Leopold-Franzens-Universität Innsbruck

Institute of Computer Science
Information Security

Defending against power analysis by balancing binary values:
a compiler based approach
Master Thesis

Alexander Schlögl

advised by
Univ. Prof. Dr. Rainer Böhme

Innsbruck, July 29, 2019

Leopold-Franzens-Universität Innsbruck

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig verasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

Power consumption can leak information about processed data. If the data in question is the secret key for a cryptographic algorithm, this can have dire consequences for security. By supplying a large number of known inputs to the algorithm, e.g. plaintexts for an encryption, and analyzing the power consumption, an attacker can infer the value of this secret key.

Many use cases for embedded devices, like SmartCards and NFC keys, perform cryptographic operations on any input sent to them. This makes supplying these plaintexts very easy. Their self contained nature also makes it easy for an attacker to measure their power consumption in a controlled environment, and a lack of parallelism reduces the difficulty of their analysis. With this in mind, embedded devices are easy and potentially valuable targets for power analysis attacks.

In order to have some amount of security for stored keys, various countermeasures against power analysis have been explored. Defenses either try to add additional factors to the power consumption, increasing the amount of computational effort required for a successful attack, or reduce the influence of data on the power consumption altogether. Some defenses only require changes to the software, but are algorithm specific and need to be explicitly implemented. This requires developers with significant amounts of security knowledge, which is often not the case.

Other defenses are applicable to any algorithm in general, but require significant changes to the hardware. One such hardware defense, called dual-rail logic, works by *balancing* every bit with its inverse. This keeps power consumption constant under the assumption that it is directly proportional to the number of 1 bits in a value (the Hamming weight), which is a widely used model for power analysis attacks.

In my thesis I explore a balancing scheme, similar to dual-rail logic, done in software. This should provide increased robustness against power analysis attacks for *any* code, while still being able to run on “off the self” hardware without specialized circuitry. The balancing is done by storing 8 bit values together with their inverse, and storing both in the same 32 bit register. I also provide an arithmetic for performing all operations required for a modern RISC architecture on these balanced values, compositioning the new variants from existing operations.

By providing a plugin for the LLVM compiler, I also provide an automatic transformation from any code written for 8 bit architectures into my balanced form, without requiring additional work from the developer. As such my defense requires neither specialized hardware, nor developers with expert security knowledge. The versatility of the LLVM compiler also makes my transformation applicable to many different source languages and target platforms, additionally increasing the generality of this approach.

Evaluation of my arithmetic shows a reduction in the variance of Hamming weights during execution, which should reduce the information an attacker can gain by analyzing the power consumption. The balancing is not perfect, as compositioning my balanced arithmetic from existing ALU operations forces imbalanced intermediate values. Even with these forced imbalances the reduction in Hamming weight variance is significant, suggesting increased robustness against real-world attackers, without additional hardware or implementation cost.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Unintended signal emissions are a major source of information leakage in modern processors [9]. If information about cryptographic secrets is leaked in such a way, this can have a massive security impact. One of these so-called side channels that is especially easy to measure is power consumption. Due to the way transistors work, switching a bit from 1 to 0 (and vice versa) requires some amount of power [19]. This means that by observing the power consumption an attacker can infer the number of switched bits at a given time, and thus gain information about processed data. When attacking the power consumption side channel, this connection between data and power consumption is often reduced to the Hamming weight model, which assumes the power consumption is proportional to the number of 1s in processed data [6]. By measuring the power consumption traces during execution an attacker can gain information about the Hamming weight of processed data. If an attacker knows which cryptographic operation is being performed and can control the input (both reasonable assumptions for embedded devices), she can infer the value of the cryptographic secret via statistical analysis of the power traces [6].

Restricted physical access often prohibits capturing power traces, and even when measurements are possible the high degree of parallelism in modern hardware adds a lot of random noise to the power consumption. However, this is not the case for embedded devices. Many of their use cases (SmartCards, etc.) make restricting physical access impossible and even *require* handing them over to users. An attacker can thus fully control all inputs, measure all outputs, and especially control and monitor the power supply. The processors used in embedded devices are also often very simple, to keep production costs minimal. A low clock-rate, no parallelism, and a lack of caching mean that captured power traces contain very little noise. This makes embedded devices relatively easy and often valuable targets for power analysis attacks.

In order to protect the cryptographic secrets on embedded devices, defenses against power analysis have been amply explored. However, the most commonly used defenses are either algorithm-specific, like masking, or require significant changes to the hardware, like dual-rail logic[36]. Dual-rail logic is especially notable because it is algorithm independent. By computing the inverse along with the actual value, dual-rail logic *balances* the number of 1s in intermediate values, and thus keeps the power consumption constant. This makes it in theory impossible for an attacker to gain information via the power consumption.

Unfortunately this strategy suffers from multiple engineering problems, such as tiny differ-

ences in clock timings between the regular and inverted path[2], or variances in the production of transistors[28]. It also requires a significant increase in circuit size, doubling the required size or more[2]. While this increase alone would probably be tolerable, implementing dual-rail logic requires *specialized* hardware, which explodes the cost compared to off-the-shelf hardware. Even with these caveats, dual-rail logic still has the benefit that *any* code can be run on the modified circuitry, without any alterations, while still experiencing increased security against power analysis.

For the rest of my thesis I adopt the terminology in works on dual-rail logic, which refer to the difficulty for an attacker as robustness, instead of as security [35, 28].

In my thesis I explore the possibilities of implementing similar balancing in software. It works by only using part of the available word size for actual data, leaving the rest for balancing. Specifically, I store 8 bit values, along with their inverse, in a 32 bit register, causing their Hamming weight to be constant. I propose an arithmetic on these balanced values, giving a replacement for all integer operations required for a modern RISC instruction set. With this arithmetic and the balanced values, one can then execute *any* program, without special modifications, while theoretically benefiting from an increase in robustness against power analysis attacks.

I design and implement a plugin for the LLVM compiler that transforms code written for 8 bit architectures into this balanced form. With this plugin the additional work required from a developer to benefit from this robustness increase is kept to a minimum, as balancing variables and using my balanced arithmetic operators is done automatically. If the original program uses at most 8 bit values, it does not require any changes at all. Due to the modular nature of LLVM, my transformation can be applied to many different source languages and target platforms. The plugin itself is also highly modular, and testing different balanced operators can be done without touching the plugin.

Finally, I evaluate my proposed system consisting of the arithmetic, the balancing plugin and the build process used for this toolchain. In order to do this, I instrument the QEMU emulator. I extend it by calculating the Hamming weight of every emulated machine instruction, and building a histogram during execution. This loses any temporal information, but is necessitated by the complexity and many layers of indirection in QEMU's build process. The captured histograms do not guarantee increased robustness, however they give some indication of the effectiveness of the general approach, and of my concrete implementation.

The rest of this thesis is organized as follows. Chapter 2 explains different variants of power analysis attacks and defenses against it. Chapter 3 gives an overview of related work on software approaches to power analysis defense, approaches to dual-rail logic, and related security work utilizing LLVM. In Chapter 4 I explain the design of my balancing and my balanced arithmetic, and discuss the evaluation of the balanced operations in it. Chapter 5 covers the implementation details of my balancing pass. The procedure for evaluation is explained in Chapter 6. Both Chapter 5 and Chapter 6 give a brief introduction to their required tools, respectively LLVM and QEMU. In Chapter 7 I present my results, and Chapter 8 offers a summary and discussion of my thesis.

Chapter 2

Power Analysis

Power analysis utilizes the fact that different operations have different power consumptions. By capturing the power consumption traces (power traces in short) and examining them, an attacker can reason about the variables determining the control flow during program execution. If these variables are cryptographic secrets, or keys (e.g. private keys in RSA), they are leaked to the attacker in what is called a simple power analysis (SPA) attack[18]. In many cases the control is not related to the key, requiring more complex attacks. For these cases capturing a large number of traces and performing statistical analysis can leak information about *individual values*. The two main attacks of this type are called differential power analysis (DPA)[18] and correlation power analysis (CPA)[6]. All variants and the setup required are explained in this section.

2.1 Setup

Performing power analysis requires access to the device in a controlled environment. The attacker needs to control the power supply for the target, and be able to alter the containing electronics. This is fairly easy for embedded platforms, as they are often made to be self-contained. Even for more complex targets like IoT devices, the processor can be removed from its circuit board and put in an attack environment[31].

The setup for a power analysis attack is as follows: An attacker solders a resistor between the target processor and the ground connector of its power supply. She then measures the voltage difference between both ends of the resistor with an oscilloscope. This voltage is directly proportional to the current flowing through the resistor, and thus to the power consumption of the target. The voltage recordings from the oscilloscope are then transferred to a computer, and can be analyzed there.

2.2 Simple Power Analysis

In the simplest form of power analysis, SPA, power traces are usually examined visually. Repeating patterns of operations can often be identified, leaking information about the control flow. If the control flow is dependent on a key, an attacker can infer its value. An example target would be RSA decryption being calculated via the square-and-multiply algorithm. The

difference between the multiply and the square operation is directly observable from power traces. As the order of these operations is linked to the private key, identifying the control flow leaks the private key. Figure 2.1 shows a trace for square-and-multiply in RSA decryption, including the leaked private key bits.

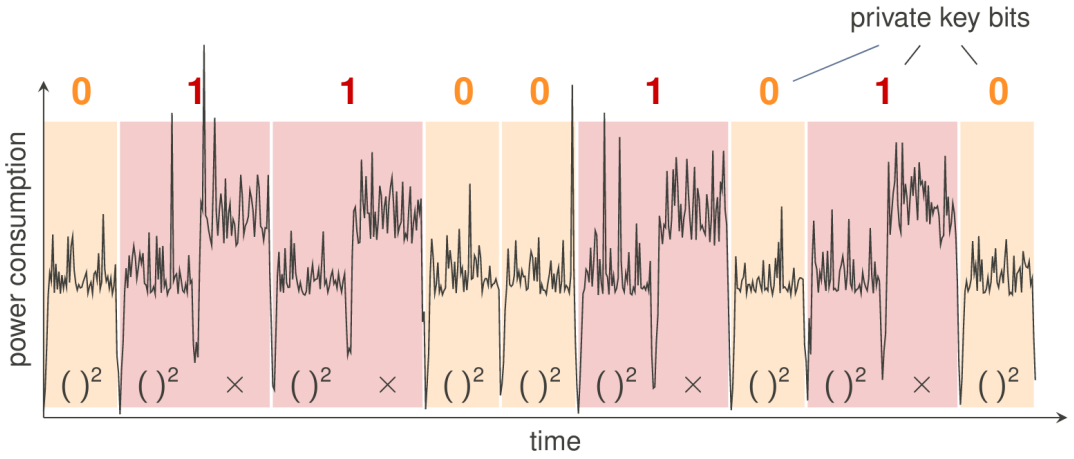


Figure 2.1: Simple power analysis on square-and-multiply RSA[5]

Power traces are often not as clear as in Figure 2.1, and contain some amount of random noise. By averaging multiple traces this noise can be reduced, allowing for easier analysis.

2.3 Differential Power Analysis

The control flow is often not enough to leak the entire key, and it is very hard to gain information about the actual data from only SPA. For this a more complex variant of power analysis can be used, namely *Differential power analysis* (DPA). DPA requires a large number of power traces, with the cryptographic algorithm being run on a known plaintext in each execution. It also requires that the attacker knows which cryptographic algorithm is running on the target. Both of these requirements are usually fulfilled in embedded devices.

The attacker additionally requires a *power model*, which is used to calculate the expected power consumption for a given value. While not entirely accurate[6], the Hamming weight has proven to be a very effective power model in practice.

Knowing the algorithm running on the target, the attacker can predict the result of intermediate computations, given some assumption about the key. In practice she usually guesses a single key bit, and then calculates the expected result of some bitwise operation, e.g. XOR. The other bits are not important for her attack at the moment and are thus ignored. After calculating the expected power consumption for all plaintexts, she splits the power traces in two subsets, based on the value of the expected result. Then she calculates the mean of both sets, and calculates their difference. If the guess for the current key bit was correct, then all traces in a single subset will have the same value for the current bit. This means that the difference between both means shows spikes in the places where the attacked operation happens.

If the guess was wrong on the other hand, the values of the current bit are randomly distributed in both sets, and the difference will show no spikes of non-zero values. The values of the other bits can be ignored, as they are close enough to being random in both sets, and thus filtered out by calculating the difference. Other power consumption noise, coming from different parts of the processor, is randomly distributed over every trace, and thus filtered by taking the mean. Both the values for other bits and other power consumption influences are note entirely random, but close enough to it that these assumptions work well in practice.

Figure 2.2 shows a typical DPA result with the mean power consumptions of both sets, the difference between the two, and the difference with the Y axis magnified by a factor of 15. This analysis was performed on the output of the least significant bit after the first S-box substitution in AES.

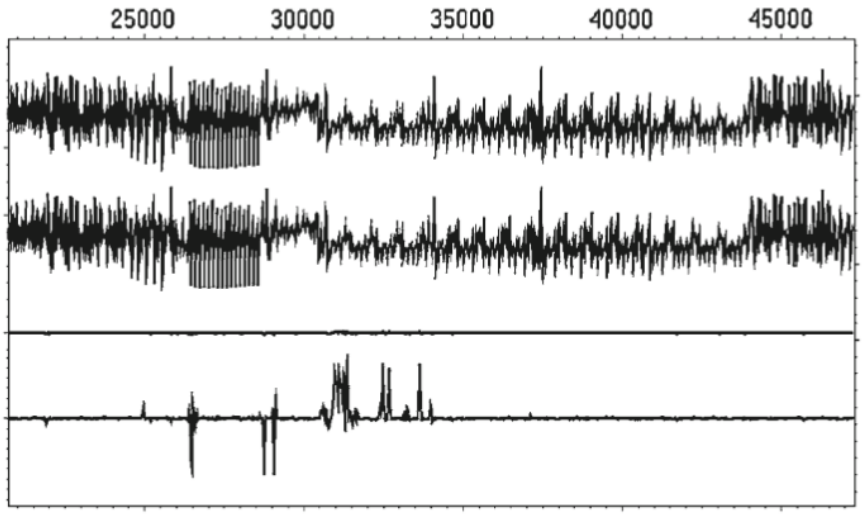


Figure 2.2: Difference in means during DPA[20]

2.4 Correlation Power Analysis

CPA is the most complex of these attacks, but also offers the best results. The attacker starts by making a list of candidate values for a part of the key, usually individual bytes. With a power model, usually the Hamming weight like for DPA, she can calculate the expected power consumption for combination of key guess with every plaintext.

For a fixed key guess, these expected power consumptions are a function over all plaintexts. For a single point in time, the actual power traces are *also* a function over all plaintexts, with the actual key being a fixed parameter unknown to the attacker. By testing how much the expected power consumptions correlate with the actual power consumptions, the attacker can find the confidence values of the guesses for the current part of the key. She then assumes the value with the highest correlation coefficient is correct, and continues her attack for the rest of the key.

Of course the attacker does not know exactly which point in time she is supposed to examine. However, it is very unlikely that an incorrect key guess at a random point in time has a closer correlation than the correct key guess at the correct time, and so an attacker can simply use the maximum correlation coefficient.

2.5 Defenses Against Power Analysis Attacks

Defenses against this class of attack usually work by either adding additional factors to the power consumption, thus increasing the computational effort required for analysis, or by reducing variances in power consumption altogether. This reduced variance decreases the information an attacker can gain from the same number of power traces, giving her reduced confidence in her result or requiring her to capture more traces.

Masking[12][8] for example is an algorithm specific defensive measure that adds a third factor to the power consumption by first performing adding a masking value to the plaintext via an invertible operation. The cryptographic algorithm then works on the masked value, and only in the end unmasks the result. As such the attacker has to calculate her correlation for each possible combination of key byte and mask value. This increases the number of traces she needs to capture (to still provide the same confidence in her analysis) and the computation time of her analysis.

Other defensive measures focus on creating a worse signal to noise ratio for the entire power consumption. One technique that has gained a lot of traction is dual-rail logic[36]. It works by calculating the inverse of every intermediate result along with the actual result, thus balancing the number of 1s. This results in a constant Hamming weight and therefore a data-independent power consumption.

Unfortunately, dual-rail logic suffers from multiple engineering problems. The power required to set the value of a bit to 1 is dependent on properties of the underlying transistors, which are subject to variances in manufacturing.[28] Minimal differences in clock timings between both paths can also reduce the security of dual-rail logic[2]. Storing the inverse also requires significantly larger circuitry, doubling the circuit size or more[2].

Even with these caveats, dual-rail logic has the major advantage that once it is applied, *any* code can be run without modifications while still benefiting from the increased robustness.

Chapter 3

Related Work

I considered related work from three major areas: Software based power analysis defenses that can in theory be applied to any algorithm, work on dual-rail logic, and security related research utilizing LLVM.

First introduced by Messerges[25], masking is in theory applicable to any algorithm. In his paper he identified the core operations in the candidates algorithms for AES. He then devised a way to apply a masking to the input of these operations, in such a way that no intermediate value is stored in memory without being masked. For example, his mask for boolean operations was applied via XOR: $x_{masked} = x \oplus r$, where r is a randomly selected mask. Boolean operations can then be applied to x_{masked}, and intermediate results do not directly give an attacker information about any secrets in the algorithm. An attacker can still try to extract both r and the secret key at the same time, however this exponentially increases her required effort.

As not only boolean operations are used, an additional masking scheme for arithmetic operations is required. Additionally, a conversion method between both masking variants is needed, and this transformation can at no point store x as an intermediate value. Messerge's approach stores either x or $\overline{x}$, depending on a random value.

While this provides adequate robustness against DPA attacking a single bit of the key, it is insufficient for attacks on multiple bits, as shown by Coron et. al.[8]. By attacking two bits at the same time, they could find whether both bits were identical or not, and thus gradually reduce the number of possible keys.

Other work has been done to avoid leaks such as this[1][30], however all of these approaches are currently algorithm specific, with most work being targeted at AES. The decomposition from [25] could be used to automate this masking process for algorithms utilizing these basic operators, and the masking could then be applied automatically during compilation.

Another software approach is inspired by secret sharing. Goubin et. al.[14] used their so-called "Duplication" method to defend DES against power analysis. They did this by splitting the input v into multiple parts $v_1, ..., v_n$ using XOR, then performing all operations on the individual parts, and finally recombining them into the encrypted v'. This works because all operations of DES are invariant to the XOR split performed on v.

The approach was generalized by Chari et. al.[7] using the identity $v = f(v_1, v_2, ..., v_n)$. They state that this secret sharing method can then be applied to any algorithm, as long as

no operation invalidates this identity. For example, if one were to use XOR for f as for DES, the Galois Field multiplication in AES would result in an invalid v'.

As such, while the general approach is applicable to all algorithms, the choice of f depends on the present operations, and thus prohibits automated application of this method. However, given multiple candidates for f and a set of operators invariant for each candidate, one could perform static analysis of a program and try to find an f that can tolerate all operations. Some algorithms could then automatically be made more robust using secret sharing.

dual-rail logic related work is mostly concerned with solving the engineering challenges mentioned in Chapter 2. Balancing the power consumption via $\overline{x}$ was first proposed by Saputra et. al.[33], where they loaded negations of memory values into a dummy register. This reduced the effect the Hamming weight of data had on the power consumption, and thus reduced the information leakage. Sokolov et. al.[36] combined the security benefits of this method with Dual-Rail encoding in general, which had previously been used in circuits without an explicit clock signal[26].

Sokolov et. al. use the fact that both valid code words in Dual-Rail encoding have the same Hamming weight. By encoding every bit with this encoding they drastically reduce the effect of data on the power consumption.

There are multiple papers that use the powerful analysis infrastructure built around LLVM's intermediate representation (LLVM IR) for security. One such paper is by Junod et. al.[17], which focuses on automatically obfuscating code during compilation. They insert bogus instructions and replace existing instructions with equivalent ones ($a+b = a-(-b)$), all in LLVM IR. This is very similar to what my balancing pass does, albeit with a different goal. They also change the control flow to increase the difficulty of reverse engineering, however in a constant fashion. If the changes were dynamic and different between multiple executions, their pass might also increase robustness against power analysis.

Lyu et. al.[24] also use LLVM and QEMU to analyze binaries for different platforms. By using QEMU to translate machine code into QEMU's intermediate representation, and then translating that into LLVM's intermediate representation, they can utilize the plethora of analysis tools available for LLVM.

Chapter 4

Theory

As previously discussed, power analysis attacks are usually performed under the Hamming weight power model. With this model, values with identical Hamming weights are indistinguishable. Having only perfectly balanced values then means that an attacker can gain no information via the power consumption, as all values look exactly the same.

As having *only* balanced values is not possible with my choice of balancing (which will be justified in Sections 4.1 and 4.2), the goal then would be to come as close to this ideal as possible, i.e. having a minimal number of unbalanced values. However, due to time constraints I do not try to find the optimal balancing scheme in my thesis, instead finding a balancing scheme and an arithmetic that is *correct*, and balanced enough to measurably impact power consumption.

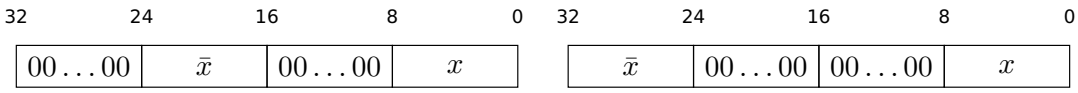
4.1 Balancing Individual Values

In resemblance of dual-rail logic itself, I balance the Hamming weight by storing the inverse in the same register as the actual value. I store only 8 bit of actual data, as code written for 8 bit architectures is much more common than 16 bit architectures. This also gives me 16 bit in the same register to reduce imbalances during operations.

The next step is deciding on a way to store the values. While bit-interleaving schemes would have been possible, and might have performed better balancing wise, I decided instead to store all data bits in a single byte in the register. This allows me to use operations in the ALU, instead of having to find complicated workarounds, especially for arithmetic operations like addition, subtraction, and multiplication. As such, the finished balancing contains the actual data byte x and the balancing byte $\overline{x}$ each stored in one of the four bytes of a register.

With that fixed the only remaining decision is where in the register to put x and $\overline{x}$. I wanted to have room between x and $\overline{x}$ for shifts, so this left only 2 candidates (and their inverses) for what I call the *balancing schemes*. Figure 4.1 shows the two schemes I chose for my project.

I found balanced operations for both schemes, but in the end decided to use Scheme 1 as a default because it exhibits nicer behavior for shifts. Both are worth mentioning however, because many of my operations will result in values formatted in Scheme 2 and require explicit transformation. By finding standardized transformations in both directions I could reuse them in the rest of my arithmetic.



(a) Balancing Scheme 1 (b) Balancing Scheme 2

Figure 4.1: Balancing Schemes

4.2 Balancing Binary Operations

Any operations applied to balanced values should also return balanced values. With my balancing schemes, this requires not only calculating the result, but also its inverse. This inverse is also the reason why there *must* be imbalanced intermediate values.

For many operators (denoted by $\circ$), $\overline{x \circ y}$ is not $\overline{x} \circ \overline{y}$. As an example, $\overline{x \wedge y} = \overline{x} \vee \overline{y} \neq \overline{x} \wedge \overline{y}$, using DeMorgan's law. This means that for calculating balanced binary and (AND), binary or (ORR) has to be applied at some point. It is generally not possible to execute different ALU operations on different parts of the register. As such, both AND and ORR must be executed at some point, resulting in imperfectly balanced intermediates.

Again my goal is not to reduce these imbalanced intermediates to a minimum, but to reduce them enough to impact power consumption. As such, no claims on the optimality of my operations is made, both in terms of balancing and the number of intermediate operations. They are, however, correct. Their derivation is a formal notion of this fact, and I also exhaustively search all possible combinations of operands for incorrect values, as discussed in Section 4.3.

The notation for my operators is as follows. For every operation I give the intermediate steps, with a single line denoting an intermediate value. The values are in the form

$$\%i = x_1 \quad \| \; x_2 \quad \| \; x_3 \quad \| \; x_4 \quad | \; \texttt{operation}$$

where $\%i$ denotes the "name" of the current intermediate, and x_1 through x_4 are the individual bytes of a register, with x_1 having the most significant, and x_4 the least significant bits. The `operation` denotes how the current value is obtained, and is missing for input operands.

Transforming Scheme 1 to Scheme 2

The transformation from Scheme 1 to Scheme 2 looks as follows:

$$\begin{array}{llll|l}
\%1 = 0 & \| \; \overline{x} & \| \; 0 & \| \; x & \\
\%2 = \overline{x} & \| \; \overline{x} & \| \; x & \| \; x & | \; \%1 \; \text{LSL} \; 8 \\
\%3 = \overline{x} & \| \; 0 & \| \; 0 & \| \; x & | \; \%2 \; \texttt{AND 0xff0000ff}
\end{array}$$

LSL here stands for logical shift left.

Transforming Scheme 2 to Scheme 1

The other direction works very similar to the first, and is shown below. Note that ROR stands for rotational right shift, i.e. the values shifted out on the right are shifted back in on the left.

$$
\begin{array}{llll}
\%1 = \overline{x} & \| \ 0 & \| \ 0 & \| \ x \\
\%2 = \texttt{0xff} & \| \ \overline{x} & \| \ 0 & \| \ x & | \ \%1 \ \texttt{ORR} \ (\%1 \ \texttt{ROR} \ 24) \\
\%3 = 0 & \| \ \overline{x} & \| \ 0 & \| \ x & | \ \%2 \ \texttt{AND} \ \texttt{0x00ff00ff}
\end{array}
$$

ORR

Before finding a balanced variant of bitwise OR, I needed to find an expression for the inverse of the result. For this I utilized DeMorgan's law: $\overline{x \vee y} = \overline{x} \wedge \overline{y}$. With this equality ORR looks as follows:

$$
\begin{array}{llllll}
\%1 = 0 & \| \ \overline{x} & \| \ 0 & \| \ x & & \\
\%2 = 0 & \| \ \overline{y} & \| \ 0 & \| \ y & & \\
\%3 = 0 & \| \ \overline{x} \ \texttt{ORR} \ \overline{y} & \| \ 0 & \| \ x \ \texttt{ORR} \ y & | \ \%1 \ \texttt{ORR} \ \%2 \\
\%4 = 0 & \| \ \overline{x} \ \texttt{AND} \ \overline{y} & \| \ 0 & \| \ x \ \texttt{AND} \ y & | \ \%1 \ \texttt{AND} \ \%2 \\
\%5 = \overline{x} \ \texttt{AND} \ \overline{y} & \| \ \overline{x} \ \texttt{ORR} \ \overline{y} & \| \ x \ \texttt{AND} \ y & \| \ x \ \texttt{ORR} \ y & | \ \%3 \ \texttt{ORR} \ (\%4 \ \texttt{LSL} \ 8) \\
\%6 = \overline{x \ \texttt{ORR} \ y} & \| \ 0 & \| \ 0 & \| \ x \ \texttt{ORR} \ y & | \ \%5 \ \texttt{AND} \ \texttt{0xff0000ff} \\
\%7 = 0 & \| \ \overline{x \ \texttt{ORR} \ y} & \| \ 0 & \| \ x \ \texttt{ORR} \ y & | \ \texttt{transform\_2\_1}(\%6)
\end{array}
$$

AND

As $\overline{x \wedge y} = \overline{x} \vee \overline{y}$, AND works almost the same as ORR, but uses different parts of the intermediate results.

XOR

XOR is at its base a combination of AND and ORR: $x \oplus y = (\overline{x} \wedge y) \vee (x \wedge \overline{y})$. It is better to create a balanced XOR from scratch, instead of compositioning it from ORR and AND, because both ORR and AND have the same imbalanced intermediate values.

The inverse of the result can be found through repeated application of DeMorgan's law and simplification. I will skip the details of this simple transformation, and show only the result: $\overline{x \oplus y} = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$.

$\%1 = 0$	$\parallel \overline{x}$	$\parallel 0$	$\parallel x$	
$\%2 = 0$	$\parallel \overline{y}$	$\parallel 0$	$\parallel y$	
$\%3 = \overline{x}$	$\parallel \overline{x}$	$\parallel x$	$\parallel x$	$\mid \%1$ ORR $(\%1$ LSL $8)$
$\%4 = y$	$\parallel \overline{y}$	$\parallel \overline{y}$	$\parallel y$	$\mid \%2$ ORR $(\%2$ ROR $24)$
$\%5 = \overline{x}$ AND y	$\parallel \overline{x}$ AND $\overline{y}$	$\parallel x$ AND $\overline{y}$	$\parallel x$ AND y	$\mid \%3$ AND $\%4$
$\%6 = x$ XOR y	$\parallel \overline{x \text{ XOR } y}$	$\parallel x$ XOR y	$\parallel \overline{x \text{ XOR } y}$	$\mid \%5$ ORR $(\%5$ ROR $16)$
$\%7 = \overline{x \text{ XOR } y}$	$\parallel x$ XOR y	$\parallel \overline{x \text{ XOR } y}$	$\parallel x$ XOR y	$\mid \%6$ ROR 8
$\%8 = \overline{x \text{ XOR } y}$	$\parallel 0$	$\parallel 0$	$\parallel x$ XOR y	$\mid \%7$ AND `0xff0000ff`
$\%9 = 0$	$\parallel \overline{x \text{ XOR } y}$	$\parallel 0$	$\parallel x$ XOR y	$\mid$ `transform_2_1`$(\%8)$

With this construction, XOR has *no imbalanced intermediate values*. It is the only operator that is perfectly balanced. Unfortunately it is not possible to transform arbitrary logic into XORs, as XOR is not a universal operation. However, it is a very common operation in cryptographic algorithms, and as such having perfectly balanced XOR provides a significant increase in robustness.

ADD

For the inverse of arithmetic operations I utilized the definition of the negation in 2s complement: $-x = \overline{x} + 1$. This also means that $\overline{x} = -x - 1$ and therefore:

$$\overline{x + y} = -(x + y) - 1 = -x - y - 1 = \overline{x} + 1 + \overline{y} \cancel{+1} \cancel{-1} = \overline{x} + \overline{y} + 1$$

Using associativity of addition the balanced variant of ADD looks like the following:

$\%1 = 0$	$\parallel \overline{x}$	$\parallel 0$	$\parallel x$	
$\%2 = 0$	$\parallel \overline{y}$	$\parallel 0$	$\parallel y$	
$\%3 = 0$	$\parallel \overline{x} + 1$	$\parallel 0$	$\parallel x$	$\mid \%1 +$ `0x00010000`
$\%4 = c$	$\parallel \overline{x + y}$	$\parallel c'$	$\parallel x + y$	$\mid \%3 + \%2$
$\%5 = 0$	$\parallel \overline{x + y}$	$\parallel 0$	$\parallel x + y$	$\mid \%4 \wedge$ `0x00ff00ff`

Both c and c' denote possible carry values that need to be filtered.

SUB

For subtraction I again use the definition of 2s complement, giving me the following for the inverse result:

$$\overline{x - y} = -(x - y) - 1 = y - x - 1 = y + (-x - 1) = y + \overline{x} = \overline{x} + y$$

Applying the same definition to the regular result yields

$$x - y = x + \overline{y} + 1$$

resulting in a quick and convenient balanced subtraction:

$$
\begin{array}{llll|l}
\%1 = 0 & \| \ \overline{x} & \| \ 0 & \| \ x & \\
\%2 = 0 & \| \ \overline{y} & \| \ 0 & \| \ y & \\
\%3 = 0 & \| \ y & \| \ 0 & \| \ \overline{y} & | \ \%2 \ \texttt{ROR} \ 16 \\
\%4 = 0 & \| \ y & \| \ c & \| \ \overline{y} + 1 & | \ \%3 + \texttt{0x00000001} \\
\%5 = c' & \| \ \overline{x} + y & \| \ c'' & \| \ x + \overline{y} + 1 & | \ \%1 + \%4 \\
\%6 = 0 & \| \ \overline{x - y} & \| \ 0 & \| \ x - y & | \ \%5 \ \texttt{AND} \ \texttt{0x00ff00ff}
\end{array}
$$

MUL

The inverse result of multiplication can be calculated as follows:

$$\overline{x \cdot y} = -(x \cdot y) - 1 = (-x) \cdot y - 1 = (\overline{x} + 1) \cdot y = \overline{x} \cdot y + y - 1$$

Which gives us the following balanced multiplication:

$$
\begin{array}{llll|l}
\%1 = 0 & \| \ \overline{x} & \| \ 0 & \| \ x & \\
\%2 = 0 & \| \ \overline{y} & \| \ 0 & \| \ y & \\
\%3 = \overline{y} & \| \ 0 & \| \ 0 & \| \ y & | \ \texttt{transform\_2\_1}(\%2) \\
\%4 = c & \| \ \overline{x} \cdot y & \| \ c' & \| \ x \cdot y & | \ \%1 \cdot \%3 \\
\%5 = c'' & \| \ \overline{x \cdot y} + 1 & \| \ c' & \| \ x \cdot y & | \ \%4 + (\%2 \ \texttt{LSL} \ 16) \\
\%6 = c''' & \| \ \overline{x \cdot y} & \| \ c' & \| \ x \cdot y & | \ \%5 + \texttt{0x00ff0000} \\
\%7 = 0 & \| \ \overline{x \cdot y} & \| \ 0 & \| \ x \cdot y & | \ \%6 \ \texttt{AND} \ \texttt{0x00ff00ff}
\end{array}
$$

DIV and REM

As binary division seemed unnecessarily complex for balanced values, I instead implemented it by repeated subtraction. Using my constructs for balanced subtraction and addition (for the result), I simply subtract the divisor until it is larger than the remaining dividend. For division I return the number of subtraction, and for the remainder I simply return the remaining dividend.

This process becomes less trivial for negative numbers, used in `sdiv` and `srem` in LLVM IR. In accordance with the semantics of these instructions in LLVM IR, as specified in the LLVM language reference manual[23], I catch the signs of operands beforehand and set the sign of the result accordingly. The code for signed division is shown in Listing 5.1.

Listing 4.1: Balanced signed division

```
1
2 int balanced_sdiv(int lhs, int rhs) {
3   uint32_t ret = 0x00ff0000;
4
5   uint8_t negative = 0;
6   if(lhs & 0x00000080){
7     negative = !negative;
8     lhs = balanced_negative(lhs);
9   }
```

```
10
11   if(rhs & 0x00000080){
12     negative = !negative;
13     rhs = balanced_negative(rhs);
14   }
15
16
17   while (lhs <= rhs) {
18     lhs = balanced_sub(lhs, rhs);
19     ret = balanced_add(ret, 0x00fe0001);
20   }
21
22   if(negative)
23     return balanced_negative(ret);
24   else
25     return ret;
26 }
```

Shifting

While performing logical shifts, I need to ensure that the correct bits are pushed in. When 0s are shifted in for x I have to shift in 1s for $\overline{x}$, and vice versa. This is done by ORRing the target value with `0xff000000` or `0x0000ff00`, as needed. The shifting is performed normally and the result is then AND filtered with `0x00ff00ff` to comply with Scheme 1 again.

4.3 Testing for Correctness

While the individual steps for each binary operator are themselves a theoretical proof for their correctness, I still wanted to validate them. As there are only 256 possible 8 bit values, I could easily brute-force every combination of them, and verify the correctness of the result. For this purpose I wrote python code that allows execution of intermediate steps. By specifying the individual steps with lambdas, and then constructing the entire balanced operation from unary and binary operations, I can execute the operation step by step. The intermediate results are then stored in *numpy* arrays, allowing me to check if the results are correct, and for which values the results are incorrect, as well as where any errors happen. As an example, Listing 4.2 shows the intermediate steps for multiplication.

Listing 4.2: Step-by-step execution of balanced multiplication

```
1  m = MultiStepOperation([
2      Convert_1_2(1), #2
3      BinaryOperation(0,2, lambda x,y: (x*y) & 0xffffffff), #3 the AND is required due to
           python's arbitrary precision integers
4      BinaryOperation(3,1, lambda x,y: x + (y << 16)), #4
5      UnaryOperation(4, lambda x: x + 0x00ff0000), #5
6      UnaryOperation(5, lambda x: x & 0x00ff00ff), #6
7  ])
8  m.execute()
9  incorrectResults = m.testCorrectness(lambda x,y: (x*y)&0xff)
10 print(                                                              )
```

The *Unary-* and *BinaryOperation* classes take the indices of the layers to operate on (0 and 1 are the inputs, all others are intermediate values), as well as the operation in form of a lambda. Executing the *MultiStepOperation* will then execute all lambdas in order and store the intermediate results in *numpy* arrays. Correctness is then tested by checking if all final results are equal to the output of a reference operation ($x \cdot y$ in this case).

4.4 Evaluating the Balancedness

The balancedness of my operations is evaluated using the same python code. As all intermediate results are stored during evaluation I can easily calculate the distribution of their Hamming weights. Figure 4.2 shows one such set of histograms used for evaluating the performance of the balanced operator, in this case for multiplication. I used these histograms to check if operations needed improvement, and if that was the case, I tried to find a different, more balanced way of performing them.

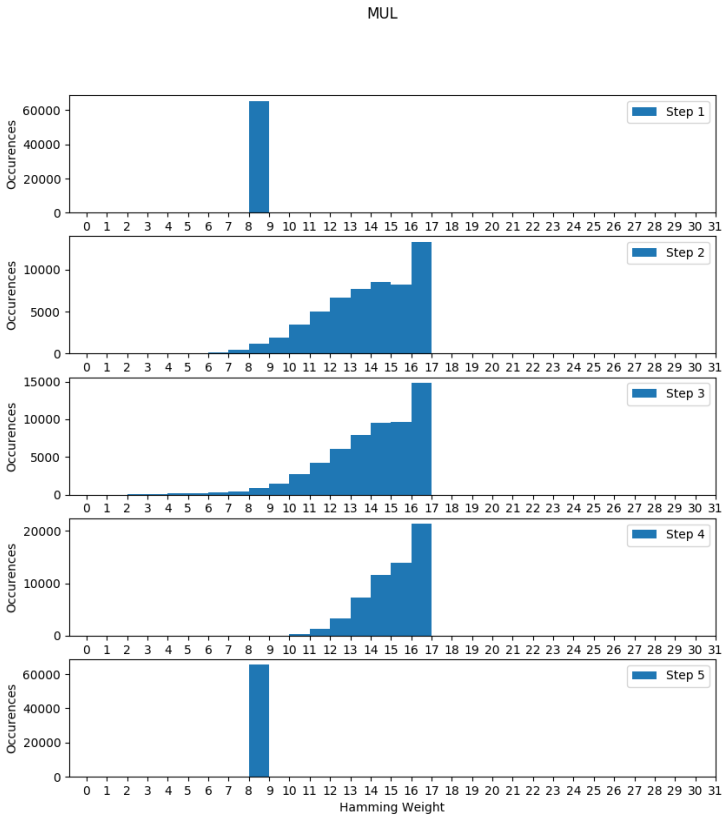


Figure 4.2: Histograms of Hamming weights during balanced multiplication

Chapter 5

Implementation

The transformation of normal code into my balanced form happens automatically inside the compiler. This requires no additional considerations of the programmer, while still providing increased robustness. My thesis is intended as a proof of concept and as such does not balance all variables in the code. It balances only variables on the stack, including function parameters, return values, literals and local variables. I chose the stack as it provides a clean cut with little ambiguity for the programmer, while still balancing enough to decrease variance in Hamming weights by a measurable amount. Figure 5.1 shows all candidates for balancing, as well as their relationship. If one candidate has their value set by another (e.g. result of binary operators being stored in variables), then this introduces a dependency for this balancing. These dependencies are shown as arrows in Figure 5.1. Memory locations (registers, heap memory, etc.) are balanced iff all values stored in them are balanced.

5.1 LLVM

The LLVM compiler infrastructure project[22] contains a number of sub-projects, but it is mostly known for being an extremely versatile compiler. It works by using an intermediate representation (LLVM IR) specifically designed to be source and target independent, while allowing for easy automatic optimization. As most of the work in a compiler goes into the so-called optimization passes. This design makes the bulk of the work applicable to all languages. A language can be added to LLVM by writing a frontend, translating source code to LLVM IR. The translation from LLVM IR to machine code for the target architecture is then done by backends, which can also easily be added. Figure 5.2 sketches this architecture.

The optimization passes at the core of LLVM take LLVM IR as input and return it as output. As such, optimization passes for LLVM IR are immediately usable for every language and every platform that is compatible with LLVM. This means implementing my balancing pass for the LLVM toolchain makes it very widely usable. For this reason, and because of the powerful analysis enabled by LLVM's well defined IR language, LLVM was chosen for my thesis.

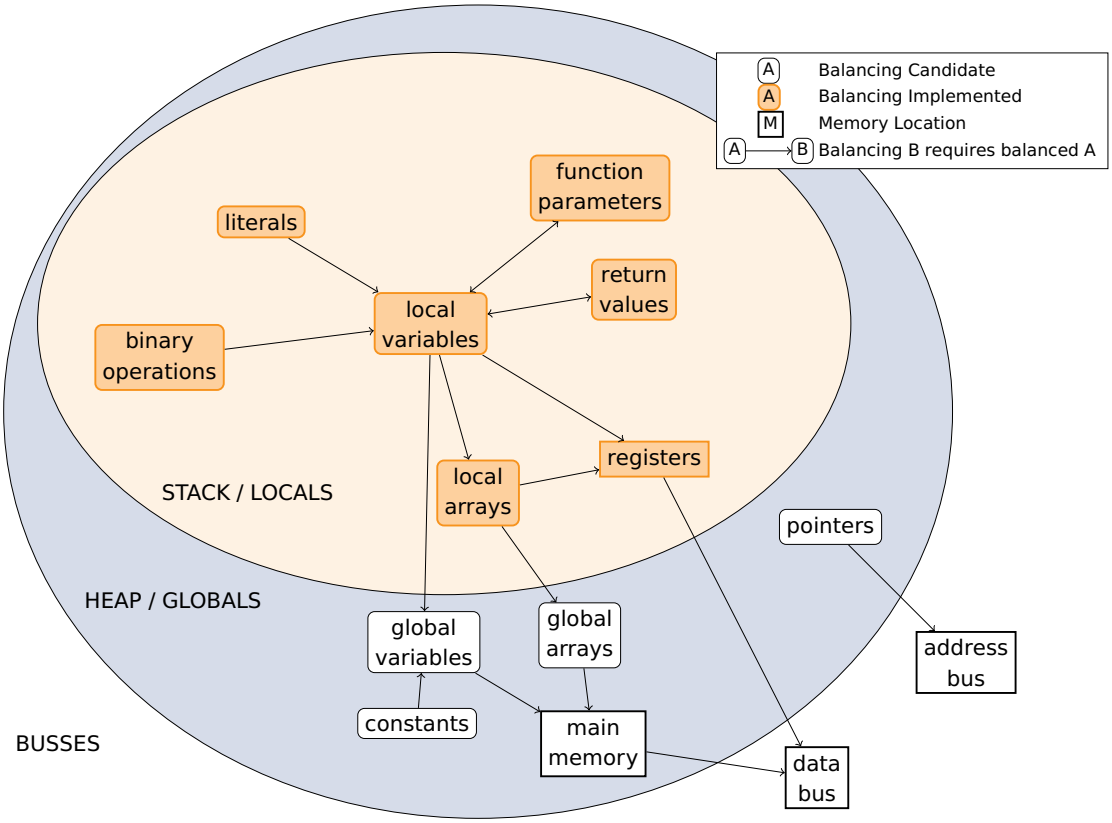


Figure 5.1: The balancing status of my thesis project

5.2 Balancing Pass

The balancing pass stores balanced values 32 bit integers instead of regular 8 bit, and then uses my balanced arithmetic operators instead of the normal operators. While this alone is fairly simple, it causes type mismatches for interactions between balanced and unbalanced memory. These then need to be fixed.

Changing the arithmetic also has some implications on comparison operators. All transformations in the order they are applied are as follows:

1. Change the type of all 8 bit integers (*int8*) to 32 bit integers (*int32*)
2. Balance constant initializers
3. Balance results of load operations if necessary
4. Unbalance values before store if necessary
5. Use balanced arithmetic operations instead of regular operators
6. Fix comparison directions
7. Fix type issues that arise in the instructions that have not been replaced

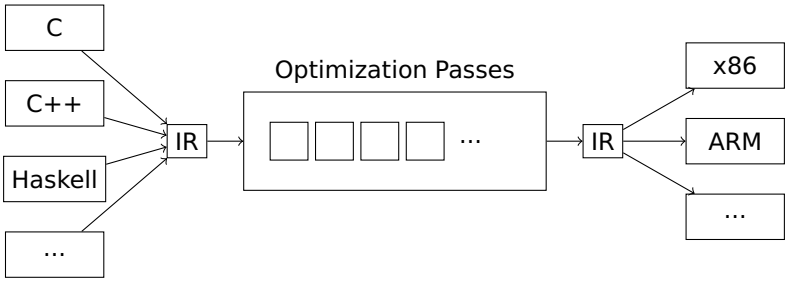


Figure 5.2: The general architecture of the LLVM compiler

The LLVM C++ API provides an iterator over all instructions in a function. My pass uses this to go over all instructions, and transforms the current instruction if necessary. This is usually done by generating a new LLVM IR instruction and replacing references to the old instruction with it.

A special case are function parameters. As these cannot be changed for an existing function, the first step in my balancing pass is to clone the original functions with changed parameter and return types.

Cloning Functions

Cloning functions is done in two parts. First the prototype for the new function is created. During this creation the pass goes through all parameter and changes their types from *int8* to *int32*. The same is done for the return type. This gives me a skeleton for the balanced function, which is inserted into the module, making it accessible in the future. All new functions are given a unique name, specifically the old name prepended with `balanced_`. This naming is important as it allows me to detect already balanced functions before cloning them. It is also required to adjust the target of function calls.

The content of the original function is then copied using a helper in the LLVM API called *CloneFunctionInto*. Without any additional parameters, the copied instructions will still reference function parameters of the original function, will result in broken code in the new function. To avoid this I use a so-called *Value Mapper* to replace the old parameters with the new ones everywhere they are referenced. This change alone would cause type mismatches and generates code that does not compile, but the other steps of my pass fix these problems.

Balancing Allocations

In order to declare and use local variables in LLVM IR the memory for them first has to be allocated using the *alloca* instruction. Even function parameters are not used directly but first copied into memory explicitly allocated for this function. Note that even though the naming is similar to C's *malloc* call, the memory for *alloca* is on the stack in this case.

The *alloca* instruction takes the type to be allocated as parameter, and returns a pointer to that type. This means that for balancing all the pass has to do is replace the *alloca* for *int8* with one for *int32*. Allocations for local arrays work the same way, the pass just needs to extract the dimensions from the old allocation.

Balancing Stores

It can happen that the target code tries to store a balanced variable (*int32*) into an unbalanced pointer (*int8*). In this case the pass unbalances the variable in a temporary before storing it.

While this does cause information leakage and a reduction in robustness, such a case can be avoided fairly easily. As only global memory is unbalanced, this does not happen when the program stores all values on the stack.

Balancing Loads

Balancing loads is a mirror case of balancing stores. When loading from an unbalanced pointer into a balanced variable, the pass first loads into an unbalanced temporary and then balances the value before storing it in the local variable.

Balancing Binary Operations

I implemented the balanced operations described in Section 4.2 in C, each as an individual function. In order to balance binary operations they need to be replaced by calls to these new functions. As all binary operations are represented by the same instruction in the LLVM API, the pass needs to examine the *opcode* of the instruction. Based on that it decides which function call to generate.

For most operations the balanced operation is a direct implementation of the respective steps in Chapter 4. Division, and remainder however are implemented by repeated addition/-subtraction. As an example, Listing 5.1 shows the balanced function for the `sdiv` (signed division) operation in LLVM IR.

Listing 5.1: Balanced sdiv

```
1  int balanced_sdiv(int lhs, int rhs) {
2    uint32_t ret = 0x00ff0000;
3
4    uint8_t negative = 0;
5    if(rhs & 0x00000080){
6      negative = 1;
7      rhs = balanced_negative(rhs);
8    }
9
10
11   while (lhs <= rhs) { //~x <= ~y iff x >= y
12     lhs = balanced_sub(lhs, rhs);
13     ret = balanced_add(ret, 0x00fe0001);
14   }
15
16   if(negative)
17     return balanced_negative(ret);
18   else
19     return ret;
20 }
```

The semantics, especially the handling of negative values are made to be consistent with the semantics of LLVM.

Balancing Pointer Arithmetic

Balanced values cannot be used for array indexing directly. Therefore, whenever a balanced variable is used as index for an array access it is unbalanced before use. All array accesses use the *getelementptr* instruction in LLVM IR, so this is easy to catch.

Balancing Compares

In both of my balancing schemes the inverse occupies the more significant bits than the value itself. This changes the direction of comparison operations, meaning < becomes >, and >= becomes <=.

5.3 Building the Optimization Pass

The compiler pass is built using CMake as that makes loading the required parts of LLVM very easy. Listing 5.2 shows the *CMakeLists.txt* for my balancing pass. The code is based on the template repository provided in [32].

Listing 5.2: CMake configuration for my balancing pass

```
cmake_minimum_required(VERSION 3.13)
project(balancing-pass)

find_package(LLVM REQUIRED CONFIG)
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
link_directories(${LLVM_LIBRARY_DIRS})

add_library(Passes MODULE
    Balance.cpp
)

set(CMAKE_CXX_STANDARD 14)

# LLVM is (typically) built with no C++ RTTI. We need to match that;
# otherwise, we will get linker errors about missing RTTI data.
set_target_properties(PROPERTIES
    COMPILE_FLAGS "-fno-rtti"
)
```

It uses the *find_package* function of CMake, which sets the locations for definitions, header files, and link directories. All these are needed to build my pass. The pass itself is then built as a *MODULE* library, which tells CMake to build a shared library that can be dynamically loaded at runtime by the optimizer. As the pass is loaded by the optimizer, which is built without run-time type information (RTTI), the pass needs to be built without RTTI as well.

This configuration then builds a file called `libPasses.so`. During the compilation of my test code (Section 6.4) I can then load this library as a plugin for the LLVM optimizer.

Chapter 6

Evaluation Methodology

To test the effectiveness of my balancing I compared balanced code with regular code. In order to decrease the turnaround time during development I decided to run the code in an emulator, as opposed to on actual hardware. This also increases the detail at which I can examine the results. Instead of trying to perform a full power analysis attack on actual hardware, I tried to add code to the QEMU emulator that generates simulated power traces during execution, as well as a log of executed instructions, similar to output of the *gdb* debugger. These power traces simulate an attacker that can measure the Hamming weight of every single instruction result without noise.

Unfortunately, performing the evaluation inside QEMU has the disadvantage that it does not accurately simulate the hardware capabilities of ARM, like inline computation of addresses and shifting of operands, and thus adds additional imbalances to the power traces. For this reason I moved the evaluation to gdb, which allows me to evaluate results of every ARM assembly instruction, instead of every instruction in the intermediate representation of QEMU. Even though I used gdb in the end, I included my work on QEMU in this section, as it gives some interesting insights into its internal workings and justifies why it is unsuitable for my evaluation.

6.1 QEMU

QEMU is a generic and open source machine emulator and virtualizer.[3] While it can be used as a full fledged virtualization environment and sandbox, it can also emulate different processor architectures for programs without first emulating an OS. This process, called *bare-metal emulation*, allows me to evaluate the performance of my thesis project on a simulated ARM processor running on my computer.

During the execution of emulated code, so-called guest code, it can also provide a GDB server, allowing for remote debugging of code running inside QEMU. Unfortunately this debugging has to be done in ARM assembly, as all C debugging information has been lost during the build process.

Memory Layout of QEMU Kernels

Even with bare-metal emulation, QEMU still takes its input as a kernel. Due to this, it starts execution at address `0x1000`, as everything before that address is usually reserved for interrupt handling. This requires some additional setup in my build process (see Section 6.4).

Extending QEMU

To perform any evaluation I first needed to add code to QEMU that computes Hamming weight histograms during execution. Doing this proved harder than expected, due to optimizations that happen during execution of guest code.

QEMU does not simply interpret the guest code in a simulated processor. Instead it translates the machine code for the guest platform into machine code for the host platform, and places that "patched" machine code in memory. A second executor thread then runs that code as it becomes available.

This translation backend is called the Tiny Code Generator (TCG), which not only performs the translation but also some optimizations. Instrumenting QEMU for analysis is hard due to the fact that the TCG works through multiple layers of indirection, utilizing both helper functions and preprocessor macros, some of which are defined in different files depending on the host architecture (the specific definition file is chosen while building QEMU).

This multi threaded approach also makes examining values during execution very hard. The executor thread does not know what code it is executing, it only has a pointer (the simulated program counter) to the next instruction or the next basic block. The TCG on the other hand knows which operations are being executed, but it does not know the values of the operands. It also has no way of accessing these values as they might not even be computed yet. So short of either parsing the memory at the simulated program counter or writing a symbolic execution engine for TC (essentially replacing QEMU), this emulation model cannot be used for my thesis.

Luckily, QEMU also offers emulation via the TC Interpreter (TCI). The TCI does not transform TC into host assembly, and instead simulates a guest CPU in software. During this simulation ALU operations are handled by special functions, and their return values are then stored in the simulated registers. The call sites of these functions are where my evaluation code is located, generating power consumption traces and logs of executed instructions.

6.2 Evaluation with GDB

Unfortunately, even TCI does not fully emulate the guest CPU. Instead it simulates a general CPU that executes TCG intermediate code. For my evaluation this means that address calculations, which happen inline on ARM platforms, happen explicitly and add additional unbalanced results to my power traces. Because of this I decided to use gdb for my evaluation.

I wrote a small script that automatically steps through every assembly instruction and afterwards prints the values in all general purpose registers, utilizing the logging feature of gdb to write a transcript of instructions and register states. The register states are then filtered out of the transcript with some regex matching. Finally, I wrote a small python script that detects which register value has changed after the instruction, if any, and generates a power trace from these changes. This allows me even more detailed evaluation than just

the power traces alone. From the transcript I also extracted the corresponding line in the source assembly file for every point in the power trace. With both the value and the location in the assembly file I can examine exactly when and where imbalanced values are stored in registers, as shown in Chapter 6.

6.3 Attacker Model

The traces generated by gdb are perfectly accurate and free of noise. They also have exactly one trace point per instruction, resulting in a very powerful attacker. Any statistical analysis she makes will have results with high confidence, as they are not influenced by noise. The only disadvantage such an attacker has is that she is confined to our power model. So, even though her measurements are exact, she is limited to the sources of leakage I try to mitigate, and does not have access to the power consumption of any other components.

6.4 Test Code

The main test program for my balancing pass was an implementation of AES [10] for embedded devices. This implementation [37] is tested against the test vectors for AES as specified by NIST [11]. I did make some changes to the code, moving as many variables to the stack as possible, in order to maximize the balancing my pass can perform. During development, I also used RC4 [29], SHA-3 [4] and various small programs to test correctness of my pass, but none of them as extensively as AES. AES's small key sizes (128 to 256 bits), and efficient software implementations make it suitable for embedded applications, which is why I chose it as my main test program.

Using gdb for the evaluation has the disadvantage that execution takes longer than on QEMU alone, due to the constant hand-off between gdb and the emulated CPU. On my laptop running AES encryption through QEMU with my gdb script took around 30 minutes, generating 2 million trace points in the process. In order to both reduce the evaluation time, and provide less cluttered plots in Chapter 7, I decided to focus solely on the encryption, which only takes around one sixth of the total execution time for the implementation I used.

Generating different plaintexts

In order to evaluate Hamming weight differences between executions, these executions must use different plaintexts. I did not want to try and get I/O working during execution, so I compiled different executables with different plaintexts. The plaintexts bytes in the program are taken as `#defines`, which are filled with random numbers in the Makefile, using the shell builtin `$RANDOM` and the modulo operator. Generating multiple binaries with different plaintexts was then accomplished by simply executing make multiple times.

Building the Test Code

Because I need to load my plugin and my balanced arithmetic functions during the optimization step, and because of the memory layout in QEMU kernels, the build process is a lot more involved than in normal cross compilation. As discussed in Section 5.1 the LLVM compilation

process can be split into multiple steps. I use this feature multiple times in the build process of my test code. The output of the build process for the AES code is shown in Listing 6.1.

Listing 6.1: Output of the Makefile

```
1  arm-none-eabi-as -ggdb  startup.s -o startup.o
2  clang -target arm-none-eabi -mcpu=arm926ej-s -O0 tinyAES.c -S -emit-llvm -DC0=127 -DC1=90 -
       DC2=197 -DC3=51 -DC4=205 -DC5=78 -DC6=146 -DC7=15 -DC8=146 -DC9=22 -DC10=40 -DC11=156 -
       DC12=42 -DC13=219 -DC14=211 -DC15=147 -o tinyAES.ll
3  clang -target arm-none-eabi -mcpu=arm926ej-s -O0 rtlib.c -S -emit-llvm -DC0=144 -DC1=118 -
       DC2=160 -DC3=148 -DC4=68 -DC5=140 -DC6=15 -DC7=107 -DC8=60 -DC9=91 -DC10=15 -DC11=52 -
       DC12=231 -DC13=252 -DC14=96 -DC15=219 -o rtlib.ll
4  clang -target arm-none-eabi -mcpu=arm926ej-s -O0 program.c -S -emit-llvm -DC0=165 -DC1=14 -
       DC2=20 -DC3=128 -DC4=46 -DC5=7 -DC6=35 -DC7=189 -DC8=242 -DC9=253 -DC10=57 -DC11=19 -
       DC12=180 -DC13=205 -DC14=74 -DC15=103 -o program.ll
5  arm-none-eabi-as -ggdb  u_startup.s -o u_startup.o
6  echo 110,252,242,185,151,32,218,85,233,179,98,188,96,77,30,67 >> plaintexts.txt
7  llvm-link tinyAES.ll rtlib.ll program.ll -S -o linked.ll
8  llvm-link tinyAES.ll program.ll -S -o u_linked.ll
9  opt u_linked.ll -S -o u_optimized.ll
10 opt -load="../../passes/build/libPasses.so" -balance linked.ll -S -o optimized.ll
11 llc -O0 u_optimized.ll -o unbalanced.S
12 llc -O0 optimized.ll -o optimized.S
13 arm-none-eabi-as -ggdb  unbalanced.S -o unbalanced.o
14 arm-none-eabi-as -ggdb  optimized.S -o optimized.o
15 arm-none-eabi-ld -T u_startup.ld u_startup.o unbalanced.o -o unbalanced.elf
16 arm-none-eabi-ld -T startup.ld startup.o optimized.o -o balanced.elf
```

Lines 2, 3, and 4 show the translation of C code into LLVM code, using the Clang[21] C frontend for LLVM. *Program.c* is the file containing the main function, *rtlib.c* contains the balanced binary operations, and *tinyAES.c* contains the AES implementation. The *-S* flag specifies output to be in human readable LLVM IR instead of bytecode, which allows for easier debugging. The specified *-target* platform and CPU (*-mcpu*) are written into the preamble of the LLVM IR, and carried on through the entire toolchain until the compilation into target code on lines 11 and 12. Also visible are the #defines used for the plaintext.

The LLVM IR files are merged in lines 7 and 8 using *llvm-link*. This merger puts the functions from all files in the same module as the target code, and makes them accessible to the compilation pass running on that module. This step is run twice, generating two different *.ll* files. Only one *.ll* file is going to be balanced with my optimizer pass, allowing comparison of balanced and unbalanced code on the same plaintexts.

In line 9 the LLVM optimizer is run without my pass on the unbalanced code, and in line 10 the optimizer is run with my pass. The optimization pass is contained in *libPasses.so*, and called via the *-balance* flag. As discussed in Section 5.1 both the input and output of the optimizer are LLVM IR. Again the *-S* flag is used for human readable output.

In lines 11 and 12 the LLVM IR code is compiled into target code, in this case ARM assembly. Note that the `-O0` flag is issued, which should in theory disable all optimizations in the compiler backend. The specification of the target platform is taken from the preamble of the LLVM IR file, as specified in the frontend call.

The final three steps are handled by the GNU Cross Tools. First the target code is assembled (lines 13 and 14) and then it is linked with a pre-written memory map and a fixed startup assembly file (lines 15 and 16). The memory map is required due to QEMU's memory layout, as discussed in Section 6.1. QEMU starts execution with the program counter set to address

0x1000. Unfortunately, I cannot control the memory layout of the code during and after the compilation process, so I have no guarantee that the *main* function will land at the desired address. For this I use a memory map *startup.ld* (as described in [16]), which causes the code defined in *startup.s* to be at memory address *0x1000*. The content of *startup.ld* is shown in Listing 6.2.

Listing 6.2: Memory map in *startup.ld*

```
1  ENTRY(_Reset)
2  SECTIONS
3  {
4   . = 0x10000;
5   .startup . : { startup.o(.text) }
6   .text : { *(.text) }
7   .data : { *(.data) }
8   .bss : { *(.bss COMMON) }
9   . = ALIGN(8);
10  . = . + 0x1000; /* 4kB of stack memory */
11  stack_top = .;
12 }
```

The code in *startup.s* then fixes the stack pointer and loads the entry function *c_entry* in my test code, as shown in Listing 6.3.

Listing 6.3: Startup assembly code

```
1 .global _Reset
2 _Reset:
3  LDR sp, =stack_top
4  BL balanced_c_entry
5  B .
```

My pass creates new names for the balanced functions it generates. These new names are simply the old ones, with a `balanced_` prefix added to them. Due to this the balanced and unbalanced code require different startup assembly files, *startup.s* and *u_startup.s* respectively. The startup files are assembled in line 1 for the balanced version, and in line 5 for the unbalanced version.

Chapter 7

Results

For my evaluation I compared two executions with different plaintexts, examining the power traces and instruction logs of both executions. A general overview of the results for balanced and unbalanced AES encryption is given in Table 7.1. The number of executed instructions increases by a factor of 15 for the balanced code. This is due to the intermediate steps in balanced operators, as well as the unbalancing required for array accesses, as balanced values cannot be used as indices directly. For the unbalanced code, 90.3% of instruction results have the same Hamming weight for both executions. For my balanced code, this number increases to 98.6%. Unfortunately, due to the increase in overall instructions this still causes an increase in the number of imbalanced values, theoretically increasing the attack surface.

The increase in instructions also causes a significant performance impact, which should be equivalent to the relative increase in the number of instructions. On the other hand, the code size does not increase too much, as all my operators are implemented as functions which are currently not inlined. This behavior could be changed in the future, allowing for a configurable trade-off between code size and function call overhead.

	AES	
	unbalanced	balanced
No. of instructions	22 876	339 168
Relative increase	1	14.888
Balanced operations	20 571	334 521
Unbalanced operations	2211	4647
Balancedness	0.903	0.986
Code size	76 KB	78 KB

Table 7.1: Properties of balanced and unbalanced test code

7.1 Robustness

With the detailed analysis enabled by instrumenting gdb, I located the exact instructions in my program where imbalanced values originate. In this section I will try to explain their cause,

and discuss possible mitigations.

Figure 7.1 shows a scatter plot of the locations of executed instructions over time. The x-coordinate corresponds to the current point in the trace, and the y-coordinate corresponds to the line in the assembly file where the currently executed instruction is located. I highlighted the different functions that comprise AES, as well as the region my balanced operators occupy in the source file. The purpose of most of these functions should be obvious by their name, the only exception being `xtime`. It is used to perform multiplication in $\mathbb{GF}_{28}$ more efficiently. The block of instructions not in any highlighted around line 700 is code I added to copy the S-box to the stack. A more detailed view of the regions belonging to my balanced operators is shown in Figure 7.2.

In Figure 7.3 I show the same execution plot, this time highlighting the points where the difference in Hamming weight between both executions is non-zero. The key expansion has no differences for both executions, as it is independent of the plaintext and thus the same for both executions. This is also true for copying the S-box to the stack. Imbalances in my operators are expected, but not as many as occur. Additionally, imbalances in the `ShiftRows` and `MixColumns` operations should not exist, as all these functions do is move entire 32 bit words. There also should be no imbalanced values in the `AddRoundKey` and `xtime` functions, because while they do require operations with imbalanced intermediates, the results of all these operations are perfectly balanced.

The reason for this lies in the code the ARM backend generates. Figure 7.4 shows a plot of the points with different Hamming weights, with right shift instructions highlighted. The only place these should happen in my code is in `balanced_lshr`, my balanced right shift operator. They also happen in `balanced_2_1`. The instructions causing this could be inlined into a rotation in ARM, which would avoid the imbalanced intermediate entirely. However, this is not generated correctly for my code.

The rest of the shifts is caused by a pattern generated by the backend for store operations. Instead of storing the entire word, it is right shifted by multiples of 8 bits, and the least significant byte of the result is stored with an offset instead. This pattern might be required due to misaligned memory locations on the stack, and unfortunately I did not have the time to examine why this happens. As this shift and store pattern is also experienced by others [34], “fixing” this might require making changes to the LLVM ARM backend, which would greatly exceed the scope of my thesis.

When these shifts are filtered, the points in Figure 7.5 remain. In this figure I highlighted move and load instructions, as these are not the root cause of imbalanced values, but only symptoms of them. Also removing these leaves me with Figure 7.6, where the functions containing the cause of these imbalances are highlighted. Imbalances in `balanced_2_1` could be avoided if the rotate instruction was generated correctly.

The imbalances in the `unbalanced_int` function cannot be avoided without making changes to the way memory is accessed. As the index used in the `SubBytes` function in AES is directly linked to the key (especially in the first round), this theoretically leaks the full key to an attacker. However, for a real world attacker with noisy measurements this still increases the attack difficulty, as she can no longer attack the result of the S-box lookup. The permutation caused by the S-box lookup means values with similar Hamming weights before the lookup usually have very different Hamming weights after the lookup, which makes performing any form of statistical power analysis a lot easier. This is not possible for my balanced code, as then the result of the lookup has a constant Hamming weight.

Imbalances in `balanced_and` are due to DeMorgan’s law, but it might be possible that a

different balancing scheme or a different implementation of this operator can avoid these imbalances. The same is true for `balanced_mul`. I am unsure of the reason for imbalanced values in `balanced_lshr`, and did not have the time to dive into why exactly these imbalances happen. The remaining imbalances are in my entry function, and are the initial setting of the plaintext. These are not related to the key and thus no security risk.

After filtering imbalances not in my operators, and those created by move and load instructions, the number of remaining imbalanced results is 1450. This is less than the number of imbalanced intermediates in the unbalanced code. However, for the unbalanced code I did not do this filtering due to time constraints, so making a hard robustness claim here is difficult.

7.2 Performance

The current balancing pass increases the number of instructions by a factor of 14.888. This increase is caused by the more complicated balanced operators and overhead for additional function calls. These additional calls are because currently my operators are implemented as functions, in order to minimize the increase in code size. More inlining would be possible, which would reduce the number of instructions, but increase the code size.

Additionally, there might be faster variants of implementing the balanced operators. One could even consider accepting some more information leakage to reduce the number of intermediate operations required. Taking hardware specifics, such as the ARM barrel shifter into account could also help reduce the performance impact. Hardware specific optimizations would however reduce the generality of my balancing pass. The focus of this thesis was to test wether such a balancing pass was *possible*, so performance optimizations are left for future work.

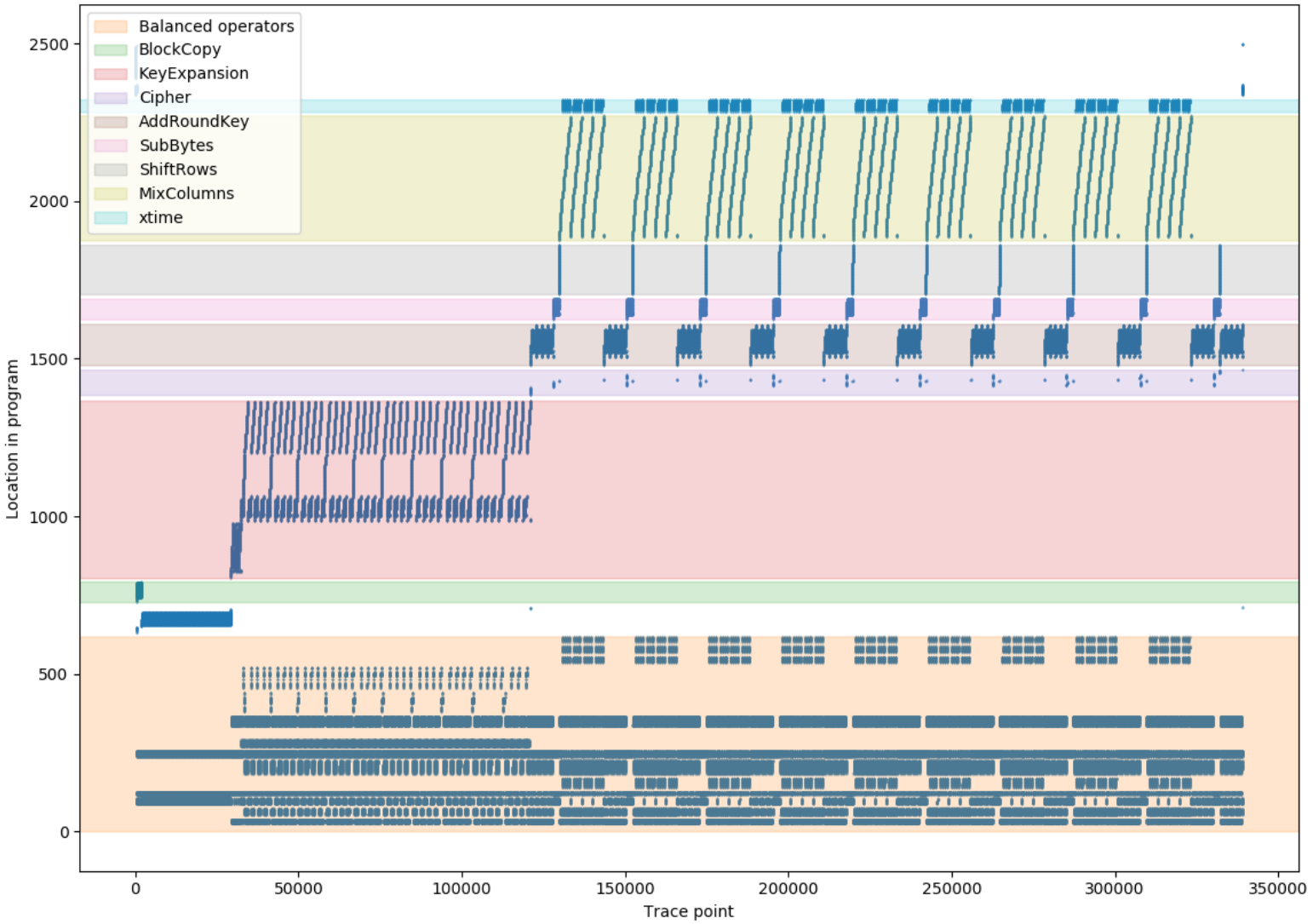


Figure 7.1: Program regions of AES functions

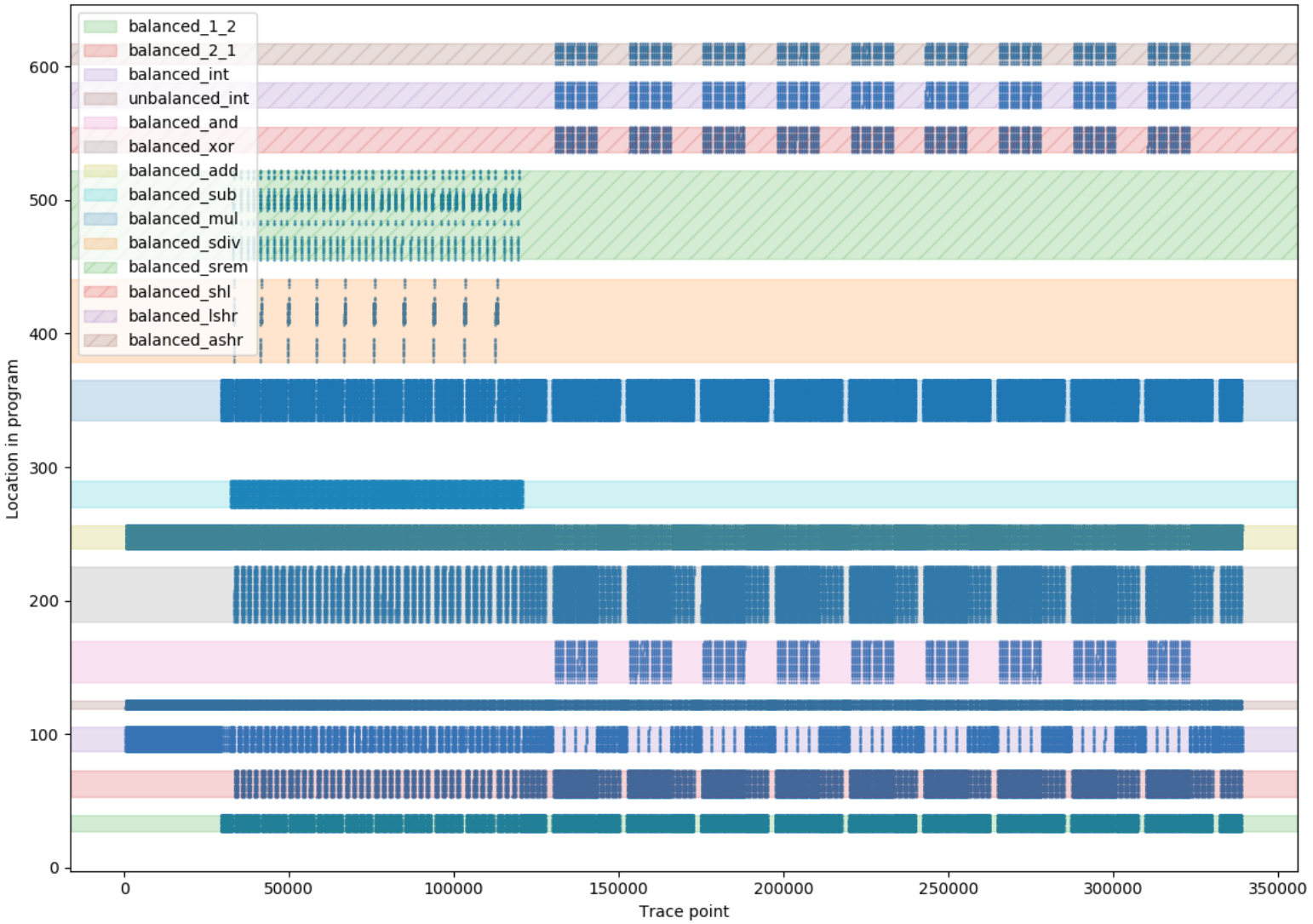


Figure 7.2: Program regions of balanced operators

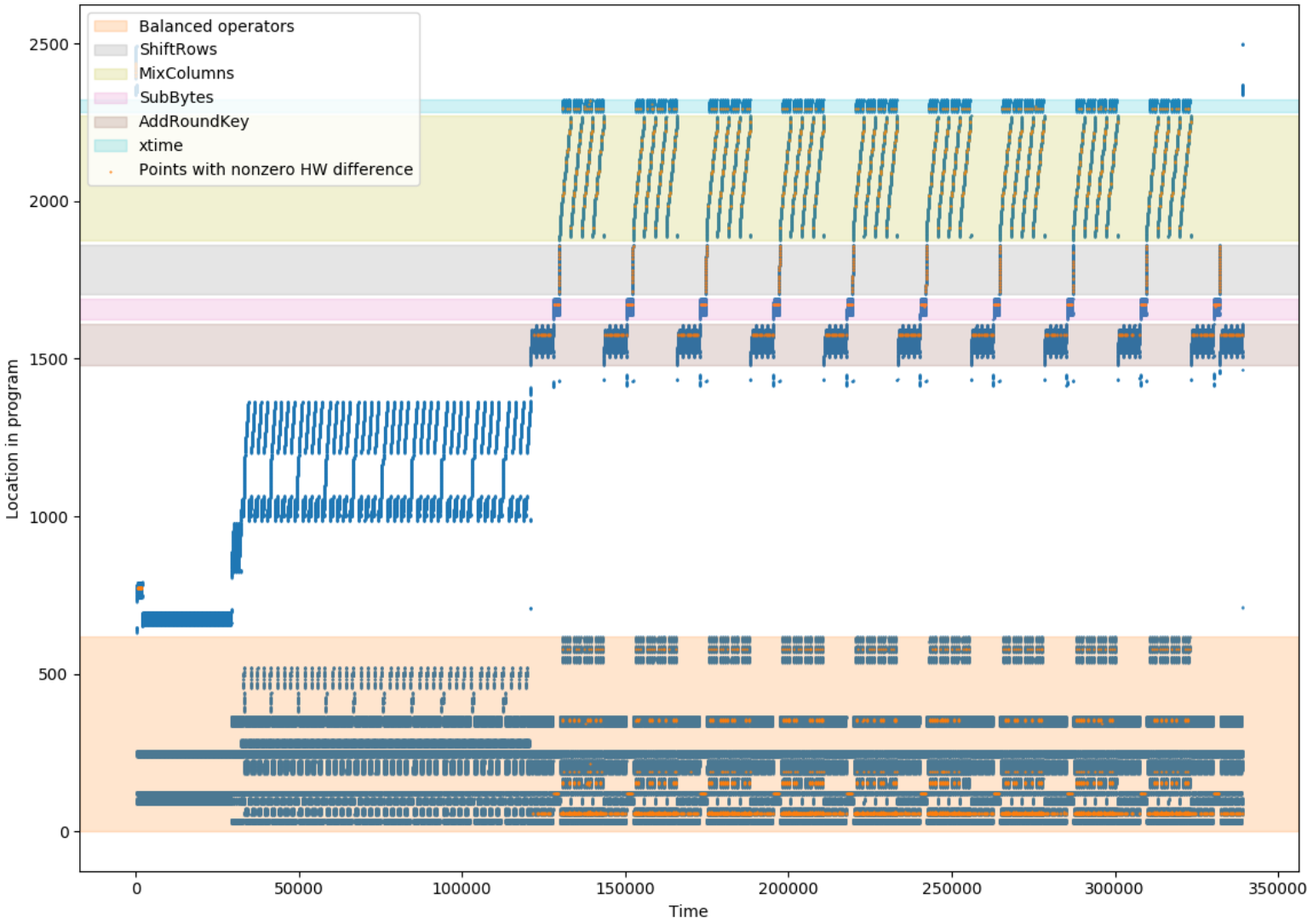


Figure 7.3: Hamming weight differences in power trace

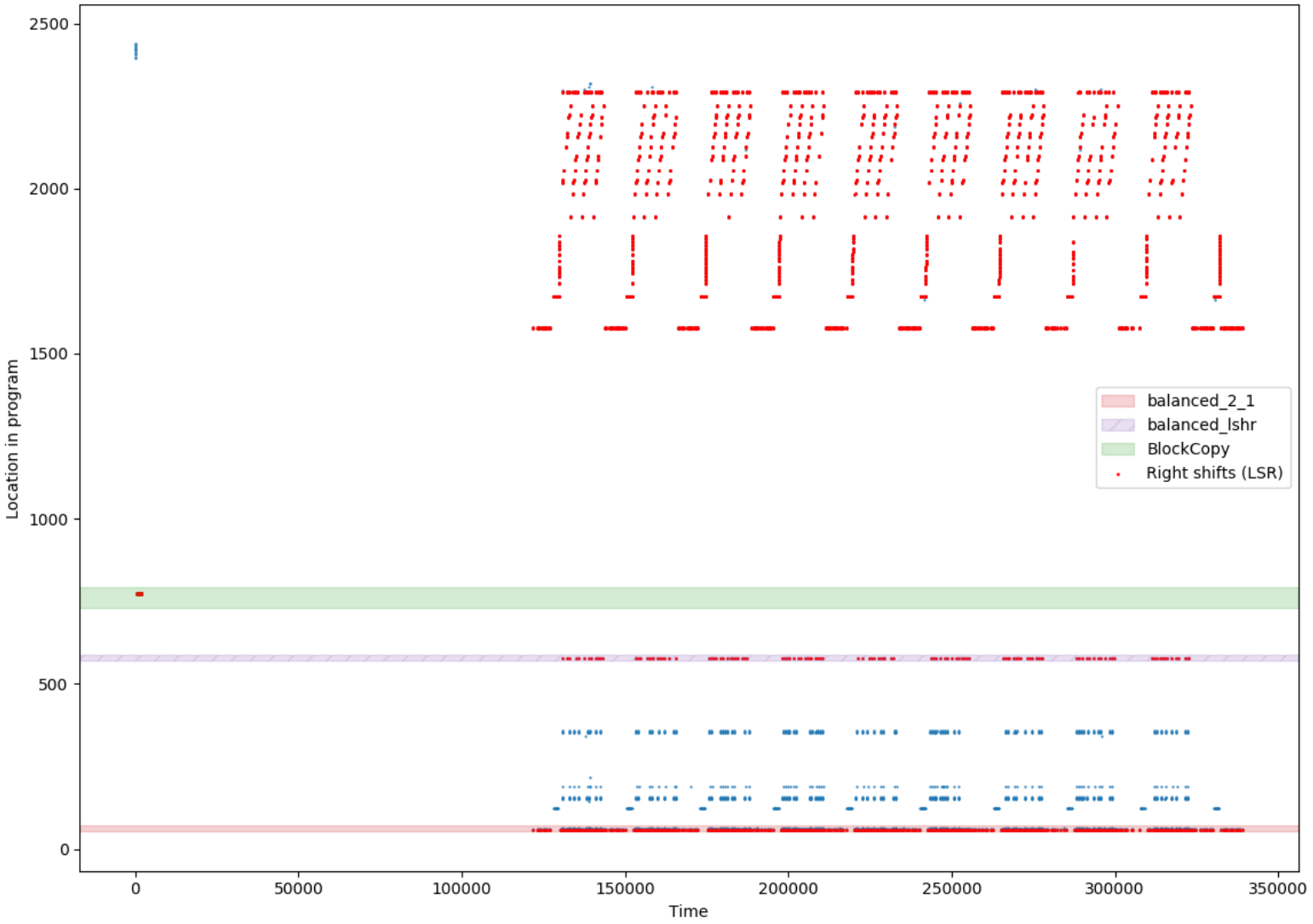


Figure 7.4: Hamming weight differences due to right shifts

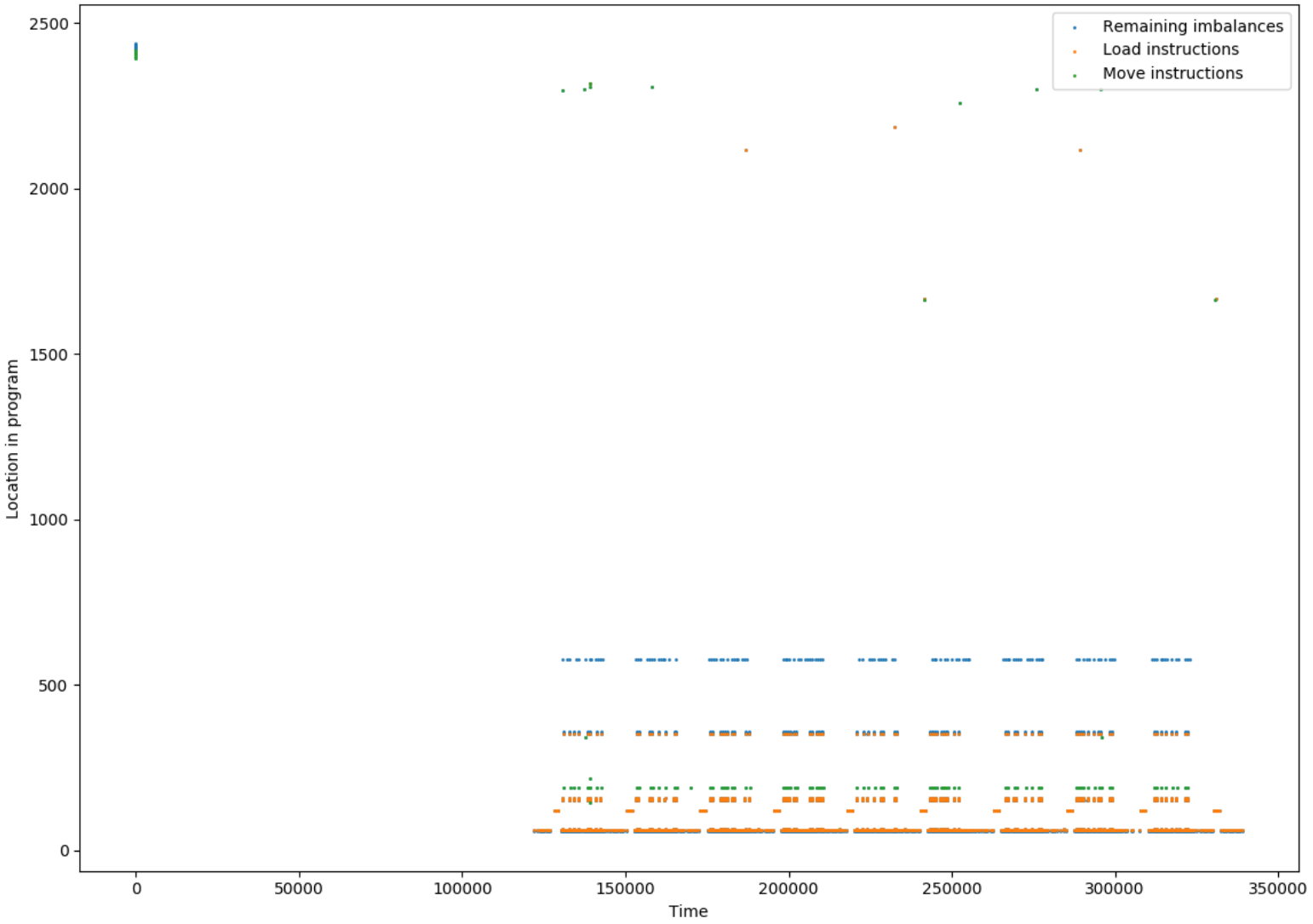


Figure 7.5: Hamming weight differences due to load and move instructions

Figure 7.6: Sources of imbalanced intermediates

Chapter 8

Conclusion

In my thesis I propose and evaluate a compiler generated approach to balancing values for increased robustness against power analysis, similar to dual-rail logic, in software. The robustness increase is achieved by not using the entire available word size, and instead storing the inverse of data in order to balance the Hamming weight. Using such balanced values requires an entirely new arithmetic, which is the first contribution of this thesis. I then provide a compiler pass for LLVM that automatically transforms C code using at most 8 bit integers into my balanced arithmetic. The resulting code is then evaluated by instrumenting gdb on a CPU emulated using QEMU. Gdb output is examined to find imbalances in the final code, and a discussion of reasons and possible mitigations for these imbalances is offered.

8.1 Summary of Results

The arithmetic created for my balancing scheme is correct for all possible 8 bit operands. As it provides balanced replacements for all integer operations required for a modern RISC architecture, it is sound and complete. However, it does not provide perfect balancing for all operations. While my chosen balancing scheme enables the reuse of existing ALU operations, it also forces some imbalanced intermediates. In my scheme the inverse of a value is used for balancing, which can require different operations than the actual result, due to DeMorgan's law. As it is impossible in current ALUs to execute different instructions on parts of a register at the same time, there *must* be imbalanced intermediate values with this scheme.

While I cannot make hard claims on the reduced information leakage of my individual operators, the distribution of Hamming weights of intermediate values (Figure 4.2) suggests reduced leakage compared to regular operators. The number of imbalanced values also increases much less than the total number of instructions. Unfortunately, due an increase in instructions in general, the number of imbalanced values is still larger than in unbalanced code. However, the imbalanced values increase by a factor of 2, while the instructions in general increase by a number of 15 for AES encryption. This means that the signal-to-noise-ratio for an attacker is reduced from 9.7% to 1.4%.

In theory the entire secret key is still leaked, as the S-box lookup in AES requires using an intermediate value (which in the first round is the plaintext XOR the key) as array index. This cannot be balanced without making changes to the way memory is accessed, which would explode the scope of my thesis. In practice, however, the difficulty of an attack is still

increased, as the only possible target is now the unbalancing, instead of the S-box lookup. Attacking the S-box lookup has the advantage that it “shuffles” the values and therefore their Hamming weights, making statistical analysis of the Hamming weights easier. This is not possible for code balanced with my pass, as then the result of the lookup is already balanced.

Many imbalances that currently remain are due to behavior of the LLVM ARM backend. It frequently uses a pattern of right shifts and byte stores, which completely destroy my balancing. Fixing this could require making changes to the backend, but that also exceeds the scope of my thesis. After filtering these, and after filtering imbalances due to load and move instructions (which only propagate imbalances and do not cause them), the number of imbalanced instruction results is 1450, out of a total of 339168 instructions.

My approach has been tested on C code written for an 8 bit architecture, but the general nature of LLVM should make it applicable to any source language whose frontend generates 8 bit integers. As it is possible to convert *any* code into code using only 8 bit integers, my general approach is applicable to *any* code written in an LLVM compatible language.

I also created an extension to QEMU, generating simulated power traces and execution logs during emulation. While these logs are not suitable for my ARM specific evaluation, they can provide insights into the execution of more general architectures. My evaluation method using gdb, correlating changed values to the instructions causing them, can also prove beneficial, especially for tracking down information leakage via side-channels.

8.2 Limitations

The performance impact of my balanced arithmetic is very large, increasing the number of instructions by a factor of 14.888. This increase is in part due to a higher complexity of binary operations, and in part due to the current way they are implemented. To keep the code size small the operations are currently implemented as functions, which requires additional overhead of saving and restoring registers before and after each operator. Inlining all or a part of these operators offers a trade-off between code size and instruction count, possibly reducing the performance impact of my balancing. With current ALUs and a RISC architecture, an increase in the number of operations by 5 seems to be a lower bound, as that is the lowest number of intermediate operations in my balanced arithmetic.

It is also important to note that this is for 8 bit word sizes on a 32 bit capable platform. With this reduced word size it would require algorithms for handling large numbers (along the lines of GnuMP[15]) to reach the full 32 bit value range again. Using such a large number arithmetic on top of my balanced arithmetic would additionally increase the number of instructions by at least 4 (to reach 32 bit words again), with the real performance impact probably being much higher due to additional introduced overhead.

The balancing is currently also incomplete, as only stack values are balanced. Balancing global values (and thus all value types), would completely free the programmer from any limitations on their program. Currently there is also no formal validation of my method. To make any real claims on increased robustness an actual attack would have to be performed. Attacking my balanced code via the simulated traces generated by gdb would very likely succeed, as these traces have no noise. A real attack on actual hardware would be required to evaluate the performance of my balancing and the thus reduced signal-to-noise-ratio of the power consumption.

8.3 Future Work

The most obvious future work for my thesis would be extending the balancing to global values. This would reduce the constraints on the programmer while still allowing her to benefit from increased robustness. Performing an actual power analysis attack on hardware running code balanced by my pass would also be a valuable contribution, as it allows evaluation of the actual robustness increase provided by my thesis.

Alternatively, a nice feature would be the ability to mark certain variables for balancing, for example via special types or annotations. This marking would then be transmitted to LLVM IR by a plugin for the frontend for the current language (for C this would be Clang). The balancing pass could then create a graph of all variables influencing the balancing target and balance all of them. This would reduce the performance impact, while still providing increased robustness where needed.

A different, simpler opportunity for future work would be implementing balancing in the type system itself, creating a set of balanced classes and overloading their operators to create a balanced arithmetic. While this would make the balancing only applicable to a single source language, it would avoid the need to alter the compiler itself. Such a method would require ensuring the compiler does not destroy the balancing during optimization, which makes the evaluation of such a project potentially very complex.

It is currently also not clear if the balanced operations are optimal. Future work could find either a faster arithmetic (with possible trade-offs for robustness), or an optimality proof of the current arithmetic. Reducing the performance impact could make this approach more generally applicable, and give it some real use cases.

In the realm of side channel defenses there are a number of possibilities for future work. My thesis has evaluated the effects of a generally applicable *hardware* defense automatically applied in software during compilation. Following a similar approach, one could utilize static analysis in the compiler to generalize software defensive measures. Masking could be applied automatically, with new masks (and lookup tables) being computed during compile-time, if so desired. This has already been done [27], albeit using a custom compiler and not as a plugin to an existing and widely used compiler like LLVM.

Defenses based on secret sharing schemes [13] could also be automatically applied, based on their invariant operations and operations happening in the program. By searching a predefined list of schemes, complete with their invariant operations, the compiler could offer applying such a scheme for increased robustness against power analysis.

Side channel defenses are only a limited subset of the possibilities of compiler enabled security in general. The power that well defined intermediate representations in modern compilers (especially LLVM IR) afford us can enable much more sophisticated security analysis than was previously possible. Using this we should be able to automatically catch and fix many trivial security issues, and provide high level feedback where no automatic fix is possible. Where code analysis plugins for IDEs now point out possible simplifications in logical predicates, they could point out potential security risks in the future.

Bibliography

[1] Mehdi-Laurent Akkar and Christophe Giraud. “An implementation of DES and AES, secure against some attacks”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2001, pp. 309–318.

[2] Karthik Baddam and Mark Zwolinski. “Path switching: a technique to tolerate dual rail routing imbalances”. In: *Design Automation for Embedded Systems* 12.3 (2008), pp. 207–220.

[3] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[4] Guido Bertoni et al. “Keccak”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2013, pp. 313–314.

[5] Rainer Böhme. *Information Security II lecture notes*. 2017.

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

[7] Suresh Chari et al. “Towards sound approaches to counteract power-analysis attacks”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 398–412.

[8] Jean-Sébastien Coron and Louis Goubin. “On boolean and arithmetic masking against differential power analysis”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 231–237.

[9] Jean-Sébastien Coron, Paul Kocher, and David Naccache. “Statistics and secret leakage”. In: *International Conference on Financial Cryptography*. Springer. 2000, pp. 157–173.

[10] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[11] Morris Dworkin. *Recommendation for block cipher modes of operation. Methods and techniques*. Tech. rep. National Inst. of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[12] Jovan D Golić and Christophe Tymen. “Multiplicative masking and power analysis of AES”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2002, pp. 198–212.

[13] Louis Goubin and Ange Martinelli. “Protecting AES with Shamir’s secret sharing scheme”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2011, pp. 79–94.

[14] Louis Goubin and Jacques Patarin. “DES and differential power analysis the “Duplication” method”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 1999, pp. 158–172.

[15] Torbjörn Granlund. “GnuMP”. In: *The GNU Multiple Precision Arithmetic Library* 2.2 (1996).

[16] *Hello world for bare metal ARM using QEMU*. url: https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/ (visited on 06/12/2019).

[17] Pascal Junod et al. “Obfuscator-LLVM - software protection for the masses”. In: *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*. IEEE. 2015, pp. 3–9.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential power analysis”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[19] Paul Kocher, Joshua Jaffe, Benjamin Jun, et al. *Introduction to differential power analysis and related attacks*. 1998.

[20] Paul Kocher et al. “Introduction to differential power analysis”. In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27.

[21] Chris Lattner. “LLVM and Clang: Next generation compiler technology”. In: *The BSD conference*. Vol. 5. 2008.

[22] Chris Lattner et al. “The LLVM compiler infrastructure”. In: *URL http://llvm. org* (2010).

[23] Chris Lattner and Vikram Adve. *LLVM language reference manual*. 2006.

[24] Yi-Hong Lyu et al. “DBILL: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend”. In: *ACM Sigplan Notices*. Vol. 49. 7. ACM. 2014, pp. 141–152.

[25] Thomas S Messerges. “Securing the AES finalists against power analysis attacks”. In: *International Workshop on Fast Software Encryption*. Springer. 2000, pp. 150–164.

[26] Simon Moore et al. “Improving smart card security using self-timed circuits”. In: *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*. IEEE. 2002, pp. 211–218.

[27] Andrew Moss et al. “Compiler assisted masking”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2012, pp. 58–75.

[28] Alin Razafindraibe, Michel Robert, and Philippe Maurine. “Formal evaluation of the robustness of dual-rail logic against DPA attacks”. In: *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer. 2006, pp. 634–644.

[29] *RC4*. url: https://en.wikipedia.org/wiki/RC4 (visited on 07/21/2017).

[30] Matthieu Rivain and Emmanuel Prouff. “Provably secure higher-order masking of AES”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2010, pp. 413–427.

[31] Eyal Ronen et al. “IoT goes nuclear: Creating a ZigBee chain reaction”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 195–212.

[32] Adrian Sampson. *LLVM for Grad Students*. 2015. url: http://www.cs.cornell.edu/~asampson/blog/llvm.html (visited on 06/12/2019).

[33] Hendra Saputra et al. “Masking the energy behavior of DES encryption”. In: *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society. 2003, p. 10084.

[34] Laurent Simon, David Chisnall, and Ross Anderson. “What you get is what you C: Controlling side effects in mainstream C compilers”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 1–15.

[35] Rafael Soares et al. “Evaluating the robustness of secure triple track logic through prototyping”. In: *SBCCI'08: Symposium on Integrated Circuits and Systems Design*. ACM. 2008, pp. 193–198.

[36] Danil Sokolov et al. “Design and analysis of dual-rail circuits for security applications”. In: *IEEE Transactions on Computers* 54.4 (2005), pp. 449–460.

[37] *Tiny AES in C*. url: `https://github.com/kokke/tiny-AES-c` (visited on 07/29/2019).