise the non-linear matrix from the $\vee$-sum of the non-linear matrices of its operands. Only the operands that are instructions are considered, constants do not yield any effect. In fact, the result of combining via `XOR` a variable bit with a constant bit is either

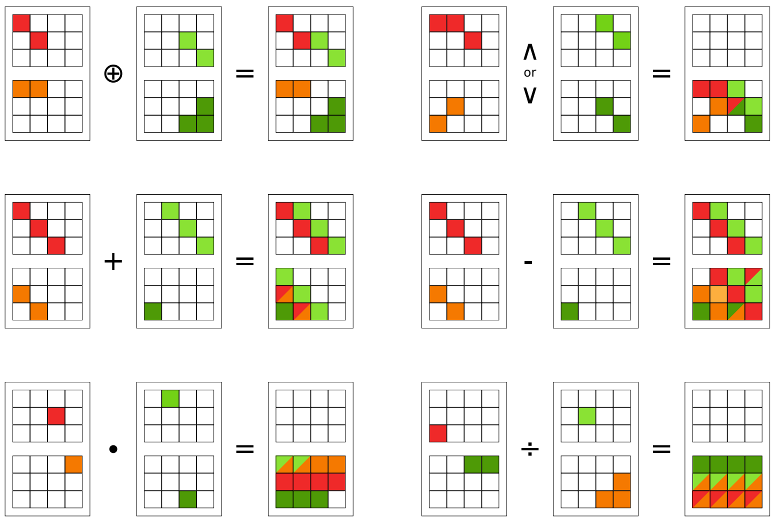


Figure 2.5: Forward propagation rules exemplified: arithmetic and boolean.

the variable bit or its negation, and it both cases the result depends entirely on the former.

Non-linear boolean operations: `OR`, `AND` *(Figure 2.5)* For both operations, the non-linear matrix is given by the $\vee$-sum (elementwise) of both the linear and non-linear matrices of their operands that are instructions themselves. The linear matrix is null, as these two operations have a purely non-linear effect.

Constant operators to `AND` and `OR` produce a masking effect. In particular, the result of an `AND` will have a zero in each bit in the same position of a zero in the constant. Likewise for `OR`, all the 1 bits in the constant operand will produce a 1 in the same position of the result. In both cases, the bit in question has no relation with the input variable, and therefore the respective column in the result matrices is set to zero.

Arithmetic operations: `ADD`, `SUB` *(Figure 2.5)* Both `ADD` and `SUB` operations happen in two phases: first a linear sum of the input, then the propagation of carries (resp. borrows). The first phase is exactly the XOR above, so the linear matrix of the result is initialised to the $\vee$-sum of the linear matrices of the operands, and likewise for the non-linear matrix of the result with the non-linear matrices of the operands. Again, constant operands are not considered in this first step.

In the second phase, each column of the non-linear matrix of the result is $\vee$-

Algorithm 2.5.1: Forward propagation through `ADD`

```
  Input: $(A_L, A_{NL}), (B_L, B_{NL})$ : dependency matrix pairs of $A$ and $B$
  Output: $(C_L, C_{NL})$ : dep. matrix pair of $C = A + B \mod 2^w$ ($w$ wordsize)

  // Linear sum
1 $C_L \leftarrow A_L \vee B_L$
2 $C_{NL} \leftarrow A_{NL} \vee B_{NL}$
  // Carry propagation
3 $C_{NL} \leftarrow C_{NL} \vee (C_L \ll 1)$   // $C_L \gg 1$ for SUB
4 $temp \leftarrow (C_{NL})_0$
5 for $i \leftarrow 1 \ldots cols(C_{NL}) - 1$:
6     $(C_{NL})_i \leftarrow (C_{NL})_i \vee temp$
7     $temp \leftarrow (C_{NL})_i$
8     return $C_L, C_{NL}$
```

summed all the columns before it (towards the LSB) for `ADD`, or after it (towards the MSB) for `SUB`. This reflect the ways carries and borrows propagate their non-linear effect, i.e. from LSB to MSB in the case of an `ADD`, and from MSB to LSB in the case of a `SUB`.

The pseudo-code for these two operations is presented in Algorithm 2.5.1.

Algorithm 2.5.2: Forward dependency propagation algorithm for `MUL`

```
   Input: $(A_L, A_{NL}), (B_L, B_{NL})$ : dependency matrix pairs of $A$ and $B$
   Output: $(C_L, C_{NL})$ : dep. matrix pair of $C = A \cdot B \mod 2^w$ ($w$ wordsize)

1  $A_{LNL} \leftarrow A_L \vee A_{NL}$
2  $B_{LNL} \leftarrow B_L \vee B_{NL}$
3  for $j \leftarrow 1 \ldots cols(C_{NL})$-1:
4      for $i \leftarrow 1 \ldots j$-1:
5          $(C_{NL})_j \leftarrow (C_{NL})_j \vee (A_{LNL})_i \vee (B_{LNL})_{j-i}$
6  $C_L \leftarrow \mathbf{0}$
7  $temp \leftarrow (C_{NL})_0$
8  for $i \leftarrow 1 \ldots cols(C_{NL}) - 1$:
9      $(C_{NL})_i \leftarrow (C_{NL})_i \vee temp$
10     $temp \leftarrow (C_{NL})_i$
11 return $C_L, C_{NL}$
```

Arithmetic operations: `MUL`, `DIV` *(Figure 2.5)* A multiplication is seen in binary schoolbook form, which only involves `ADD`s and single-bit multiplications for di-

Algorithm 2.5.3: Optimized forward propagation algorithm for `MUL`

```
Input: $(A_L, A_{NL}), (B_L, B_{NL})$ : dependency matrix pairs of $A$ and $B$
Output: $(C_L, C_{NL})$ : dep. matrix pair of $C = A \cdot B \mod 2^w$ ($w$ wordsize)
1 $C_L \leftarrow \mathbf{0}$
2 $C_{NL} \leftarrow A_L \vee A_{NL} \vee B_L \vee B_{NL}$
3 $temp \leftarrow (C_{NL})_1$
4 for $i \leftarrow 1 \ldots cols(C_{NL}) - 1$:
5     $(C_{NL})_i \leftarrow (C_{NL})_i \vee temp$
6     $temp \leftarrow (C_{NL})_i$
7 return $C_L, C_{NL}$
```

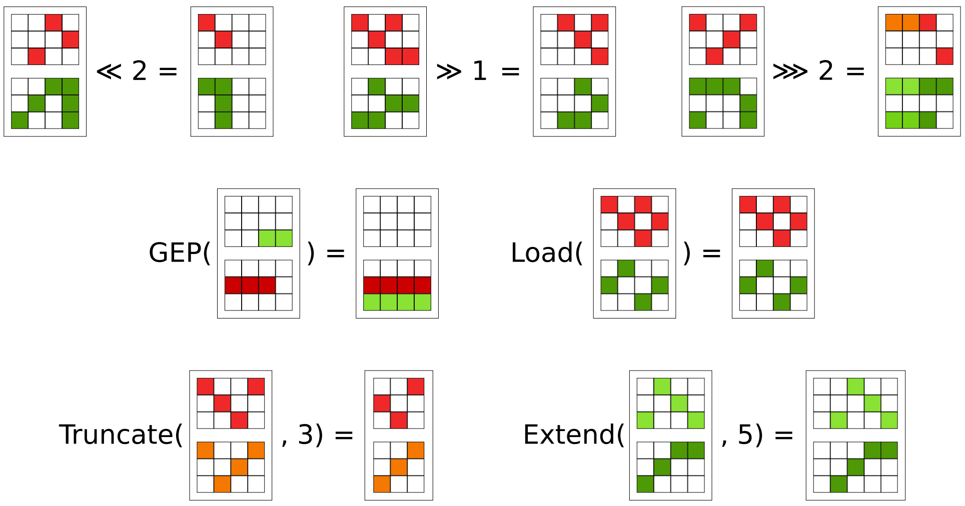


Figure 2.6: Forward propagation rules exemplified: shifts, casts and memory.

gits, which effectively are `AND` operations. The propagation rules already defined for these two operations are combined for the appropriate digits as shown Algorithm 2.5.2. With a reordering of the operations, the procedure can be optimised, obtaining Algorithm 2.5.3.

The effects of a division are propagated through a *conservative approximation* by $\vee$-summing the columns of the linear and non-linear matrices of both operands, and then using the result column to fill the non-linear matrix of the result, while leaving the linear matrix empty. This assumes that the effect of a `DIV` is the total, non-linear, diffusion of the input on the output.

Shifts: `SHL`, `LSHR`, `ASHR` *(Figure 2.6)* In case of a shift by a constant, the matrices of the result are the ones of the other operand, with the columns respectively shifted by that constant. In case of an arithmetic (right) shift, the column corresponding to the most significant bit still alive is used to fill all the other null columns towards the new MSB.

If the shift amount is variable, a left shift $A \ll B$ is treated as the multiplication $A \cdot 2^B$, and the right shift $A \gg B$ as the division $A/2^B$, and both are handled as explained before.

Truncations and extensions *(Figure 2.6)* While never found in theoretical discussions, real world implementations often require the conversion between types with different widths through a truncation or an extension. The former is treated by dropping the columns associated with the truncated bits, and likewise zero extensions by appending null columns. Signed extensions use the column corresponding to the old MSB to pad the newly inserted columns, instead of a null vector.

Table lookups *(Figure 2.6)* The only applications of table lookups within the implementation of a symmetric cipher are assumed to be the S-Boxes, which provide the confusion step in a cipher round. Their effect is therefore *approximated* as a non-linear diffusion of the linear and non-linear contributions of all the non-constant indices, typically only one. In order to achieve this effect, the columns of the L and NL matrices of the lookup indices are $\vee$-summed (elementwise), and the resulting column vector is used to fill the non-linear matrix associated with the instruction. The linear matrix of the result is left empty. Each column of the NL matrix occupying the same position of a 1 in the the vector associated to the S-Box by *DeadBits* (cfr. Section 2.3.2) is replaced by a column of zeroes.

In the case of LLVM IR, the address calculation and the transfer from memory are two distinct operations. The first one is performed by the `getelementptr` instruction (or GEP for brevity), which, in the case of (possibly nested) arrays, takes a base pointer and an ordered sequence of indices, and returns the address for the indexed element. The second operation is carried out with a `load` instruction, which takes an address (possibly calculated with a GEP), fetches the value from memory and returns it. This split is handled in the analysis by associating the diffused matrix to the result of the GEP and then copying the matrices of the GEP to the `load`.

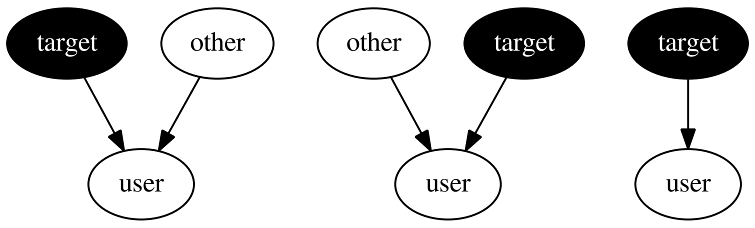


Figure 2.7: Backwards analysis naming conventions.

Other operations It has been observed in the experimental phase that the compiler might optimise certain routines to use a byte swap intrinsic operation, which consists in the reordering of the bits in the variable in groups of 8 (octets). This operation is handled by mirroring the bit rearrangement on the columns of the input matrices and assigning the transformed L and NL to the L and NL matrix of the result.

2.5.3 Backwards propagation

Backwards dependencies are propagated from an instruction back to the operands, i.e. from users to uses, and this is the reason for the backwards bit in the name. Calculating backwards dependencies effectively means inverting the effect of an instruction, which not always possible. In fact, most of the instructions examined in the following sections will be treated with *conservative approximations*, and for this reason the backwards analysis is less precise than the forward one. This loss of precision due to non-invertible functions is not a defect of the analysis, because the same is experienced by an attacker, who will have to make more complex hypothesis. The rules are summarised in Figure 2.8 and Figure 2.9.

Most of the examined instructions are binary operators, and both operands are taken into account when calculating backwards dependencies. In order to avoid any confusion when talking about a user and its operands, I will adopt the terminology exemplified in Figure 2.7, where the *target* node (marked in black) is the instruction whose vulnerability matrices are being calculated. If the *user* is a unary operation, the *target* is the only operand and there is no *other*.

An instruction might and, in most cases, will have multiple users. For this reason, each user gives a *contribution* to the matrices of the target rather than entirely determining it, and these contributions are $\vee$-accumulated (elementwise) to give the L and NL matrices of the target. The titles of the following sections refer

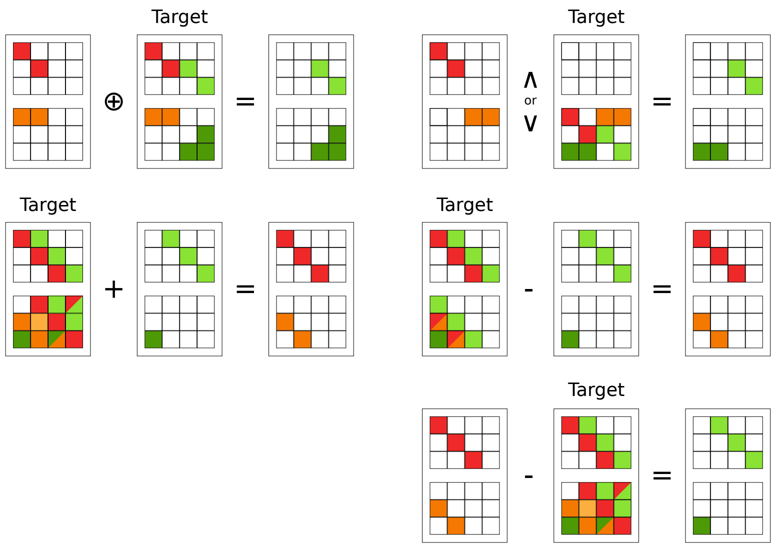


Figure 2.8: Backwards propagation rules exemplified: arithmetic and boolean.

to the type of the *user*, and not to the one of the *target*.

Linear boolean operations: `XOR` *(Figure 2.8)* In case of a `XOR`, we have $user = target \oplus other$, and this relation can be rewritten as $target = user \oplus other$. The L and NL contributions to the *target* are therefore calculated as in the case of a forward `XOR` (cfr. Section 2.5.2) by $\vee$-summing the linear (resp. non-linear) matrices of the *user* and the *other* to give the contribution to the linear (resp. non-linear) matrix of the *target*. If *other* is a constant, the contributions are directly given by the linear and non-linear matrices of the *user*.

Non-linear boolean operations: `AND`, `OR` *(Figure 2.8)* `AND` and `OR` are the first examples of operations whose effects cannot be precisely inverted. This is due to the fact that these are not bijective functions.

If *other* is an instruction, the L and NL dependencies of *user* and *other*, and the existing L dependencies of the *target* are $\vee$-summed together to give the non-linear contribution to the *target* and the linear contribution is left unmodified. Instead, if *other* is a constant, the linear and non-linear matrices of the *user* are masked as it happened in forward propagation (cfr. 2.5.2) to give the linear and non-linear contributions.

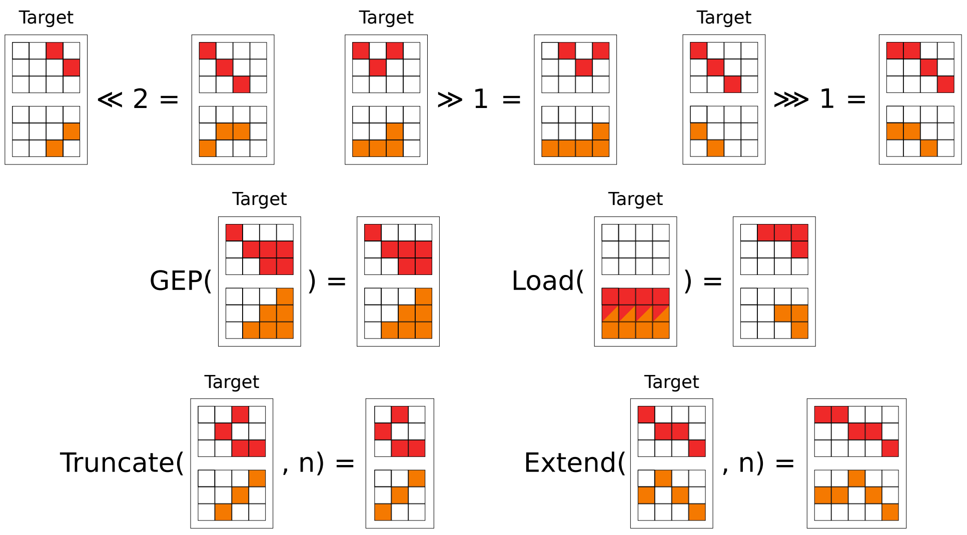


Figure 2.9: Backwards propagation rules exemplified: shift, cast and memory.

Arithmetic operations: `ADD`, `SUB` *(Figure 2.8)* Addition and subtractions can be perfectly reversed by observing that if $user = target + other$, then $target = user - other$. The dependencies matrices are then calculated using the same rules of a forward subtraction, as explained in Section 8. The rules for a subtraction are similar, however a distinction between two cases must be made, because this operation is not commutative:

$$user = other - target \Rightarrow target = other - user \tag{2.1}$$
$$user = target - other \Rightarrow target = user + other \tag{2.2}$$

Case (1) is analogous to the backwards addition, while case (2) is handled as a forward addition (cfr. again Section 8).

Shifts: `SHL`, `ASHR`, `LSHR` *(Figure 2.9)* If the target is the shift amount, then the shift operation yield no backwards contribution, since the only effect of the target on the user is a rearrangement of the contributions of the other operand, which are otherwise left unmodified.

Instead, if the target is the shifted quantity, two cases are possible.

Shift by constant amount. The linear and non-linear contributions of a shift by constant are respectively obtained by shifting the L and NL matrices of the user of the same amount in the opposite direction. The bits that are discarded when calculating the result of the user cannot be recovered in any way, so the newly inserted columns are zero vectors regardless of the fact that the shift is logical or arithmetic.

Shift by variable amount. The linear contributions of a shift by variable are given by the L matrix of the user, while the calculation of the non-linear contributions are more involved: first the columns of the L and NL matrices of other operand are $\vee$-summed (cell by cell), and the result is used to fill a matrix of the same size of the L and NL of the user. This matrix is then $\vee$-summed to the NL matrix of the user, finally giving the non-linear contribution to the target.

Truncations and extensions *(Figure 2.9)* Other than removing or adding empty columns, truncations and extensions preserve the bits of their only operand. The L and NL contributions are directly obtained from the L and NL matrices of the user.

When reversing an extension, the matrices are first truncated to the size of the target. Conversely, the matrices of a truncation user are extended to match the size of the target. The columns added when reversing a truncation are always empty, since the bits lost in the operation cannot be recovered, and consequently a backwards dependency cannot exist. In the same way, no special consideration is necessary for the columns eliminated when reversing an extension. In fact, the dropped columns are either null for an unsigned extension, or equal to the column of the MSB before the extension -- and in both cases have no effect on the target.

Table lookups *(Figure 2.9)* Table lookups are assumed to be used for S-Boxes only, as it was the case with forward propagation. A `load` operation is reversed by $\vee$-summing (elementwise) the linear and non-linear matrices of the user, masking away columns corresponding to dead bits (see Section 2.3.2) and finally $\vee$-reducing the columns and using the resulting vector to fill the NL matrix of the target, while the L matrix is left untouched. Instead, `getelementptr` is reversed by copying the L and NL matrices of the user back to the target.

Other operations Byte swap operations are handled in backwards propagation by remembering that these are involutive, unary operations. Since $user = byteswap\,(target)$, we have that $target = byteswap\,(user)$, and the L and NL contributions to the target are obtained by rearranging the columns of the L and NL matrices of the user in the same order in which the corresponding bits of the user value are moved.

2.6 Vulnerability index calculation

At this point in the analysis, each instruction in the block cipher implementation has been associated with a forward and a backwards pair of matrices. From these matrices, a vulnerability index is synthesised to quantify the weakness of an instruction against side-channel attacks and decide where countermeasures need to be applied, which will be the object of Section 2.8.

As explained in Section 1.1 and Section 1.2, the side-channel attacks that can be thwarted by algorithmic methods involve making hypothesis on values, and the verification of these hypothesis becomes exponentially more difficult in the number of key bits each bit depends on. This fact allows us to define a vulnerability index for each instruction as the smallest number of key bits a bit in that value depends from. The analysis acts on a bit level, thus each bit in the variable has a separate vulnerability index, but software implementations of masking schemes act at a register level, so we associate a security margin to each variable, defined as the minimum between the security margins of the individual bits. The lower the index, the more vulnerable the instruction. 0 represents a special case, in which the associated bit does not depend on the key at all, and therefore is not vulnerable.

Four vulnerability indices are calculated under different attack models, and the overall vulnerability index is given by the minimum among them:

Classic passive attacks The L and NL matrices are $\vee$-combined (cell by cell), and the minimum among the sums by column is taken. The same procedure is repeated for the forward and the backward pairs, and the minimum between the two is the final result.

Advanced passive attack The NL matrix is summed by column, and each result is incremented by 1 if the sum of the corresponding column of $\neg C_{NL} \wedge C_L$ is larger than 1. The same procedure is repeated for the backward and forward pair also in this case, and the minimum between the two is taken.

Differential fault analysis The backwards NL matrix is summed by column, and the minimum of the sums is taken as index. Only the NL has to be considered

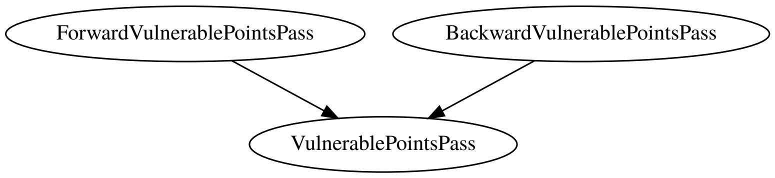


Figure 2.10: Instruction filtering and aggregation passes detail.

because the differential information on linear key contributions is not usable in this attack model, and only the backwards one, because a DFA attack always starts from the output of the cipher.

Safe error attacks Under the assumption that a single bit stuck-at fault is introduced, a safe error attack reveals the value of that bit during the execution. The vulnerability index is therefore the same of a passive attack.

2.7 Instruction filtering and aggregation

The last set of passes, shown in Figure 2.10, identifies the vulnerable instructions that will need to be protected. Once each instruction has been assigned a forward and a backwards vulnerability index as shown in Section 2.6, the vulnerable instructions in each direction are identified by `ForwardVulnerablePointsPass` and `BackwardVulnerablePointsPass`. Instructions that are marked as either forward or backwards vulnerable (or both) are then collected in a set by the last pass, `VulnerablePointsPass`, ready for the countermeasure application phase.

Vulnerable instructions identification The identification of vulnerable instructions starts from the forward (resp. backwards) attack points identified in Section 13 and proceeds at fixed point according the respective rule:

Forward rule an instruction is forward vulnerable if its vulnerability index is below the specified threshold, *and* one of its operands is already marked as vulnerable *or* at least one operand is part of the keyschedule.

Backwards rule an instruction is backwards vulnerable if its vulnerability index is below the threshold, *and* it is a direct user of a subkey fragment *or* one of its uses depends on the plaintext *and* is marked as vulnerable.

Table 2.1: Masking countermeasures used in the implementation.

Instruction	Masking order	
	1	≥ 2
ADD, SUB	Coron [24] (32, 64 bits) Karroumi [45] (8, 16 bits)	Higher Order Coron [23]
NOT, XOR	ISW [42]	
AND, OR	ISW (+ De Morgan rule) [42]	
SHL, ASHR, LSHR (by constant amount)	Shift each share singularly	
Truncate, Extend	Resize each share singularly	
Table lookup	Coron randomisation [21]	

The vulnerability threshold might be set to $KeySize + 1$, in which case all potentially vulnerable instructions are masked, or to a lower threshold found as the result of a trade-off. A threshold placed at *KeySize* is particularly interesting, in that it selects only those instructions which depend on less than the entire key. Because of the diffusion properties of well-engineered symmetric ciphers, such a security threshold is reached after few iterations both from the top and the bottom, leaving the innermost rounds unmarked. This allows for a low protection overhead while retaining the same computational security margin against both side channel attacks and classical cryptanalysis.

2.8 Automatic countermeasures application

The instructions marked as vulnerable at the end of the analysis (see Section 2.7) are protected using the masking countermeasures described in Section 1.3. The chosen transformation(s) for each operation of interest are summarised in Table 2.1. In some cases, there is a single algorithm and the choice is straightforward. For others, one or more algorithms have been chosen based on the performance and memory impact considerations which can be found in Section 1.3.

I found no mention in literature of an efficient way to mask a shift by variable amount, and therefore this case is not handled. There is only one cipher which appears to be using such operation, CAST5 [1], which cannot be protected but has been analysed nevertheless. The rarity of this operation in block cipher implementations might also explain the lack of exploration is this regard.

Vulnerable instructions are processed in program order, so that operands are always protected before their uses. An associative table is kept between unmasked instructions and their masked equivalents. Every time an unmasked instruction appears as an operand to an instruction to be masked, the instruction is replaced by the masked equivalent, and the masked operand is retrieved and directly wired to it. If a non-masked use of a masked operation appear, an decoder is inserted and the decoded value is wired to it. Conversely, an encoder is wired to masked instructions having a unmasked operand. The encoders and decoders are kept in the same lookup map of the transformed operations, so that they can be transparently wired to other uses.

Algorithm 2.8.1: Masking

```
for each $I \in VulnerableInstructions$:                      // Sorted by dominance
    $Masked \leftarrow$ CreateMasked$(I, Masked)$
for each $I \in VulnerableInstructions$:                      // Sorted by dominance
    if $Uses[I] \neq \emptyset$:
        $D \leftarrow$ CreateDecoder$(I)$
        replace all unmasked uses of I with D
for each $I \in VulnerableInstructions$:                      // Sorted by reverse dominance
    if $Uses[I] = \emptyset \wedge Masked[I] \neq \bot$:
        drop $I$ from the CFG
    else if $I$ has not form store or return:
        report failure
```

2.8.1 Loops and inlining in the transformed code

The ISW AND transformation and Coron's table randomisation both tend to produce large amounts of code, a fact which increases the register pressure and inflates considerably the size of the protected binary. The choice I made was to unroll the ISW AND loops and insert the code in place, since the two nested loops (see Section 10 for the algorithm) are bounded by the masking order, which tend to be small. In fact, attacks above the 3^{rd} order are believed to be extremely challenging, if possible at all. A double loop is likely to take up more time because of the condition evaluation and branching, not to mention a function call.

On the other hand, Coron's transformation entails a nested loop with $\Theta(|S|)$ iterations in total, where $|S|$ is the dimension of the lookup table being masked -- usually a large number. In addition to the large number of rounds, the body of the loop is large in itself and includes some functions calls -- which generate additional context save and restore operations. My decision was not to unroll the loops and to generate a subroutine for each S-Box, replacing unmasked lookups with function calls.

Chapter 3

Experimental Evaluation

The methods described in the previous chapters have been applied to a range of block ciphers, either in active use or of historical relevance, in order to confirm the feasibility of the approach on two popular embedded architectures, ARMv7 and MIPS32. The internal structure, design considerations and main applications of each cipher considered are presented in Section 3.1, while the visualisations produced by the analysis tool are reported in Section 3.2, together with a discussion of the properties highlighted by each graph. Finally, Section 3.3 contains the plots of the benchmark data and a commented overview of the main results. The complete data set is contained in Appendix A, for reference.

3.1 Block ciphers analysed

The first set of block ciphers analysed is composed by the ISO/IEC 18033-3:2010 [43] portfolio, which comprises:

DES/TDEA Former NSA-approved standard, developed at an IBM facility and published as FIPS in 1977 [61], now superseded by AES. Its main peculiarity is the bit-oriented design, which results in inefficient software implementations. It has a Feistel network structure with 16 rounds and operates on a 64bit block, making it potentially vulnerable to collision attacks. The original configuration has a 56bit key and was successfully attacked by bruteforce in multiple instances, making it unsuitable for real-world use. For this reason, two variants has been introduced in

Table 3.1: Overview of the analysed block ciphers

	Key size	**Block size**	**Rounds**	**Status**	**Use**	**Structure**
	40		12	B	–	
CAST5	80	64	16	A	–	Feistel
	128		16	S	G	
DES	56		16	B	–	
2TDEA	112 (80)	64	$2 \cdot 16$	A	–	Feistel
3TDEA	168 (112)		$3 \cdot 16$	A	E	
HIGHT	128	64	32	S	L	Feistel
MISTY1	128	64	8 ($4n$)	W [8]	L	Nested Feistel
SEED	128	64	16	S	G	Nested Feistel
AES (Rijndael)	128/192/256	128	10/12/14	S	G	SPN
Camellia	128/192/256	128	18/24/24	S	G	Feistel
XTEA	128	64	64 (n)	A	L	Feistel
Noekeon	128	128	16	S	L	SPN
Serpent	128/192/256	128	32	S	G	SPN
Speck	128/192/256	128	32/33/34	S	L	ARX
Simon	128/192/256	128	68/69/72	S	L	Feistel

Security	**S**trong ($\geq$ 128 key bits), **A**cceptable ($\geq$ 80), **W**eak ($\geq$ 60), **B**roken ($<$ 60)
Use	**G**eneric, **L**ightweight, **LE**gacy (superseded)

Table 3.2: IR instruction classes used by the implementations under exam

	Operand sizes in bits					
	$\oplus, \neg$	$\wedge, \vee$	$\boxplus, \boxminus$	**Lookups**	**C. Sh/Rot**	**V. Sh/Rot**
AES (Rijndael)	8, 32	32	–	8	32	–
Camellia	8, 32, 64	32, 64	–	8	32, 64	–
Serpent	32	32	–	–	32	–
Speck	64	–	64	–	64	–
Simon	64	64	–	–	64	–
Noekeon	32	32	–	8	32	–
HIGHT	32, 64	–	32	8	8, 32, 64	–
XTEA	32	–	32	–	32	–
MISTY1	16, 32	16, 32	–	8, 16	32, 64	–
SEED	32, 64	32, 64	32	32	32, 64	–
DES						
2TDEA	32	32	–	16, 32	32	–
3TDEA						
CAST5	32	32, 64	32	32	32, 64	32

1999 [62], 2TDEA and 3TDEA, which are internally structured as a sequence of 3 DES routines whose keys K_1, K_2, K_3 compose the TDEA key. In particular, 2TDEA has $K_1 = K_3$ and K_2 independent, thus a key size of $2 \cdot 56 = 112$bit and 3TDEA three independent keys, resulting in a key size of $3 \cdot 56 = 168$bit. The *effective* security margin of these two configurations is 80bit for 2TDEA and 112bit for 3TDEA because of a meet-in-the-middle attack. 3TDEA thus offers an acceptable security level, although the performance issues of DES, worsened by the triplication of the routine, discourage its adoption outside legacy contexts.

AES (Rijndael) Designed as the replacement for DES, Rijndael was published as the Advanced Encryption Standard in 2002 [28,64]. It operates on 128bit blocks, has a substitution-permutation-network structure and a variable key size (128, 192 or 256 bit), which determines the number of rounds (10, 12 and 14, respectively). The design allows for efficient implementations both in software and hardware. Because of the large amount of scrutiny it has received over the years, AES is considered a very secure option in all its configurations.

Camellia Designed by Mitsubishi Electric and NTT of Japan in 2000, it has been approved for use by the European Union's NESSIE program and the Japanese CRYPTREC. It is currently covered by a patent, but a royalty-free license is available for all implementors and the design is discussed in RFC-3713 [57]. Camellia has a Feistel network design with the addition of intermediate transformation rounds, called FL functions, and input and output key whitening. It operates on a 128bit block and has 3 key sizes, 128, 192 and 256 bit, which in turn determine the number of rounds -- 18 in the first case and 24 in the others. In particular, Camellia 192 expands the key to 256 bits and then uses the same algorithm of Camellia 256. Both NESSIE and CRYPTREC regard the security level of this cipher to match the one of the AES.

MISTY1 Designed for Mitsubishi Electric in 1995, it was included in the NESSIE portfolio of the European Union in 2003 [67] and by Japan own CRYPTREC program in the same year, from which it was dropped in 2013. It has a nested Feistel network structure with a 128bit key, a 64bit block and a variable number of rounds -- 8 recommended [58], or any multiple of 4. It also have a characteristic asymmetric design with one 7-to-7bit and one 9-to-9bit S-Box. A successful attack against the full cipher was published in 2015 [8, 77], and while still outside the realm of the feasible (2^{64} chosen ciphertexts and $2^{69.5}$ time in the fastest case), it is reasonable to assume that the algorithm will be practically broken in the near future. The cipher is covered by a patent, although a non-profit license is available and the algorithm is documented in RFC-2994 [58].

SEED Developed in 1998 by the Korea Information Security Agency (KISA) for securing national Internet communications, since 40bit export ciphers were deemed insufficient at the time. SEED, which is is rarely used outside South Korea, has a 128bit block size and operates on a 128bit block with 16 rounds and a nested Feistel network structure. It is comparable to MISTY1, although it has a more complex key schedule and two options for the round structure, one with 2 8-to-8bit S-Boxes and another with a simpler round function but 4 8-to-32bit S-Boxes. The algorithm is described in RFC-4269 [55].

CAST-128 Developed in 1996 as an application of the CAST structure described in [2], CAST-128 (also known as CAST5) has been approved by the Government of Canada for use by the Communications Security Establishment and was the default choice in GnuPG until superseded by AES in version 2.1 in 2014. In addition to its peculiar structure, it can be configured with a variable key length from 40 to 128bit in 8bit increments, and operates on a 64bit block in either 12 or 16 rounds. The most common variant, CAST5 128bit, is deemed secure for all uses, although the narrow block size opens the way to collision attacks. The cipher is covered by a patent but available worldwide on a royalty-free basis for all applications, a description is also available as RFC-2144 [1].

HIGHT HIGHT is a cipher designed for lightweight applications and for use on low-end devices, first published in 2006 [39]. It has a generalised Feistel network structure with 32 rounds, input and output whitening, a 64bit (narrow) block and a 128bit key. It has no S-Boxes and only uses byte level operations, allowing efficient implementations also on 8bit CPUs.

In addition to the ISO suite, four other ciphers were included, to provide an insight on the behaviour of less common internal structures:

Serpent Designed in 1998 and submitted as an AES candidate [6], it ranked second after the current standard, Rinjndael, despite having a higher security margin. In fact, it was penalised in the selection process solely because of its poor performance, due to the number of rounds being twice as much as what was deemed sufficient, an conscious decision intended as a reassurance against future breakthroughs in cryptanalysis. Serpent has a substitution-permutation network structure, a 128, 192 or 256bit key and operates on a 128bit block in 32 rounds. The the round functions have been designed to allow the parallelisation of all operations, and its 8 4-to-4bit S-Boxes have efficient computational (bitslice) implementations. This cipher has been placed in the public domain.

XTEA XTEA was introduced in 1997 as a lightweight cipher for embedded applications, where it is still used because of its reasonable security margin and extremely simple round function with no S-Boxes, which results both in a small code size and high throughput. It has a Feistel network structure with a 128bit key, 64bit block and a variable number of rounds -- 64 recommended. There is no published specification, although a description and a reference implementation are provided by the authors in [79] .

Speck/Simon Speck and Simon are two families of lightweight block ciphers released by the NSA in 2013 [11] and intended for embedded applications, especially low security ones such as those found in printer cartridges. Both ciphers have similar profiles, with a block size variable from 32 to 128bit, and a key size that varies accordingly from 64 to 256 bits. Speck has an ARX (and, rotate and xor) structure optimised for software implementations, and Simon a Feistel network designed for hardware, with bit-oriented operations in the key schedule. The only configurations considered in the analysis are 128, 192 and 256bit key with a 128bit block.

Noekeon Noekeon is lightweight cipher submitted to the NESSIE program of the European Union in 2000 [27] in two variants, *direct*, which is more efficient but vulnerable to related-key attacks when possible, and *indirect*, marginally slower but not vulnerable. It has a 128bit key and operates on a 128bit block in 16 rounds. Its bitslice structure has been explicitly designed with the protection against side-channel attacks in mind, a fact that makes it unique among the ciphers considered.

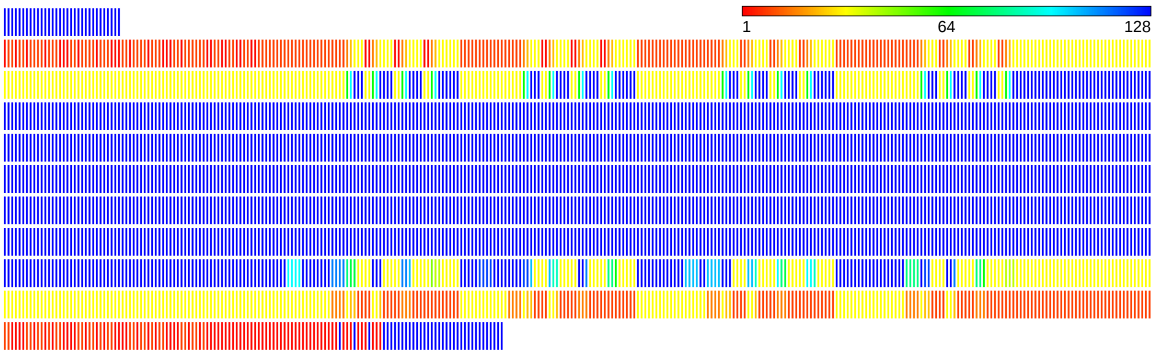
The selection of algorithms presented in this section virtually covers the entire spectrum of real world applications and offers a comprehensive view on how the analysis and masking methods behave in various situations, and thanks to their different internal structures. The features of the ciphers are summarised in Table 3.1, in particular their status and intended usage.

Table 3.2 lists the IR instruction classes used by each implementation, outlining a few interesting points. All algorithms use boolean linear operations $\oplus$ and $\neg$, while non-linear boolean operations $\vee$ and $\wedge$ are mostly seen in alternative to modular arithmetic $\boxplus$ and $\boxminus$, with the exceptions of SEED and CAST-128, which use both. Most ciphers also use table lookups for S-Boxes, excluded XTEA, which has none, and Serpent, which adopts computational (bitslice) ones. To conclude, constant shifts and rotations are widely used as the easiest way to achieve diffusion, with CAST-128 being the only algorithm to adopt shifts by variable amount.

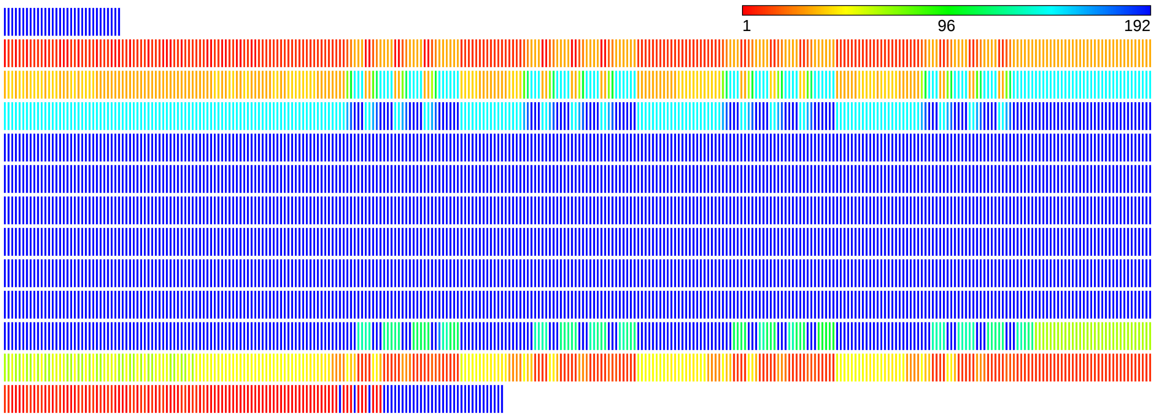
3.2 Visualisations

The analysis tool may also be used to produce a visualisation of dependency matrices and vulnerability indices in form of heatmaps. Before plotting, dependency matrices are $+$-reduced by rows, yielding a vector of integers having in each cell the number of key (or subkey) bits on which the corresponding bit of the variable depends. This vector is plotted as a group of vertically stacked segments, each corresponding to one bit, with the LSB on top and the MSB at the bottom. The vulnerability index is already expressed in form of an integer, so it can be plotted as a single segment with no transformation involved. Each group (for vulnerability matrices) or segment (for vulnerability indices) is associated with an instruction in the cipher, and these blocks are laid out in rows, one for each round identified during the analysis (see Section 2.2.3). Each individual segment is in turn filled with a solid colour from a red to blue palette obtained by HSV interpolation in the hue range $0° - 240°$, as shown in Algorithm 3.2.1, plus black for 0. All the heatmaps included in this section have been directly generated by the analysis tool, with no post-processing involved other than scaling and centering.

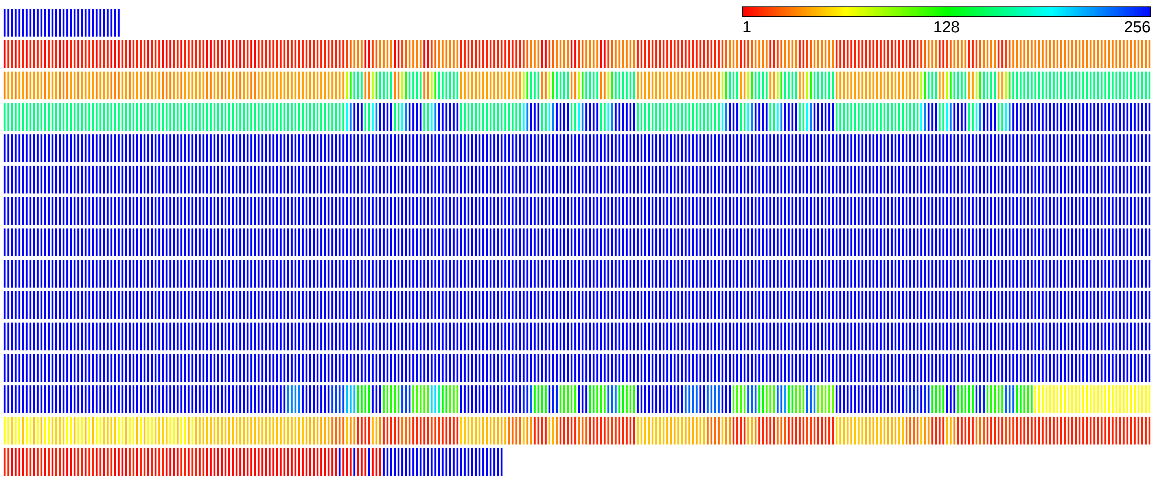
Figure 3.1 shows the vulnerability index of AES-128, -192 and -256. In all three cases the diffusion is completed by the outermost three rounds, containing the impact of a masking at keyize level. The large size of an AES round is well visible, together with the relatively small number of rounds. The key input points



(a) AES-128



(b) AES-192



(c) AES-256

Figure 3.1: Security margin of AES

Algorithm 3.2.1: Interpolation algorithm used to colour heatmaps

```
Input: $i \in IndicesSet \subseteq \mathbb{N}$
       $m = \max(IndicesSet)$
Output: A colour corresponding to $i$, encoded as $H, S, V$
1 if $i = 0$:
2     return $0, 0, 0$
3 else
4     return $\frac{i}{m} \cdot 240°, 1, 0.5$
```

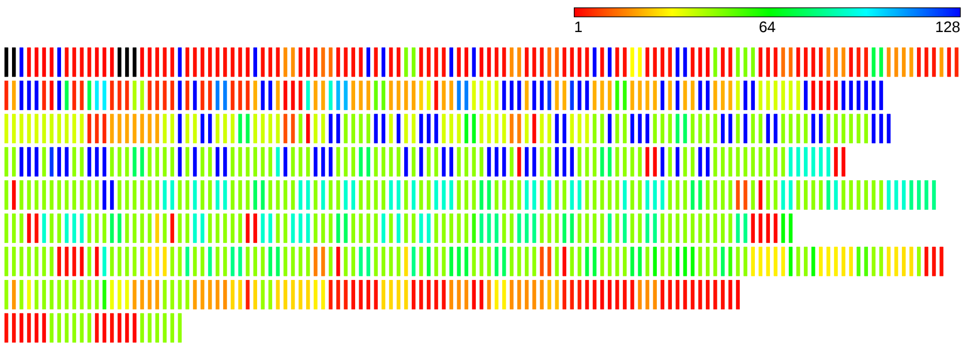


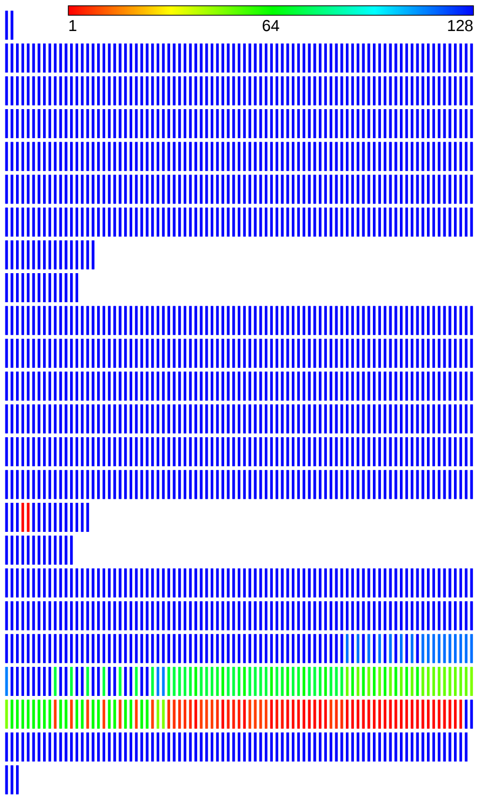
Figure 3.2: Security margin of MISTY1

are also apparent in the second round (the small clusters of red bars), as well as the diffusing effect of the S-Boxes, after which the vulnerability suddenly decreases.

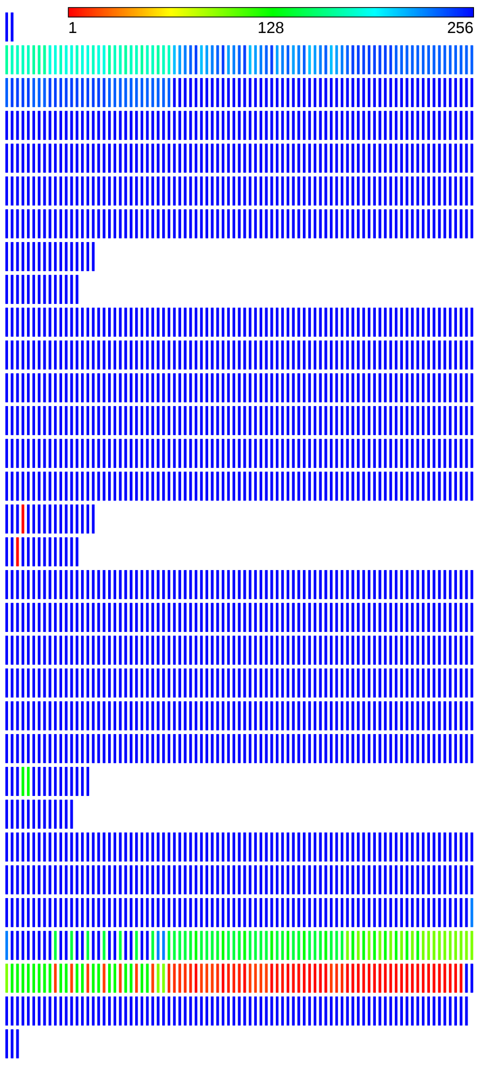
In the case of Camellia-128 and -256, the S-Boxes in the key schedule, paired with the pre- and post-whitening, guarantee a very fast diffusion of the key and leave only a very small attack surface. The heatmap reported in Figure 3.3 confirms this behaviour, showing in particular two instructions in correspondence with the second FL layer (cfr. Section 3.1) with a very low security margin.

Figure 3.5 confirms the good properties of CAST-128, due to the combination of modular arithmetic and shifts by key-dependent values. The diffusion is completed by the second round in both directions, save for one instruction in the fourth from the bottom. The relatively small footprint of this cipher is also well visible, especially when compared with other ciphers for generic use, such as AES.

TDEA2 and TDEA3 both use the same encryption algorithm, only with different key schedules. In particular, the TDEA routine is composed of 3 DES chained, and this fact is clearly visible in the heatmap in Figure 3.7. The central DES is well diffused, on the contrary of the two external ones, which only depend on a part of

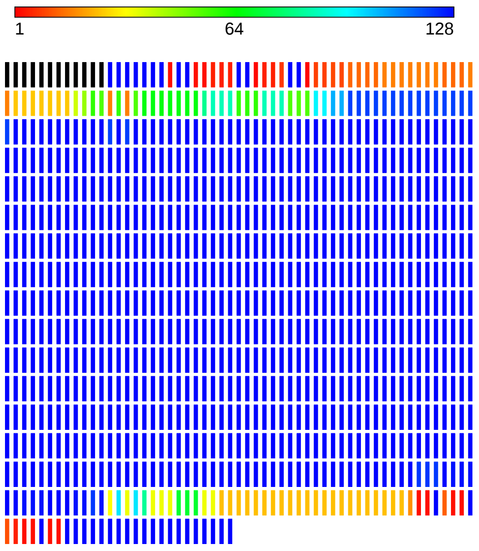


(a) Camellia 128

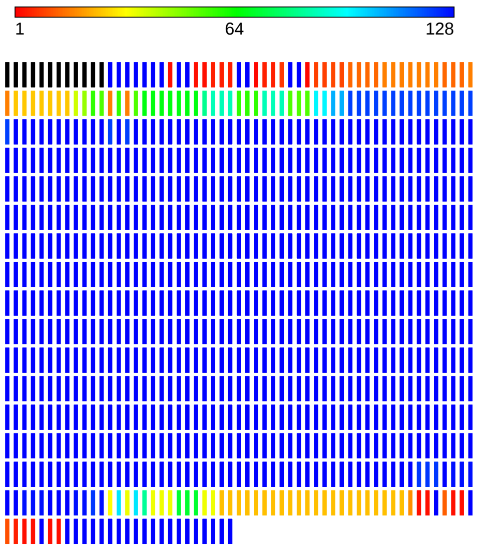


(b) Camellia 256

Figure 3.3: Security margin of Camellia



(a) Direct



(b) Indirect

Figure 3.4: Security margin of Noekeon

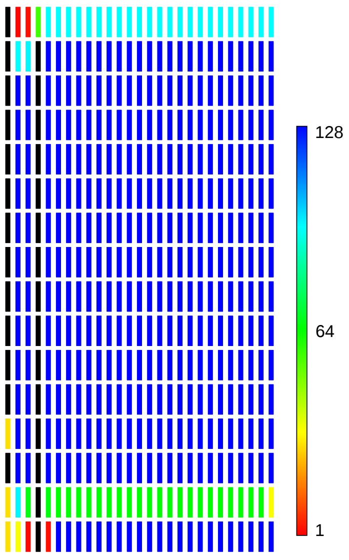


Figure 3.5: Security margin of CAST-128

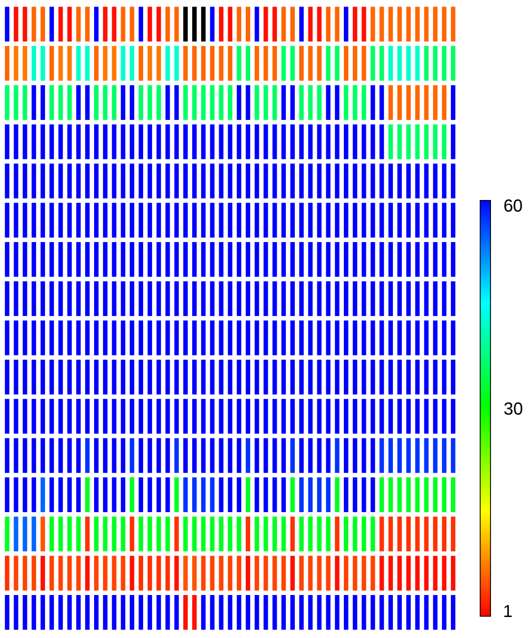
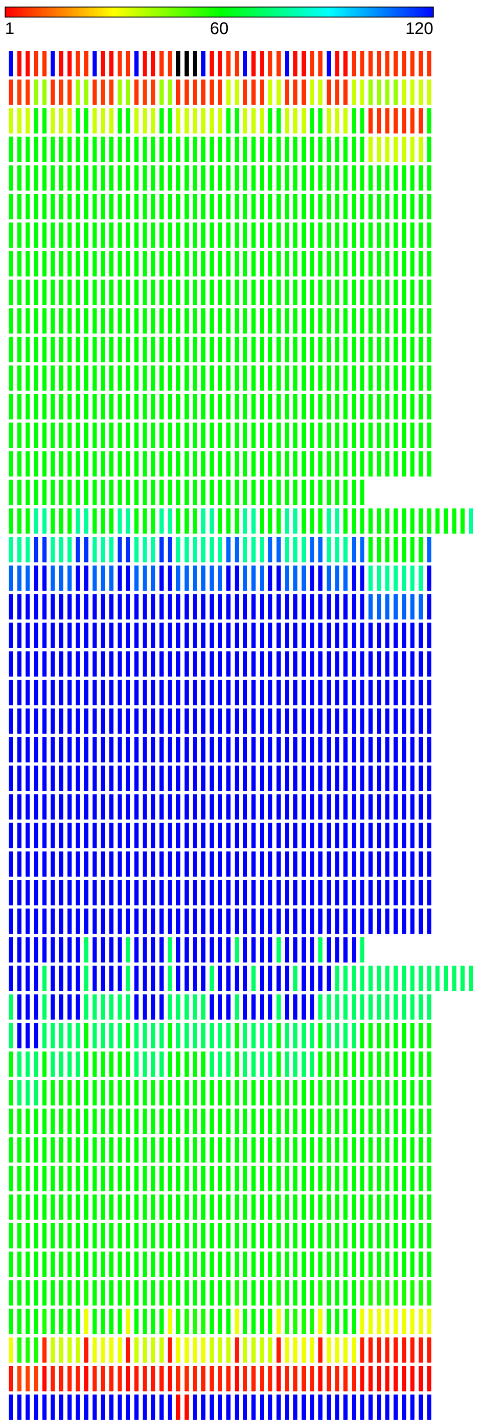
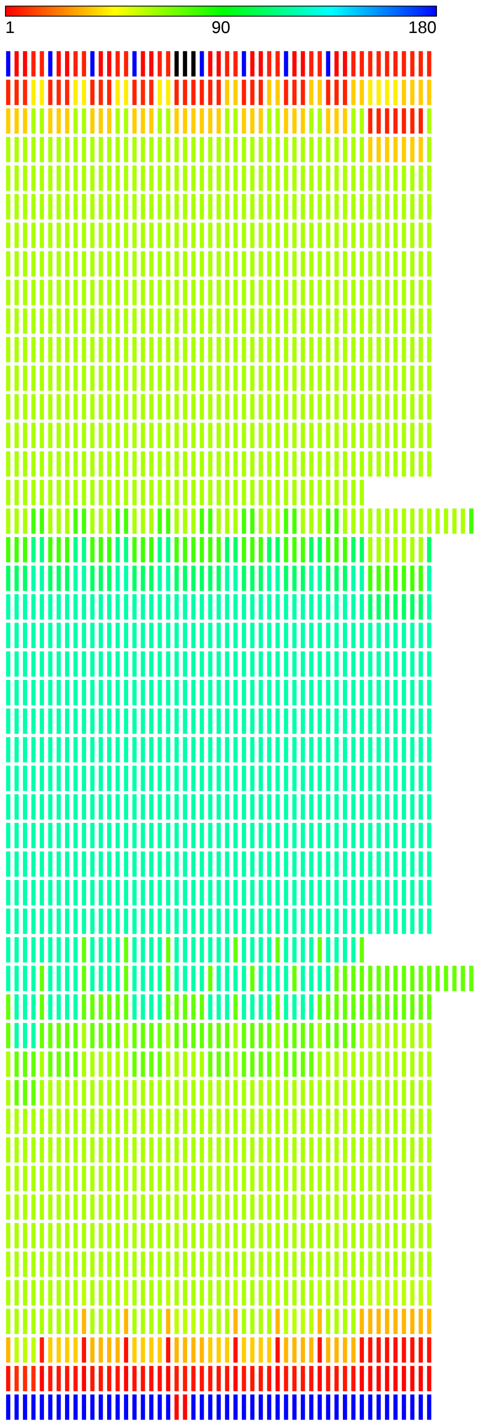


Figure 3.6: Security margin of DES



(a) 2TDEA



(b) 3TDEA

Figure 3.7: Security margin of TDEA

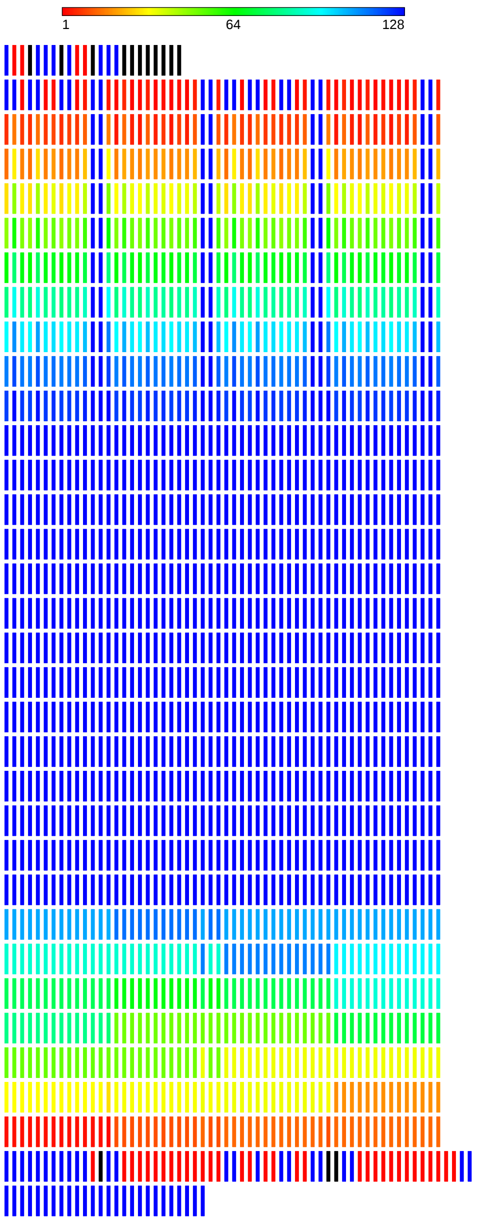


Figure 3.8: Security margin of HIGHT

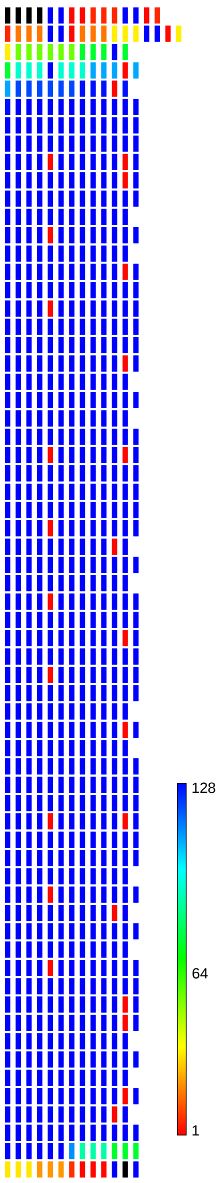


Figure 3.9: Security margin of XTEA

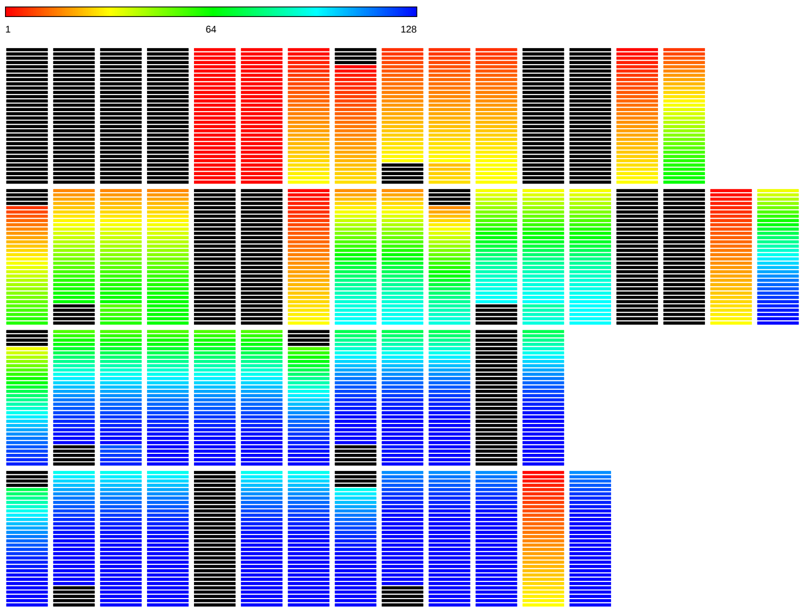


Figure 3.10: Forward key dependencies of the first four rounds of XTEA

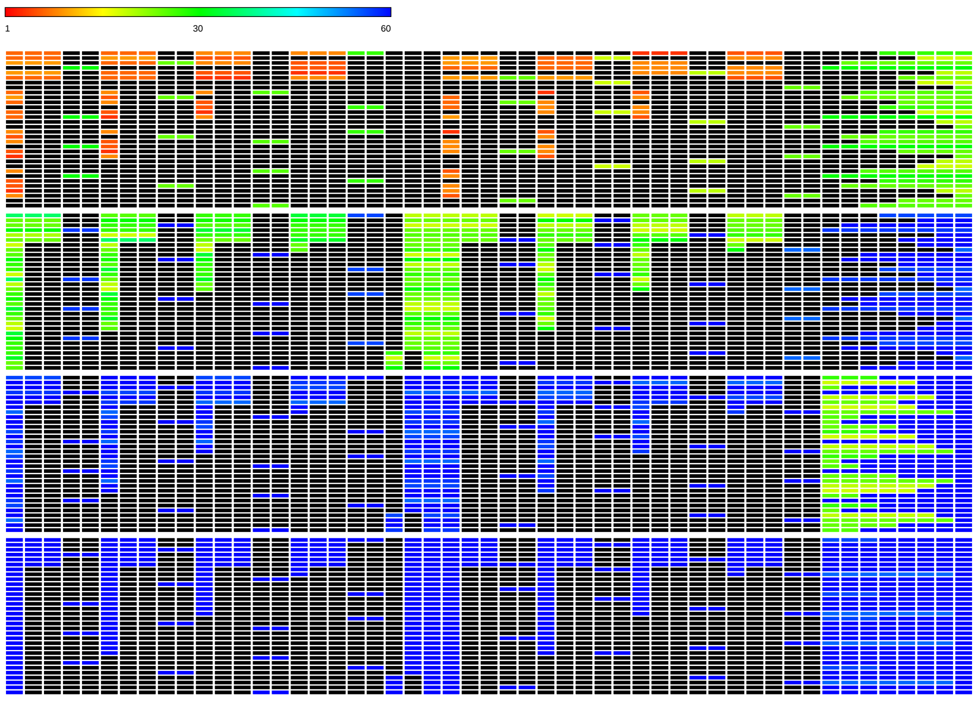
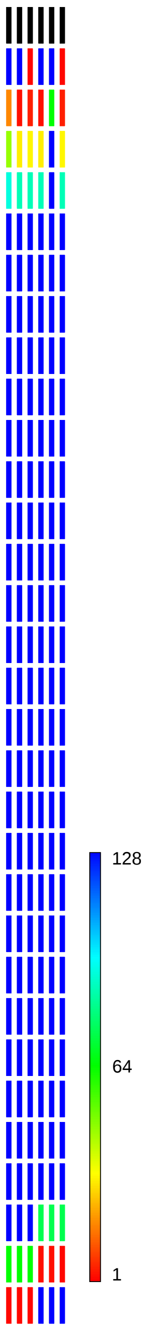
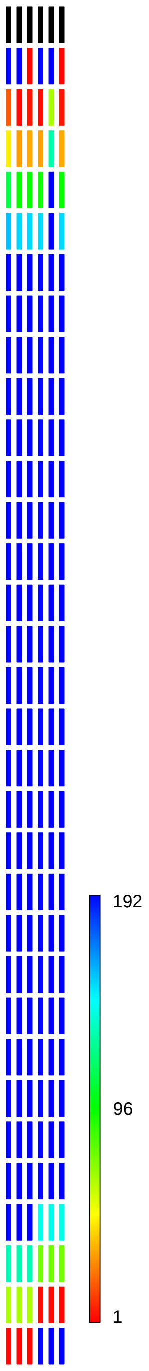


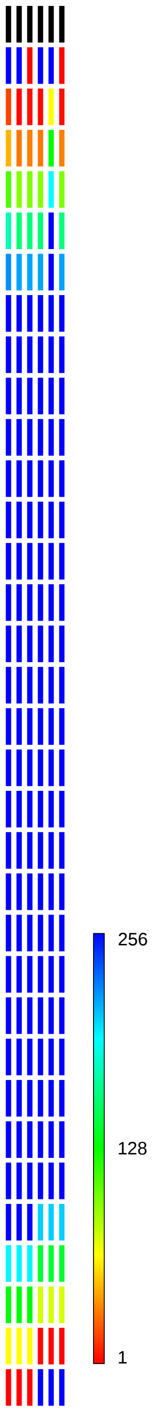
Figure 3.11: Forward key dependencies of the first four rounds of DES



(a) 128/128

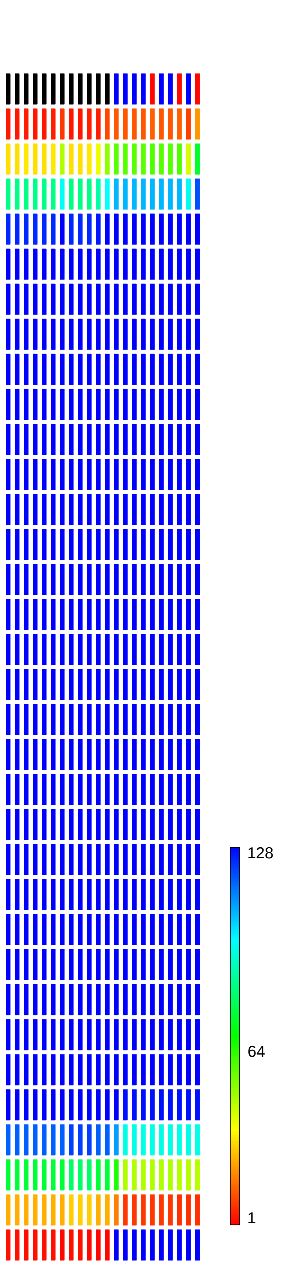


(b) 128/192

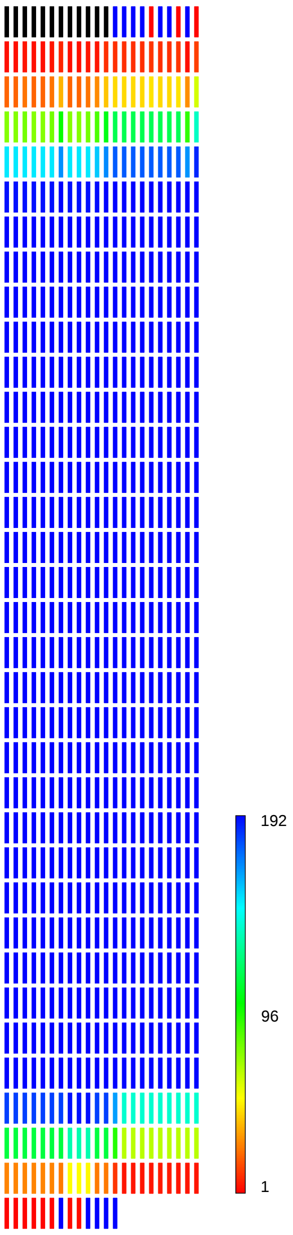


(c) 128/256

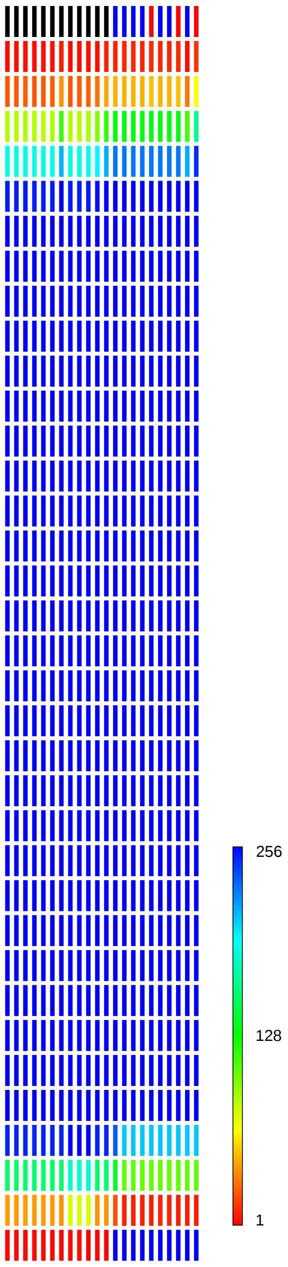
Figure 3.12: Security margin of Speck



(a) 128/128

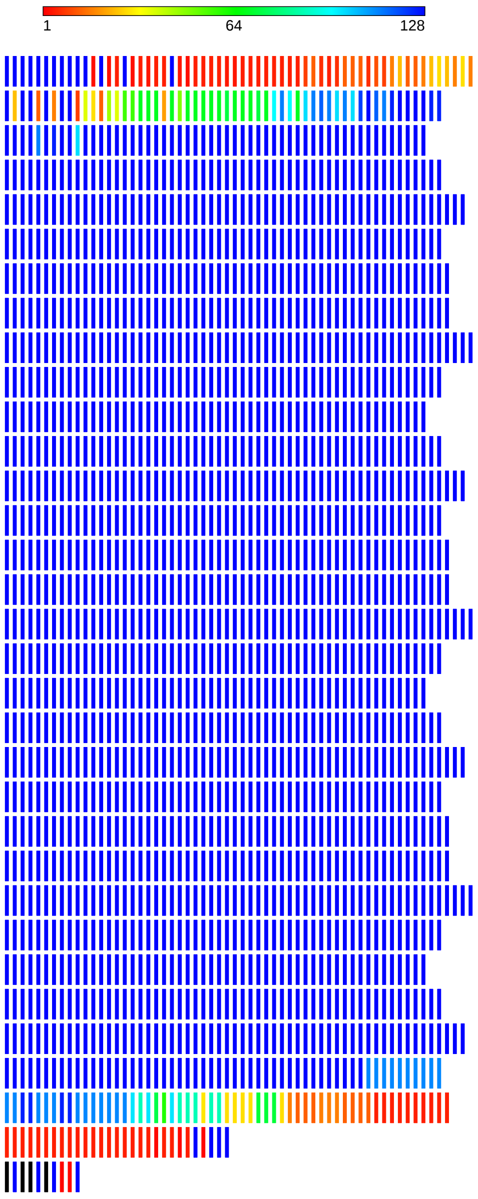


(b) 128/192

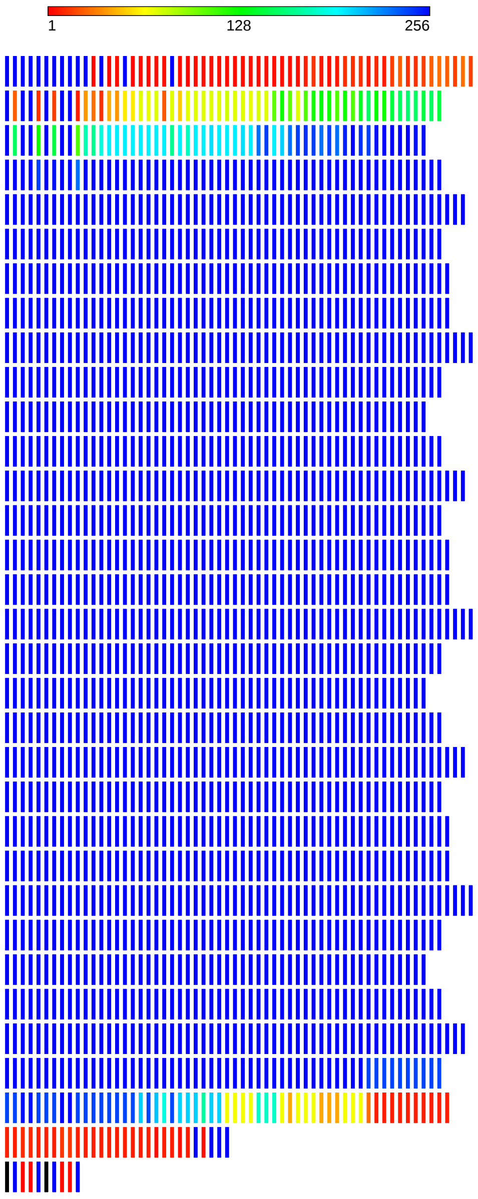


(c) 128/256

Figure 3.13: Security margin of Simon



(a) Serpent 128



(b) Serpent 256

Figure 3.14: Security margin of Serpent

the key. A reordering of the instructions across the separation between the first two DES can also be observed. For comparison, Figure 3.6 reports the security margin of a simple DES, which achieves a reasonable diffusion, notwithstanding its small key size. Figure 3.11 shows a detail of the (reconstructed) key dependencies of the first 4 rounds of DES. The effect of the optimised S-Boxes with pre-calculated shifts can be observed, where only few bits in a 32bit value have dependencies on the key. The bit-oriented nature of DES is well visible too, especially at round 2, with single bits of the same instruction having different vulnerability.

Figure 3.8 points out the slow diffusion of HIGHT, a fact that is partially explained by the small round function, paired with a large number of rounds (32), which still guarantee a complete diffusion by the 8th round in both directions. The blue columns in the first few rounds correspond to the points where well-mixed portions of the key enter the cipher stream. On a different note, Figure 3.2 makes it clear that almost no instruction in MISTY1 reaches a sufficient level of security, due to the very poor diffusion properties of this cipher. In this case, keysize protection might even be detrimental, because of the costs associated with continuously masking and unmasking the operands.

The results for Noekeon are presented in Figure 3.4, which confirms the equivalent security of the two key modes (cfr. Section 3.1) against side-channel attacks and the very fast diffusion, which reaches completion by the second round of 16 in both directions. The cipher was designed with side-channel attack resistance in mind, and the analysis confirms that the authors succeeded in this regard. Good diffusion, together with great performances, place Noekeon among the best choices for lightweight encryption applications.

Serpent is another cipher with very good properties. As shown in Figure 3.14, the diffusion is complete by the second round for -128, and the third for -256 (and -192 too). Serpent admittedly has a higher security level of the AES, but a very different internal structure, with a comparatively smaller round function and computational (bitslice) S-Boxes. These two properties guarantee excellent performances to the protected version, even if the cipher is penalised by the large number of rounds.

Going back to lightweight ciphers, Figure 3.10 presents the reconstructed key dependencies of the first few rounds of XTEA, where the typical “gradient” behaviour of modular additions is visible. Figure 3.9 reports the security margin, in which the key input points are well visible as red bars inside each round. The vulnerability index of Speck, another lightweight cipher based on modular additions, is shown in Figure 3.12 and the very fast diffusion effect of modular sums can again be appreciated, together with the extremely small size of the round function, whose main components are two rotations by a constant and one modular addition. Simon, the hardware-optimized companion to Speck, exhibits similar properties, and completes diffusion by the fourth round, as visualised in Figure 3.13.

3.3 Benchmark

All the algorithms described in Section 3.1 have been implemented in C starting from the respective reference document. The code has been optimised to the fullest extent possible, but without making assumptions on the machine, such as the availability of a certain instruction, or using specially crafted assembly code.

The intent is to obtain a comprehensive analysis of the performance obtainable from “standard” source code compiled for two different but very popular architectures: MIPS32 and ARMv7. Both are common sights in embedded devices, with the latter being prevalent, and they have been chosen precisely for being representative of the class of devices that have cryptographic material on board and operate as authentication or encryption mechanisms, in smart cards, security tokens, or encrypted device controllers. The crucial detail in these applications is that the key is usually burned in ROM or otherwise embedded in the SoC, and therefore not readily available, making them the most common target of side-channel attacks.

The first selected device is a ST Microelectronics STM32F407 board, featuring a ARMv7 Cortex-M4 CPU operating at 120 MHz clock and endowed with 128kiB of RAM, 1MiB of flash and a hardware random number generator, running bare metal. The second board is an Imagination Creator CI20 board equipped with a dual core MIPS32 processor clocked at 1.2GHz, 32k L1 instruction and data cache, 512k L2 I&D cache, and 1GiB of DDR3 RAM, running Debian 7 with Linux 3.0.8. In both cases the code has been compiled with the modified version of Clang+LLVM based on 3.8.0 trunk.

The MIPS machine does not exactly conform to the specifications of an embedded device, however the overhead measurements could still shed some light on the behaviour of a different architecture in the same scenario.

An important distinction to make is that the ARMv7 code runs on bare metal, with virtually zero overhead and small variance almost exclusively due to clock skews. On the other hand, the MIPS board runs a stripped down version of a Linux-based OS, which means that several concurrent processes might cause a context switch or deplete the system-wide random pool, making the measurements more fuzzy.

In case of MIPS, measurements where collected using the kernel timer `CLOCK_PROCESS_CPUTIME_ID`, which only accounts for cycles attributed to the current process, at a reported precision of 1ns. For ARMv7, the board was programmed to raise and lower an output using the high speed GPIO feature and the measurements were taken using an oscilloscope to capture the temporal distance between the rising and falling edges of the collected signal.

A last but very important detail is the generation of randomness. The ARMv7 SoC was endowed with a hardware RNG with an approximate refresh time of 4 clock cycles, meaning that fresh bits were readily available whenever requested by the program. The MIPS board had no such facility, and therefore the psuedo-RNG software implementation offered by the Linux kernel was used, which has a significantly higher overhead and delivers randomness at a larger variance due its to internal maintenance operations. In order to overcome this last issue, the randomness was dispensed through a 256MiB buffer, filled from the system RNG before the start of the benchmark.

The following sections present and discuss the results of the benchmarks on the two platforms in exam, with the goal of finding the algorithms that exhibit the best behaviour, as in performance, masking overhead and code footprint. For ARMv7s, results are given by the average of 5 runs, while on MIPS the collection strategy was “best 8 out of 10”, in order to account for the periodic bursts in execution time caused by the maintenance routines of the system RNG. On both architectures the single iteration of the benchmark consisted in the encryption of a single block of an hard-coded random plaintext, or the test vector where available.

The size of the generated code is reported alongside times and overheads, as it is of great concern for embedded platforms, those that might benefit the most from the protected code. In this regard, STM32F204 is already at the higher end of the spectrum with 1MiB of flash memory, other devices might be even more limited (512kiB or less), and the masked algorithm would not terribly useful if it couldn’t even fit in memory, not to mention that the encryption routine is supposed to be a component of a larger application, which will necessarily increase the code size. The reported figures include the benchmark hooks which signal the begin and end of the routine. For this reason, the reported values might be a few bytes larger than the encryption code alone.

In all graphs, a solid lines represents data for the keysize-level masking (cfr. Section 2.7), and the corresponding dashed line brings the data for full masking. For reference, the complete data set is reported in Appendix A. Ciphers with unique key sizes, namely DES/TDEA and CAST5-80, have been included in the tables, but excluded from the analysis.

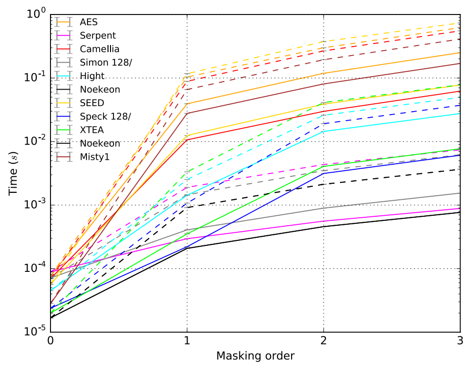
3.3.1 128 bit ciphers

Figure 3.15 reports the benchmark data for 128bit ciphers on ARMv7 and MIPS32. A comparison of the time and overhead plots on ARMv7 shows that AES has the worst performance, closely followed by MISTY1, although the latter has a sensibly worse overhead. In both cases, the initial overhead is almost two order of magnitudes, but then grows at a slow pace with the increase of the masking order. The large overhead of AES is justified by the computationally heavy protection of S-Boxes and the large round function, while MISTY1 pays the price of its weak diffusion properties, which result in most of the instructions requiring masking, even in the inner rounds.

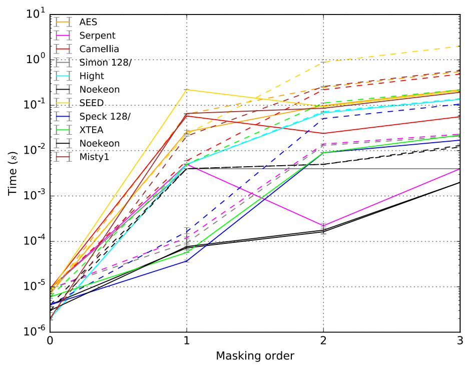
SEED and Camellia also exhibit a similar behaviour, although with better running times -- almost by half an order of magnitude. HIGHT performs better than all the previous, but its curves are characterised by a steeper ascent, explained by the higher cost of masked additions. The sudden change of slope at the second order is explained by the different algorithms used.

Speck and XTEA have a similar profile to SEED, but an order of magnitude faster, as they also use modular additions as part of their round functions. The internal structures of the two are very similar, with a tiny round function, 64 iterations and diffusion achieved with modular sums. This explains the very similar curves, both in time and overhead, although Speck appears to be slightly more efficient. Despite the steep overhead curve, these two start with excellent performance, and therefore maintain good figures even at higher orders.

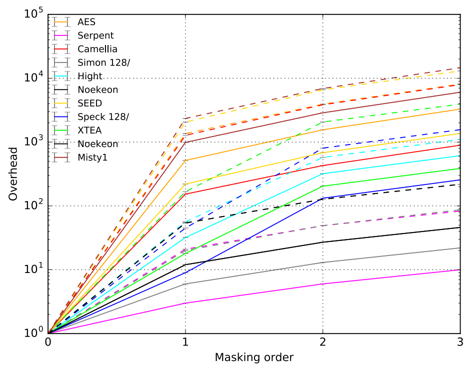
Noekeon, Simon, and Serpent exhibit the best performance (in this order), with a moderate initial masking cost, and a slow increase with the masking order. Serpent in particular appears very interesting in this regard, since it starts with an unmasked time close to AES and Camellia, but evolves at a much lower rate, with an overhead consistently below 10$\times$ and better than those of Noekeon and Simon.



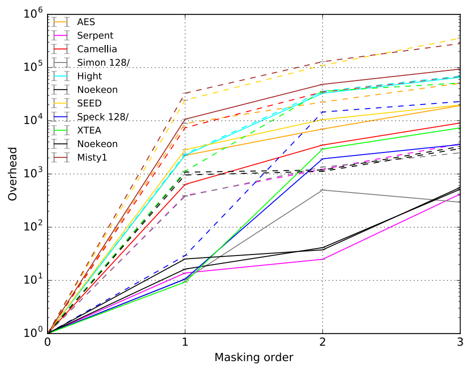
(a) Time on ARMv7



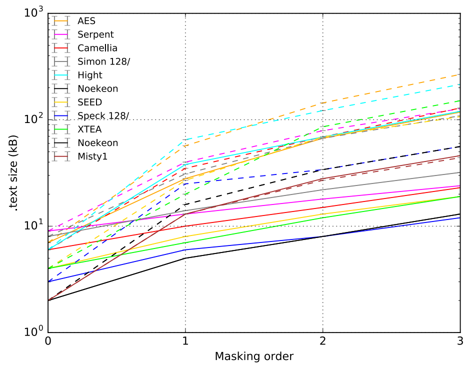
(b) Time on MIPS32



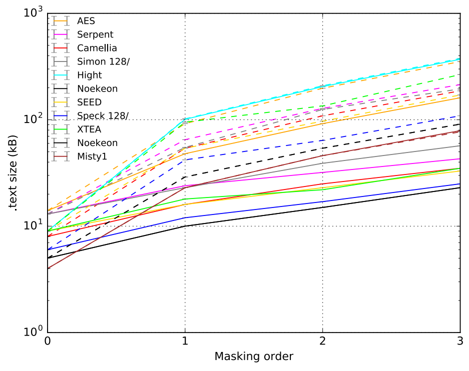
(c) Masking overhead on ARMv7



(d) Masking overhead on MIPS32



(e) `.text` size on ARMv7



(f) `.text` size on MIPS32

Figure 3.15: Benchmark results for 128 bit ciphers

This fact is explained by the simultaneous lack of table lookups and additions, whose protection constitute the bulk of the overhead in the other cases, and it is even more interesting when considering that Serpent has not been engineered as a lightweight cipher, on the contrary of the other two.

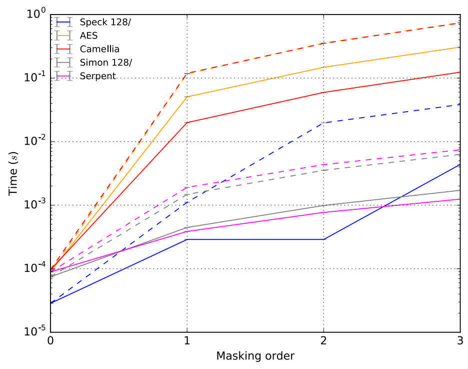
The behaviour on MIPS32 follows closely, although Camellia, SEED and Serpent exhibit a sudden increase of the computation time around the first order, a fact for which no reasonable explanation was found. The only notable difference is in AES, which performs visibly better than SEED, a fact that might be explained by the much slower RNG. The most interesting fact is that the MIPS machine is clocked at almost an order of magnitude higher than the ARMv7, but performs comparably. This is explained by the presence of a hardware random number generator on the latter architecture, which provides an essential boost, particularly at higher orders and in those cipher that include lookups and modular additions.

Both architectures have comparable trends in code size, although MIPS32 code tends to be larger. On ARMv7, AES has the largest footprint, with all the other ciphers exhibiting similar behaviours and Noekeon and Speck having the smallest size. On MIPS32 HIGHT has the largest footprint, followed by AES and all the others. Particularly interesting are the curves of Serpent and Simon, which exhibit the most moderate growth, again due to the lack of heavy table lookups and additions.

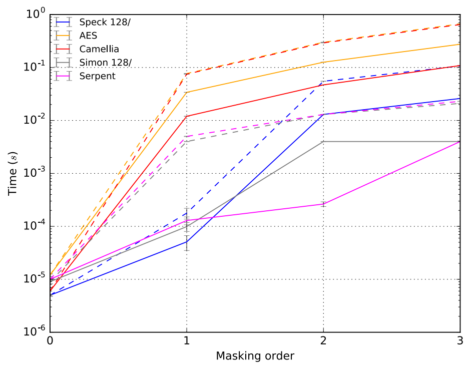
To conclude, Noekeon in both variants (cfr. Section 3.1) has the best performance and the smallest footprint across all plaftforms, while Serpent exhibits the smallest masking overhead. It is worth mentioning that the trend for this latter is less clear on MIPS32 at the first order, but becomes apparent at higher ones.

3.3.2 192 bit and 256 bit ciphers

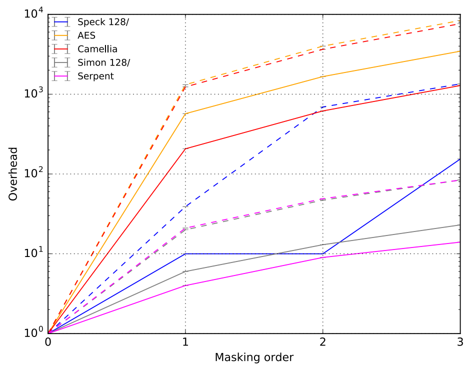
Benchmark data for 192bit and 256bit ciphers are respectively contained in Figure 3.16 and Figure 3.17. In both cases AES exhibits the worst performance and overhead, closely followed by Camellia in the same order of magnitude. Simon and Serpent have a similar behaviour on ARMv7, while the latter appears to have an advantage at higher orders on MIPS32, a fact possibly explained by their different demand of random value, which becomes apparent on a platform with a sensibly slower RNG. Speck has the best performance at low masking order, but is quickly impeded by the overhead of protected modular additions and overrun by Serpent and Simon. The crossing happens around the second order for a 192bit key on



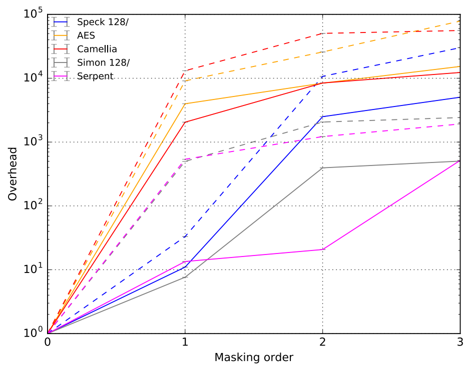
(a) Time on ARMv7



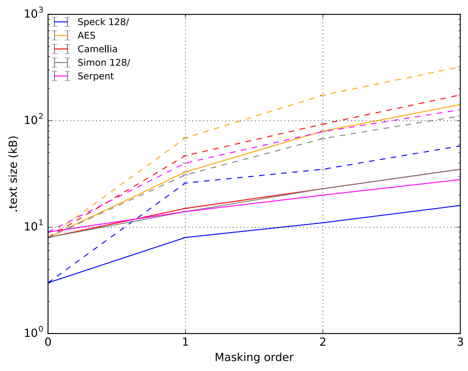
(b) Time on MIPS32



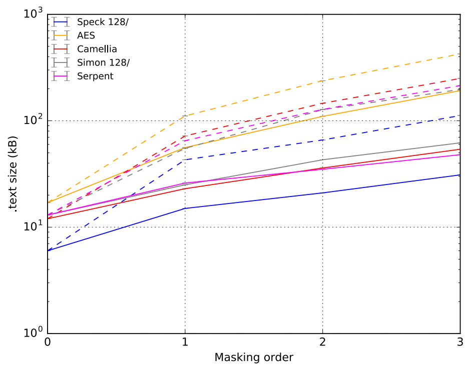
(c) Masking overhead on ARMv7



(d) Masking overhead on MIPS32

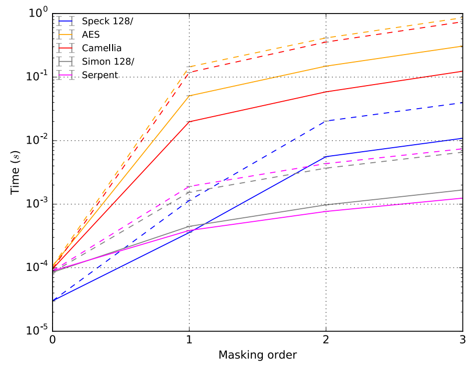


(e) `.text` size on ARMv7

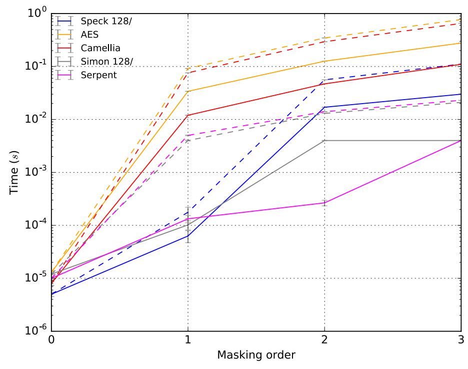


(f) `.text` size on MIPS32

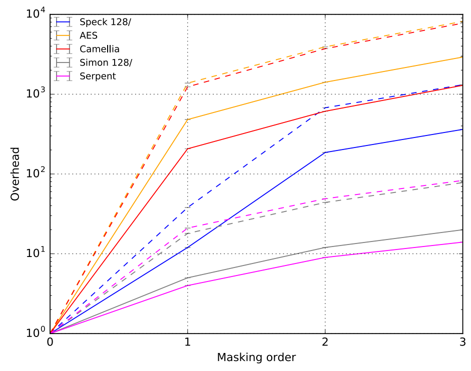
Figure 3.16: Benchmark results for 192 bit ciphers



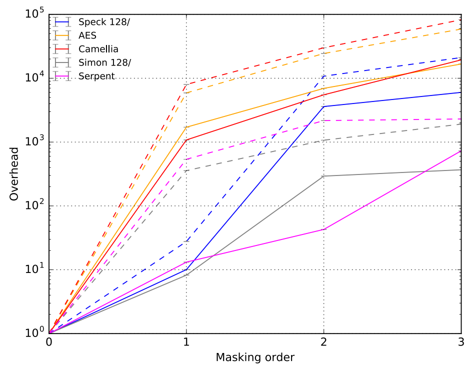
(a) Time on ARMv7



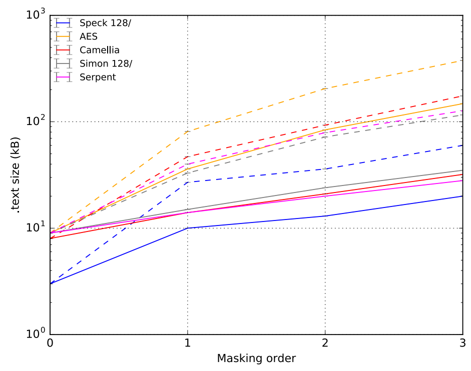
(b) Time on MIPS32



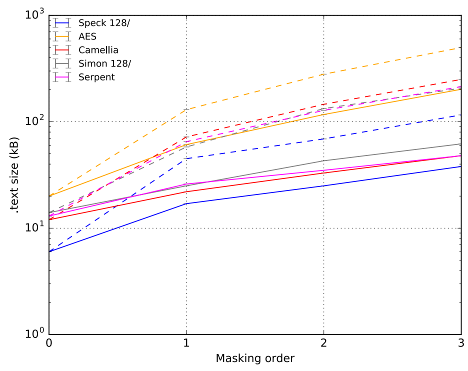
(c) Masking overhead on ARMv7



(d) Masking overhead on MIPS32



(e) `.text` size on ARMv7



(f) `.text` size on MIPS32

Figure 3.17: Benchmark results for 256 bit ciphers

ARMv7, and the first in all the other cases.

All the ciphers have a very similar footprint, with the only exception of Speck, which is consistently the most compact by half an order of magnitude or more, thanks to the small round function combined with good diffusion properties that limit masking to few outermost rounds. Also at these key sizes, Serpent has the overall lowest overhead, on both platforms.

3.3.3 Overall results

From the results, Serpent emerges as a clear winner, thanks to its moderate overhead in code size and time, and a security level significantly higher than AES (Rijndael). AES itself exhibits worse results, with a running time consistently higher by two orders of magnitude. In the range of lightweight ciphers, Noekeon offers the best results, with an overhead which grows less than an order of magnitude across three orders of masking. It is interesting to notice that unmasked Speck has a run time similar to the one of Noekeon, but the masked code suffers from the overhead of the protected 64 bit modular addition at the core of the round function. In general, the most penalised algorithms are those employing table-based S-Boxes as part of their round function, in which case masking overheads of 1000$\times$ are common, due to the inherent difficulties in masking lookups, as described in Section 6.

In absence of a masking scheme for shifts by a variable quantity, I was not able to protect CAST-128 – the only algorithm which makes use of this operation. Consequently, this cipher could not be included in the benchmarks, although figures for the unmasked version are reported in the tables in Appendix A for completeness.

Conclusion

In thesis I have described an efficient method to evaluate the resistance of software implementations of symmetric ciphers against different classes of side-channel attacks and selectively apply countermeasures in a a fully automated way.

The visualisations obtained from the analysis tool I built have given interesting insights into the security properties on many symmetric ciphers with a level of detail as fine-grained as the single instruction. In some cases I could confirm what I already knew, for instance that Misty1 is irreparably broken. In others instances I obtained results that prompted further analysis and revealed interesting properties. For instance, I obtained very promising results for Serpent and Noekeon characterised by very good diffusion properties, as revealed by the analysis, and the exclusive use of primitive operations for which efficient masking schemes exist, making them very good candidates to provide fast and secure symmetric encryption in embedded applications.

The field of automated cipher analysis and side-channel countermeasures application is very young, even though some significant research has been published [3, 10], and I hope that my work will manage to provide a non-null addition – if anything for the detailed analysis of a vast portfolio of cryptographic primitives I produced, which I believe have no precedent in literature.

A lot of issues remain open, and I took great care in engineering the analysis tool in a way that is flexible enough to enable hassle-free expansions to take into account new attack models. Just to name a few possibilities, I did not investigate protection against hamming-distance leakage [7], which is prevented at backend level by making sure that no two shares of the same value are written to the same register.

More in general, an intermediate representation does not necessarily reflect the exact code produced by the backend. In my work I have assumed that the two representation are close enough to allow conclusions drawn in one to be propagated to other. This is generally true, however no formal guarantee is given in practice. Architecture specific modifications to the compiler could be studied to prevent instruction rescheduling and optimising transformations which might undermine the security guarantees of the applied transformation.

While the set of optimisations I could enable is very similar to the full optimising selection that `clang` adopts in `-O3` mode, a couple of passes were left behind. These were problematic in that they introduced new (vectorial) operators and intrinsics. It is not impossible to expand the analysis to handle these cases as well, and it would interesting to see if the previously excluded optimisations yield any performance improvement over the current results.

In addition to that, the countermeasures rely on the ISW assumption that encoding and decoding routines are invulnerable to attacks targeting the implementation (see 1.4.1), which is not necessarily the case. No hardware architecture am I aware of offers a similar protected encoding/decoding facility, nevertheless it will be interesting to revaluate the results when one will be available.

Finally, I could not find any efficient solution in literature for masking a shift by variable amount, and therefore I was not able to protect the (only) user of it, CAST-128. When a transformation will be published for this operation, it will be nice to see how this cipher compares to the others, especially because it adopts a unique machine operation.

The masking schemes for table lookups and modular additions is likely to have room for improvement. Suffice to say that until the summer of 2014, no algorithm existed in literature for higher order masking of additions, and that very different approaches to lookup-tables have been explored. It would be interesting to update the analysis, run the same tests in a few years and compare the results to see the advancements in a field which is moving fast.

Appendix A

Benchmark data

Table A.1: Benchmark for 128bit key ciphers on ARMv7

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$76.3\mu \pm 3.2p$	$103.7m \pm 0$	$299.6m \pm 19.7\mu$	$621.2m \pm 0$
		$39.3m \pm 1.86\mu$	$118.7m \pm 3.85n$	$250.5m \pm 14.98n$
Camellia	$69.6\mu s \pm 3.2p$	$88.78m \pm 3.745n$	$267.2m \pm 11.02n$	$552.7m \pm 0$
		$10.66m \pm 931p$	$29.88m \pm 2.459\mu$	$62.16m \pm 2.29n$
Serpent	$89\mu s \pm 4.41n$	$1.903m \pm 67p$	$4.356m \pm 268p$	$7.42m \pm 547n$
		$296\mu s \pm 11.88p$	$560\mu \pm 82.3p$	$886\mu \pm 22p$
Speck 128/	$23.78\mu \pm 0$	$1.076m \pm 58p$	$19.13ms \pm 1.278\mu s$	$37.2ms \pm 1.429ns$
		$221.5\mu \pm 7.78p$	$3.148ms \pm 134ps$	$6.099ms \pm 197ps$
Simon 128/	$71.3\mu \pm 3.6p$	$1.455m \pm 82p$	$3.486m \pm 232p$	$6.204m \pm 232p$
		$407.5\mu \pm 235n$	$896.3\mu \pm 0$	$1.545m \pm 82.3p$
Noekeon D.	$16.78\mu s \pm 1p$	$914\mu \pm 41p$	$2.128m \pm 235n$	$3.666m \pm 134p$
		$208.5\mu \pm 0$	$458.5\mu \pm 235n$	$767.8\mu \pm 0$
Noekeon I.	$16.78\mu s \pm 1p$	$914\mu \pm 41.16p$	$2.128m \pm 235n$	$3.666m \pm 288n$
		$208.5\mu \pm 0$	$458.3\mu \pm 0$	$767.8\mu \pm 47.52p$
Hight	$45.08\mu \pm 0$	$2.534m \pm 124.5p$	$25.96m \pm 0$	$49.63m \pm 1.429n$
		$1.431m \pm 47.5p$	$14.44m \pm 445p$	$27.6m \pm 912p$
XTEA	$19.96\mu \pm 0$	$3.332m \pm 116p$	$41.16m \pm 1.461n$	$78.35m \pm 3.49n$
		$353\mu s \pm 0$	$4.073m \pm 268p$	$7.703m \pm 208p$
Misty1	$28.12\mu s \pm 1p$	$65.9m \pm 0$	$194.1m \pm 6.39n$	$411m \pm 17.32n$
		$27.79m \pm 1.651n$	$80.79m \pm 0$	$169.6m \pm 7.561n$
SEED	$56.84\mu \pm 1.2p$	$117m \pm 4.472n$	$374.4m \pm 7n$	$737.4m \pm 30.24n$
		$12.48m \pm 2.281n$	$38.78m \pm 1.89n$	$77.59m \pm 3.643n$
CAST5	$65.94\mu \pm 3.8p$	–	–	–

Table A.2: Benchmark for 192bit key ciphers on ARMv7

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$88.88\mu \pm 2.4p$	$117m \pm 3.419n$	$355.5m \pm 15.59n$	$745.5m \pm 0$
		$50.66m \pm 6.566\mu$	$147.7m \pm 3.6n$	$307.3m \pm 12.23n$
Camellia	$96.38\mu \pm 2.4p$	$119m \pm 8.207n$	$351.6m \pm 9.669n$	$737.1m \pm 0$
		$19.94m \pm 1.967\mu$	$59.48m \pm 3.449n$	$123.8m \pm 3.058n$
Serpent	$89\mu \pm 0$	$1.907m \pm 58p$	$4.35m \pm 0$	$7.44m \pm 208p$
		$386\mu \pm 0$	$769\mu \pm 31p$	$1.245m \pm 47p$
Speck 128/	$28.52\mu s \pm 1p$	$1.11m \pm 58p$	$19.74m \pm 771p$	$38.38m \pm 890p$
		$288.6\mu \pm 11p$	$288.6\mu \pm 11p$	$4.379m \pm 0$
Simon 128/	$74.64\mu \pm 3.9p$	$1.478m \pm 82p$	$3.54m \pm 134p$	$6.302m \pm 0$
		$446\mu \pm 164p$	$989.7\mu \pm 164p$	$1.713m \pm 161n$

Table A.3: Benchmark for 256bit key ciphers on ARMv7

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$105.4\mu \pm 2.4p$	$145m \pm 4.8n$	$413.5m \pm 0$	$860.2m \pm 0$
		$50.73m \pm 1.925\mu$	$148.8m \pm 5.1n$	$307.6m \pm 7.49n$
Camellia	$95.84\mu \pm 3.8p$	$118.6m \pm 3.745n$	$356.3m \pm 14.34n$	$742.7m \pm 0$
		$19.82m \pm 1.22\mu$	$58.59m \pm 2.42\mu$	$123.7m \pm 4.3\mu$
Serpent	$89.2\mu \pm 0$	$1.905m \pm 52p$	$4.354m \pm 0$	$7.439m \pm 255p$
		$386\mu \pm 15p$	$769\mu \pm 29p$	$1.245m \pm 58p$
Speck 128/	$30.12\mu \pm 6.67n$	$1.146m \pm 0$	$20.36m \pm 658p$	$39.58m \pm 890p$
		$358\mu \pm 15.56p$	$5.599m \pm 0$	$10.89m \pm 510p$
Simon 128/	$84.68\mu \pm 3.43p$	$1.542m \pm 235n$	$3.695m \pm 232p$	$6.578m \pm 0$
		$448\mu \pm 41.16p$	$976\mu \pm 82.8p$	$1.675m \pm 235n$

Table A.4: Masking overhead factor on ARMv7

	Fully masked / Keysize masked (1 = unmasked)								
Keysize	128			192			256		
Order	1	2	3	1	2	3	1	2	3
AES	1359	3927	8142	1316	4000	8388	1376	3923	8161
	515	1556	3283	570	1662	3457	481	1412	2918
Camellia	1276	3839	7941	1235	3648	7648	1237	3718	7749
	153	429	893	207	617	1284	207	611	1291
Serpent	21	49	83	21	49	84	21	49	83
	3	6	10	4	9	14	4	9	14
Speck 128/	45	804	1564	39	692	1346	38	676	1314
	9	132	256	10	10	154	12	186	362
Simon 128/	20	49	87	20	47	84	18	44	78
	6	13	22	6	13	23	5	12	20
Noekeon D.	54	127	218	–	–	–	–	–	–
	12	27	46						
Noekeon I.	54	127	218	–	–	–	–	–	–
	12	27	46						
Hight	56	576	1101	–	–	–	–	–	–
	32	320	612						
XTEA	167	2062	3925	–	–	–	–	–	–
	18	204	386						
Misty1	2344	6903	14.6k	–	–	–	–	–	–
	988	2873	6031						
SEED	2058	6587	13.0k	–	–	–	–	–	–
	220	682	1365						

Table A.5: Size of the `.text` section on ARMv7

	Unmasked / Keysize masked / Fully masked (KiB)											
Keysize	128				192				256			
Order	0	1	2	3	0	1	2	3	0	1	2	3
AES	7	28	67	118	8	33	80	142	9	36	84	148
		57	144	267		69	174	321		81	205	377
Camellia	6	10	15	23	8	15	23	35	8	14	21	32
		35	69	129		47	93	175		47	93	175
Serpent	9	13	18	24	9	14	20	28	9	14	20	28
		40	79	127		40	79	127		40	79	127
Speck 128/	3	6	8	12	3	8	11	16	3	10	13	20
		25	34	56		26	35	58		27	36	60
Simon 128/	8	14	22	32	8	14	23	35	9	15	24	35
		31	67	109		31	68	111		33	72	116
Noekeon D.	2	5	8	13	–	–	–	–	–	–	–	–
		16	34	56								
Noekeon I.	2	5	8	13	–	–	–	–	–	–	–	–
		16	34	56								
Hight	6	38	69	120	–	–	–	–	–	–	–	–
		65	122	215								
XTEA	4	7	12	19	–	–	–	–	–	–	–	–
		20	86	151								
Misty1	2	13	28	46	–	–	–	–	–	–	–	–
		13	27	44								
SEED	4	8	13	19	–	–	–	–	–	–	–	–
		27	69	107								
CAST5	4	–	–	–	–	–	–	–	–	–	–	–

Table A.6: Benchmark for key ciphers with other key sizes on ARMv7

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
DES	$107.6\mu \pm 3.4p$	$21.93m \pm 833n$	$63.07m \pm 4.894\mu$	$125.9m \pm 9.785\mu$
		$7.94m \pm 729n$	$19.21m \pm 2.652\mu$	$39.12m \pm 4.58n$
2TDEA	$234.7\mu \pm 6.35p$	$63.79m \pm 5.026\mu$	$185.7m \pm 3.058n$	$377.2m \pm 12.23n$
		$45.69m \pm 1.57\mu$	$129.7m \pm 6.122n$	$262.7m \pm 0$
3TDEA	$318.1\mu \pm 0$	$65.71m \pm 4.44\mu$	$187.4m \pm 10.44n$	$374.7m \pm 11.44n$
		$65.72m \pm 3.778n$	$184.5m \pm 7.49n$	$367.3m \pm 0$
CAST5-80	$51.28\mu \pm 0$	–	–	–

Table A.7: Size of the `.text` section for other ciphers on ARMv7

Order	DES		2TDEA		3TDEA		CAST5-80
0	11		22		31		3
1	22	15	57	47	65	64	–
2	37	21	102	37	107	104	–
3	58	29	178	131	168	163	–

Table A.8: Benchmark for 128bit key ciphers on MIPS32

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$7\mu \pm 855n$	$64m \pm 165\mu$ $26m \pm 2m$	$248m \pm 55\mu$ $98m \pm 157\mu$	$551m \pm 991\mu$ $219m \pm 331\mu$
Camellia	$9\mu \pm 628n$	$6m \pm 36\mu$ $58m \pm 154\mu$	$221m \pm 48\mu$ $24m \pm 22\mu$	$485m \pm 271\mu$ $56m \pm 2m$
Serpent	$9\mu \pm 650n$	$120\mu \pm 28\mu$ $5m \pm 54\mu$	$14m \pm 9\mu$ $222\mu \pm 29\mu$	$23m \pm 15\mu$ $4m \pm 30\mu$
Speck 128/	$4\mu \pm 373n$	$164\mu \pm 38\mu$ $37\mu \pm 2\mu$	$51m \pm 118\mu$ $9m \pm 8\mu$	$106m \pm 1m$ $17m \pm 16\mu$
Simon 128/	$9\mu \pm 497n$	$93\mu \pm 21\mu$ $4m \pm 29\mu$	$13m \pm 22\mu$ $4m \pm 37\mu$	$21m \pm 32\mu$ $4m \pm 40\mu$
Noekeon D.	$3\mu \pm 299n$	$4m \pm 29\mu$ $77\mu \pm 9\mu$	$5m \pm 66\mu$ $177\mu \pm 33\mu$	$12m \pm 2m$ $2m \pm 2m$
Noekeon I.	$4\mu \pm 299n$	$4m \pm 34\mu$ $72\mu \pm 7\mu$	$5m \pm 75\mu$ $163\mu \pm 15\mu$	$13m \pm 1m$ $2m \pm 2m$
Hight	$2\mu \pm 157n$	$5m \pm 91\mu$ $5m \pm 99\mu$	$72m \pm 22\mu$ $68ms \pm 218\mu s$	$139m \pm 30\mu$ $135m \pm 13\mu$
XTEA	$6\mu \pm 458n$	$5m \pm 27\mu$ $57\mu \pm 314n$	$112ms \pm 20\mu s$ $9ms \pm 34\mu s$	$219m \pm 32\mu$ $22m \pm 38\mu$
Misty1	$2\mu \pm 124n$	$21m \pm 53\mu$ $66m \pm 60\mu$	$259ms \pm 84\mu s$ $86ms \pm 47\mu s$	$583m \pm 1m$ $193m \pm 101\mu$
SEED	$8\mu \pm 737n$	$23m \pm 8\mu$ $221m \pm 62\mu$	$889m \pm 64\mu$ $94m \pm 42\mu$	$2 \pm 1m$ $207m \pm 75\mu$
CAST5	$8\mu \pm 1\mu$	–	–	–

Table A.9: Benchmark for 192bit key ciphers on MIPS32

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$12\mu \pm 678n$	$77m \pm 189\mu$	$300m \pm 1m$	$668m \pm 82\mu$
		$34m \pm 24\mu$	$126m \pm 53\mu$	$275m \pm 37\mu$
Camellia	$6\mu \pm 891n$	$75m \pm 20\mu$	$292m \pm 269\mu$	$647m \pm 565\mu$
		$12m \pm 39\mu$	$47m \pm 86\mu$	$109m \pm 13\mu$
Serpent	$10\mu \pm 416n$	$5m \pm 43\mu$	$13m \pm 9\mu$	$23m \pm 29\mu$
		$129\mu \pm 22\mu$	$264\mu \pm 27\mu$	$4m \pm 25\mu$
Speck 128/	$5\mu \pm 229n$	$178\mu \pm 41\mu$	$55m \pm 35\mu$	$107m \pm 168\mu$
		$51\mu \pm 16\mu$	$13m \pm 19\mu$	$26m \pm 38\mu$
Simon 128/	$9\mu \pm 745n$	$4m \pm 26\mu$	$13m \pm 29\mu$	$21m \pm 9\mu$
		$99\mu \pm 19\mu$	$4m \pm 15\mu$	$4m \pm 97\mu$

Table A.10: Benchmark for 256bit key ciphers on MIPS32

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
AES	$13\mu \pm 650n$	$92m \pm 53\mu$	$346m \pm 481\mu$	$770m \pm 153\mu$
		$34m \pm 35\mu$	$126m \pm 21\mu$	$277m \pm 49\mu$
Camellia	$8\mu \pm 685n$	$76m \pm 21\mu$	$294m \pm 310\mu$	$651m \pm 594\mu$
		$12m \pm 34\mu$	$47m \pm 126\mu$	$110m \pm 133\mu$
Serpent	$10\mu \pm 628n$	$5m \pm 68\mu$	$14m \pm 26\mu$	$23m \pm 44\mu$
		$133\mu \pm 21\mu$	$266\mu \pm 32\mu$	$4m \pm 29\mu$
Speck 128/	$5\mu \pm 488n$	$177\mu \pm 43\mu$	$56m \pm 114\mu$	$111m \pm 199\mu$
		$63\mu \pm 16\mu$	$17m \pm 25\mu$	$30m \pm 64\mu$
Simon 128/	$12\mu \pm 272n$	$4m \pm 76\mu$	$13m \pm 18\mu$	$21m \pm 33\mu$
		$102\mu \pm 20\mu$	$4m \pm 32\mu$	$4m \pm 44\mu$

Table A.11: Masking overhead factor on MIPS32

	Fully masked / Keysize masked (1 = unmasked)								
Keysize	128			192			256		
Order	1	2	3	1	2	3	1	2	3
AES	8.91k	22.5k	50.8k	9.1k	25.8k	78.5k	5.89k	24.4k	58.9k
	2.24k	7.01k	19.3k	3.98k	8.37k	15.3k	1.71k	6.94k	16.7k
Camellia	7.41k	34.8k	67.6k	13.1k	50.4k	55.9k	7.99k	30.0k	82.5k
	637	3.50k	9.09k	2.05k	8.42k	12.3k	1.08k	5.50k	19.4k
Serpent	393.1	1.23k	3.70k	538	1.22k	1.92k	537.23	2.17k	2.30k
	13.78	24.94	420	13.46	20.74	517	13.17	42.82	730.17
Speck 128/	29.24	14.6k	23k	33.1	10.7k	30.1k	27.68	10.78k	21.07k
	10.62	1.93k	3.58k	11.0	2.50k	5.04k	10.06	3.60k	6.0k
Simon 128/	381	1.35k	2.50k	499	2.06k	2.42k	356.8	1.07k	1.91k
	10.46	499	295.2	7.64	394.6	503	8.23	293	368.1
Noekeon D.	959	1.12k	3.00k	–	–	–	–	–	–
	25.27	37.50	558.9						
Noekeon I.	1.08k	1.19k	3.28k	–	–	–	–	–	–
	16.33	41.28	516.3						
Hight	2.39k	35.84k	69.47k	–	–	–	–	–	–
	2.26k	32.89k	65.73k						
XTEA	1.13k	35.97k	51.26k	–	–	–	–	–	–
	9.31	2.95k	7.35k						
Misty1	32.97k	129k	283k	–	–	–	–	–	–
	10.71k	48.45k	93.7k						
SEED	24.39k	110k	357k	–	–	–	–	–	–
	2.86k	10.52k	19.78k						

Table A.12: Size of the `.text` section on MIPS32

	Unmasked / Keysize masked / Fully masked (KiB)											
Keysize	128				192				256			
Order	0	1	2	3	0	1	2	3	0	1	2	3
AES	14	48	92	161	17	56	110	192	20	61	117	202
		92	198	352		111	238	425		130	280	498
Camellia	8	16	25	35	12	23	36	54	12	22	33	48
		54	109	187		72	146	250		72	146	250
Serpent	13	24	32	43	13	26	35	48	13	26	35	48
		65	128	214		65	128	214		65	128	214
Speck 128/	6	12	17	25	6	15	21	31	6	17	25	38
		42	64	109		43	66	112		45	69	116
Simon 128/	13	23	39	57	13	25	43	62	14	25	43	62
		55	125	195		55	127	197		58	133	207
Noekeon D.	5	10	15	23	–	–	–	–	–	–	–	–
		29	54	91								
Noekeon I.	5	10	15	23	–	–	–	–	–	–	–	–
		29	54	91								
Hight	9	101	205	370	–	–	–	–	–	–	–	–
		102	209	377								
XTEA	9	18	22	35	–	–	–	–	–	–	–	–
		95	135	266								
Misty1	4	23	46	79	–	–	–	–	–	–	–	–
		23	46	77								
SEED	9	16	23	33	–	–	–	–	–	–	–	–
		54	97	171								
CAST5	7	–	–	–	–	–	–	–	–	–	–	–

Table A.13: Benchmark for key ciphers with other key sizes on MIPS32

Cipher	Unmasked	Fully masked / Keysize masked		
		Order 1	Order 2	Order 3
DES	$12\mu \pm 590n$	$36m \pm 78\mu$	$148m \pm 106\mu$	$331m \pm 47\mu$
		$13m \pm 21\mu$	$45m \pm 51\mu$	$103m \pm 76\mu$
2TDEA	$24\mu \pm 869n$	$113m \pm 97\mu$	$444m \pm 34\mu$	$991m \pm 137\mu$
		$81m \pm 28\mu$	$310m \pm 229\mu$	$692m \pm 47\mu$
3TDEA	$34\mu \pm 536n$	$113m \pm 193\mu$	$445m \pm 83\mu$	$991m \pm 500\mu$
		$113m \pm 19\mu$	$436m \pm 113\mu$	$971m \pm 94\mu$
CAST5-80	$4\mu \pm 572n$	–	–	–

Table A.14: Size of the `.text` section for other ciphers on MIPS32

Order	DES		2TDEA		3TDEA		CAST5-80
0	16		36		50		5
1	34	25	81	68	94	92	–
2	57	34	142	110	153	149	–
3	92	46	238	176	251	244	–

Bibliography

[1] Carlisle Adams. The CAST-128 encryption algorithm. RFC-2144, https://www.ietf.org/rfc/rfc2144.txt, 1997. 2.8, 3.1

[2] Carlisle M Adams. Constructing symmetric ciphers using the CAST design procedure. In *Selected Areas in Cryptography*, pages 71–104. Springer, 1997. 3.1

[3] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2013. 2.2.3, 13, 2.5.1, 3.3.3

[4] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM side–channel. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 29–45. Springer, 2002. 1.1, 1.1.1

[5] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970. 1.5.1

[6] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 174, 1998. 7, 3.1

[7] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014. 3.3.3

[8] Achiya Bar-On. A 2^{70} attack on the full MISTY1. Technical report, IACR Cryptology ePrint Archive, 2015, 746. http://eprint.iacr.org, 2015. 3.1, 3.1

[9] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[10] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference*, pages 230–235. ACM, 2011. 3.3.3

[11] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013. 3.1

[12] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology–CRYPTO'97*, pages 513–525. Springer, 1997. 1.2.1

[13] BlackHat 2015. *Web Timing Attacks Made Practical*, 2015. 1.1

[14] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In *International Conference on Financial Cryptography*, pages 162–181. Springer, 2003.

[15] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004. 1.1.1, 1.1.1

[16] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. 1.1

[17] Ran Canetti, Eyal Kushilevitz, Rafail Ostrovsky, and Adi Rosén. Randomness vs. fault-tolerance. In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 35–44. ACM, 1997. 19

[18] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain. Higher-order masking schemes for s-boxes. In *Fast Software Encryption*, pages 366–384. Springer, 2012. 1.3.1, 6

[19] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances*

in Cryptology–CRYPTO'99, pages 398–412. Springer, 1999. 1.1.1, 1.1.2, 1.3, 1.3.1, 6, 1.4

[20] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy*, pages 191–206. IEEE, 2010. 1.1

[21] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Advances in Cryptology–EUROCRYPT 2014*, pages 441–458. Springer, 2014. 1.3.1, 6, 2.1

[22] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In *Cryptographic Hardware and Embedded Systems–CHES 2000*, pages 231–237. Springer, 2000. 6

[23] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 188–205. Springer, 2014. 1.3.1, 17, 1.3.5, 6, 2.1

[24] Jean-Sebastien Coron, Johann Groszschaedl, Praveen Kumar Vadnala, and Mehdi Tibouchi. Conversion from arithmetic to boolean masking with logarithmic complexity. Technical report, Cryptology ePrint Archive, Report 2014/891, `http://eprint.iacr.org`, 2014. 1.3.1, 17, 1.3.5, 2.1

[25] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. *Side channel cryptanalysis of a higher order masking scheme*. Springer, 2007. 6

[26] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In *Cryptographic Hardware and Embedded Systems–CHES 2003*, pages 89–97. Springer, 2003. 1.3.5

[27] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: NOEKEON. In *First Open NESSIE Workshop*, pages 213–230, 2000. 3.1

[28] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Information Security and Cryptography. Springer, 2002. 3.1

[29] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 107–121. Springer, 2012. 1.3.5

[30] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973.

[31] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. 1.5.1, 14

[32] Roger Forster. Manchester encoding: opposing definitions resolved. *Engineering Science and Education Journal*, 9(6):278–280, 2000. 1.2.3

[33] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261. Springer, 2001. 1.1.1

[34] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*, pages 444–461. Springer, 2014. 1.1

[35] Jovan Dj Golić. Techniques for random masking in hardware. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54(2):291–300, 2007. 1.3.1, 17

[36] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In *Cryptographic Hardware and Embedded Systems–CHES 2001*, pages 3–15. Springer, 2001. 1.3.5, 6

[37] Louis Goubin and Jacques Patarin. DES and differential power analysis the "duplication" method. In *Cryptographic Hardware and Embedded Systems*, pages 158–172. Springer, 1999. 1.3, 1.4

[38] Louis Goubin and Jacques Patarin. Procédé de sécurisation d'un ensemble électronique de cryptographie a clé secrete contre les attaques par analyse physique. *European Patent, Schlumberger*, 1999. 1.3, 1.4

[39] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bon-Seok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong,

et al. HIGHT: A new block cipher suitable for low-resource device. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 46–59. Springer, 2006. 3.1

[40] NA Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001. 1.1.1

[41] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David Wagner. Private circuits II: Keeping secrets in tamperable circuits. In *Advances in Cryptology-EUROCRYPT 2006*, pages 308–327. Springer, 2006.

[42] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003*, pages 463–481. Springer, 2003. 1.3.1, 1.4, 1.4.1, 2.1

[43] ISO/IEC 18033-3:2010 – Information technology – security techniques – encryption algorithms – part 3: Block ciphers. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=54531`. 3.1

[44] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 291–302. Springer, 2002. 1.2.2, 1.2.3

[45] Mohamed Karroumi, Benjamin Richard, and Marc Joye. Addition with blinded operands. In *Constructive Side-Channel Analysis and Secure Design*, pages 41–55. Springer, 2014. 1.3.1, 17, 4, 1.3.5, 2.1

[46] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973. 14

[47] Chong Hee Kim, Jong Hoon Shin, Jean-Jacques Quisquater, and Pil Joong Lee. Safe-error attack on SPA-FA resistant exponentiations using a hw modular multiplier. In *International Conference on Information Security and Cryptology*, pages 273–281. Springer, 2007. 1.2.2, 1.2.3

[48] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology–CRYPTO'99*, pages 388–397. Springer, 1999. 1.1, 1.1.1, 1.1.1

[49] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011. 1.1, 1.1.1, 1.1.1, 1.1.1, 1.2

[50] Paul C Kocher, Joshua M Jaffe, and Benjamin C Jun. Using unpredictable information to minimize leakage from smartcards and other cryptosystems, December 4 2001. US Patent 6,327,661. 1.1.2

[51] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973. 17

[52] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. *Smartcard*, 99:9–20, 1999. 1.4

[53] David W Kravitz. Digital Signature Algorithm, July 27 1993. US Patent 5,231,668. 1.1.1

[54] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004. 1.5.1, 2

[55] Jaeil Lee, Jongwook Park, Sungjae Lee, and Jeeyeon Kim. The SEED encryption algorithm, 2005. 3.1

[56] Paolo Maistri, Pierre Vanhauwaert, and Régis Leveugle. A novel double-data-rate AES architecture resistant against fault injection. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 54–61. IEEE, 2007. 1.2.3

[57] Mitsuru Matsui, S Moriai, and J Nakajima. A description of the Camellia encryption algorithm, 2004. 3.1

[58] Mitsuru Matsui and Hidenori Ohta. A description of the MISTY1 encryption algorithm. RFC-2994, `https://www.ietf.org/rfc/rfc2994.txt`, 2000. 3.1

[59] Thomas Messerges. Using second-order power analysis to attack DPA resistant software. In *Cryptographic Hardware and Embedded Systems–CHES 2000*, pages 27–78. Springer, 2000.

[60] Thomas S Messerges. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption*, pages 150–164. Springer, 2001. 1.1.2, 1.3, 1.3.1, 6, 6

[61] U.S. Department of Commerce National Bureau of Standards. FIPS PUB-46, Data Encryption Standard (DES), 1977. 3.1

[62] U.S. Department of Commerce National Bureau of Standards. Federal Information Processing Standards Publication FIPS-46-3, 1999. 3.1

[63] U.S. Department of Commerce National Bureau of Standards. FIPS PUB-186-4, Digital Signature Standard, (DSS), 2013. 1.1.1

[64] National Institute of Standards and US Department of Commerce (November 2001) Technology. FIPS PUB-197, Advanced Encryption Standard (AES). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. 3.1

[65] Olaf Neiße and Jürgen Pulkus. Switching blindings with a view towards IDEA. In *Cryptographic Hardware and Embedded Systems–CHES 2004*, pages 230–239. Springer, 2004. 1.3.5

[66] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 77–88. Springer, 2003. 1.2.1

[67] Bart Preneel. NESSIE portfolio of recommended cryptographic primitives. `http://www.nessie.org`, 2003. 3.1

[68] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959. 1.5.1

[69] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Advances in Cryptology–EUROCRYPT 2013*, pages 142–159. Springer, 2013.

[70] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer, 2001. 1.1.1

[71] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology–CRYPTO 2015*, pages 764–783. Springer, 2015. 1.3.1

[72] Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. Block ciphers implementations provably secure against second order side channel analysis. In *Fast Software Encryption*, pages 127–143. Springer, 2008. 1.3.1, 6

[73] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 413–427. Springer, 2010. 1.3, 1.3.1, 6, 6

[74] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple photonic emission analysis of AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 41–57. Springer, 2012. 1.1

[75] Kai Schramm and Christof Paar. Higher order masking of the AES. In *Topics in Cryptology–CT-RSA 2006*, pages 208–225. Springer, 2006. 1.1.1, 1.1.2, 1.3.1, 6

[76] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979. 1.4.1

[77] Yosuke Todo. Integral cryptanalysis on full MISTY1. In *Annual Cryptology Conference*, pages 413–432. Springer, 2015. 3.1

[78] David J Wheeler and Roger M Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995. 1.3.2, 17

[79] David J Wheeler and Roger M Needham. Correction to XTEA. *Unpublished manuscript, Computer Laboratory, Cambridge University, England*, 1998. 1.3.2, 17, 3.1

[80] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999. 1.2.3

[81] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on computers*, 49(9):967–970, 2000. 1.2.2

[82] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. RSA speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Transactions on computers*, 52(4):461–472, 2003. 1.2.2, 1.2.3