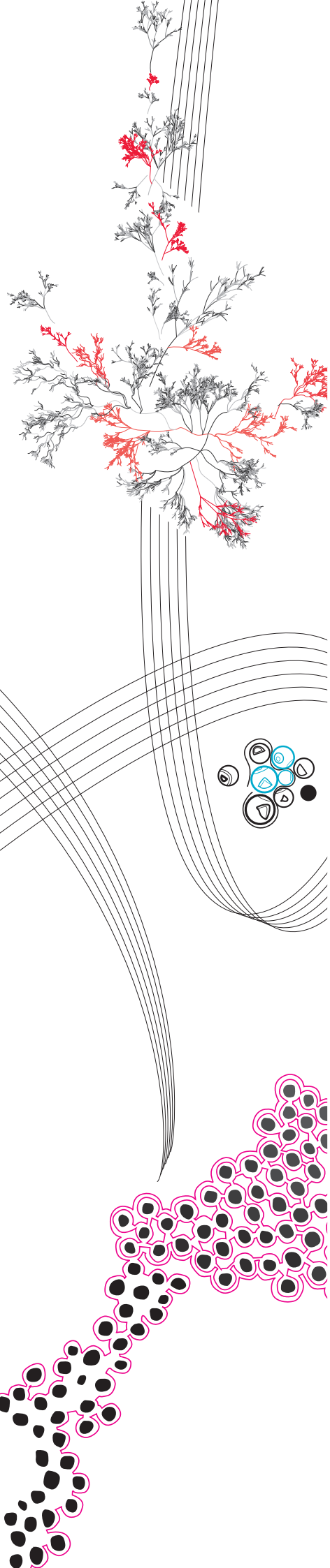

MASTER THESIS

Cycle-Accurate Timing Channel Analysis of Binary Code

Roeland S. Krak

Faculty of Electrical Engineering, Mathematics and Computer Science
Services, Cybersecurity and Safety Research Group

May 2017

UNIVERSITY OF TWENTE.

UNIVERSITY OF TWENTE.

Cycle-Accurate Timing Channel Analysis of Binary Code

by
Roeland Krak

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of Science.

May 2017

Graduation committee:
prof.dr. M. Huisman
dr. A. Peter
E. Sanfelix Gonzalez, M.Sc.

Abstract

Software execution time may leak information about secrets processed by that software through vulnerabilities known as timing channels. Previous automated analysis techniques rely on a highly abstract model of instruction execution time, which limits the accuracy of these tools. We constructed a detailed model of instruction execution time for the ARM Cortex-A7; and present SMArTCAT, a tool which relies on this timing model to identify timing channels in binary code. SMArTCAT is the first automated tool which can identify timing channels caused by instructions with parameter-dependent execution time, as well as traditional timing channels. This allows to more accurately test critical pieces of software for the presence of timing channels, which we demonstrate in several case studies.

Contents

Part I

Foundation

Chapter 1

Introduction

Machines frequently handle secret and privacy sensitive data, such as secret keys used in cryptographic software, for which it is catastrophic if an adversary obtains them. Unfortunately, many systems leak secret information through nonfunctional behavior such as execution time. For example, a program can conditionally execute either of two code paths. If the paths have different execution time, and the value of a secret parameter determines which path is executed, there is a relation between the secret and the execution time. An attacker who measures the execution time can deduce which path was executed, and thus learns information about the secret parameter.

These vulnerabilities are known as timing channels. Practice shows that exploitable timing channels are not uncommon in cryptographic software [12, 15, 27]. Recently, attention has also been drawn to timing channels which leak privacy sensitive data from web browsers [9, 29].

Previous work [8, 10, 33] describes methods to identify timing channels in code. Timing channel identification is based on two parts: a model of execution time, and an analysis technique to identify relations between secrets and execution time. The timing model provides a notion of execution time at the level of instructions. It can be applied cumulatively on multiple instructions to model time for a larger piece of code, i.e., a function or a program. The analysis technique is applied on the program code and the model of its execution time to identify timing channels.

The approaches described previously [8, 10, 33] rely on on a highly abstract notion of instruction execution time, known as the program counter model. However, it is unrealistic for general purpose processors, because it does not take into account timing differences between different instructions and different instruction operands. These features have been proven to lead to practical timing channels, as demonstrated by Andrysco et al. [9]. These approaches are insufficient to identify timing channels that are caused by these features.

Until now, there has been no research conducted which looks at the achievability, costs and benefits of timing channel analysis based on a timing model more detailed than the program counter model. This leads us to the research

question we address in this work:

> **What is the impact of detailed timing models on timing channel analysis?**

In this work we describe how we construct a novel timing model for the ARM Cortex-A7, a processor found in many embedded devices. In doing so, we contribute to the general understanding of this processor's timing behavior.

Furthermore, we introduce SMArTCAT, a tool for automated identification of timing channels. It leverages symbolic execution and our timing model for the Cortex-A7 to identify whether secret program parameters cause differences in instruction execution time. This is the first tool which can analyze programs for timing channels at this level of detail. Because of the level of detail in the timing model, it can identify a whole new class of timing channels which previously could not be identified by automated analysis.

This work demonstrates that SMArTCAT can identify timing channels in ubiquitous cryptographic software, and that it can validate that other cryptographic software does not contain any timing channels according to our attacker model and timing model.

The rest of this work is organized as follows. First, we instroduce the background to our work: chapter 2 discusses timing channels, chapter 3 discusses symbolic execution, and chapter 4 describes the ARM Cortex-A7. We then continue to the main body of our work. Chapter 5 describes how we experimentally constructed a timing model for the Cortex-A7. Chapter 6 discusses our binary analysis technique, and chapter 7 discusses how the technique was implemented in SMArTCAT. Chapter 8 describes how SMArTCAT has been used to analyse several cryptographic libraries. We then compare our work to several other automated timing channel analysis techniques in chapter 9. In chapter 10 we conclude and reflect on this work. Finally, in chapter 11, we look ahead at possible directions to advance this field of research and improve on our approach.

Chapter 2

Timing Channels

Data security in software is traditionally defined using the notions of data flow, and secret and public variables. Traditional data leaks comprise data flows from secrets to public program outputs. In contrast, side-channels comprise data leaks through *nonfunctional* program behavior, i.e., side-effects of program execution. These vulnerabilities can be exploited by attackers who perform measurements of the side-effects during execution. In this work we focus on timing channels: side-channels which exist due to secret variables that influence program execution time.

Consider Example 1. Clearly, execution time of this function depends on the secret variable, as it determines the number of iterations that the loop is executed. Thus, if an attacker can measure execution time, he can infer information about the secret variable. The example has an obvious linear relation between secret and side-channel. As we will later describe, there are also timing channels which contain much more subtle relations, and are harder to identify. On a side node, the examples we use to introduce timing channels are highly superficial, to fully focus on the different types of side channel causes.

```
function (int secret) {
    while (--secret > 0) {
        ...
    }
}
```

Example 1: Linear timing channel

Data flow analysis builds on the idea that programs handle parameters of different security levels. For simplicity we consider only two security levels: public and secret. The intuition behind data flow security is that public output of a program should only be based on public parameters, and should not be influenced by secret parameters as this leaks secret information. This notion

is known as non-interference [20]: the values of secret parameters should not influence values of public outputs.

Traditional non-interference was formally defined for deterministic sequential systems, in which it can be proved that secret variables never influence public output. To cope with concurrent programs and side-channels, Zdancewic and Myers [44] defined the notion of observational determinism, a generalization of non-interference. Originally focused on internal timing channels, this notion can again be generalized to include external timing channels as well. This concept was formally refined by Huisman et al. [23], and forms the basis of our security notion.

We consider an attacker who is able to read timed program traces, i.e., traces of execution time for each executed instruction. Observational determinism holds if executions that differ only in secret input are indistinguishable by their traces. In this work we focus solely on data leaks by execution time, and leave other types of data leakage for other work. Observational determinism is concerned with abstract observations, i.e., points in time in which the program performs an action observable to the attacker. This concept is similar to that applied by Balliu et al. [10]. However, they considered only observations of timed memory writes. On the other hand, by keeping the concept of observations abstract, we can identify timing channels which are based on different kinds of observations, e.g., standard program output, network traffic, file locks and manipulations, memory writes, or process scheduling behavior.

2.1 Timing Channel Types

There are many different patterns in program code which cause non-constant execution time. However, programmers are often not concerned with non-functional program behavior, and it is not always obvious whether variables should remain secret or not. This makes timing channels a very prominent class of vulnerabilities.

Example 1 shows a clear linear relationship between secret value and execution time. However, many timing channels only display minor and nonlinear differences in execution time. In practice, minor differences can be aggravated when they are located in tight loops or can otherwise be executed multiple times. If an attacker is able to influence how frequent a minor timing difference occurs, e.g., by choosing specific public inputs, it can be trivial to take sensible measurements from a timing channel. Consider Example 2, it shows a small timing difference depending on the secret value, i.e., `...` is only called if `secret` is true. However, if the attacker can choose `public` to be large, then, depending on `secret`, `...` will be called either many times, or not at all. Thus, the timing difference between both possibilities is large and it is easy to differentiate between them.

Many situations allow attackers to measure execution time of victim processes remotely, e.g., by looking at network communication, or by executing malicious code on the same machine. This can make it possible to exploit

```
function (int public, boolean secret) {
    while (--public > 0) {
        if (secret) {
            ...
        }
    }
}
```

Example 2: Timing channel within loop

timing channels in situations in which it is not possible to exploit other side channels, for which physical access to the victim machine is generally required. Imagine that Example 2 is part of a network protocol implementation. Observations could be made of packets sent before and after this piece of code, which allow an attacker to measure the time between those packets and thus deduce the secret.

So far we have only shown examples with timing channels based on control flow, i.e., the value of a secret determines which code is executed. However, execution time can depend on secret variables in multiple ways. Broadly speaking, we categorize timing channels as either control flow–based, or architecture-based. In the following subsections we discuss what we mean by both.

2.1.1 Control flow–based timing channels

Programs can have multiple code paths, i.e., instruction sequences which can be executed. Branch instructions determine which sequences are executed. Control flow describes how variables determine which paths are executed, because the branch conditions depend on those variables. Control flow–based timing channels are a result of control flow dependencies on secret variables. A timing channel is caused by a difference in execution time for the different possible paths. An attacker who can use the execution time to deduce which path was taken, knows whether the condition of the branch instruction was satisfied or not. This allows him to deduce information about the value of the secret variable.

It is challenging to write program-code for which execution time does not depend on the executed code path. This is caused by the underlying challenge to determine execution time for any path at all. Naively, execution time of a path equals the sum of the execution time of all individual instructions in that path. However, it is impossible to determine a general cost model for each instruction, as this differs between processors. It is thus impossible to accurately predict execution time without a specific processor in mind.

Furthermore, instruction execution time can depend on multiple factors such as other executed instructions and the state of the processor. For example, branch prediction, out-of-order execution, and caching all depend on the state

of the processor and can all influence execution time. These techniques are intended to optimize processor performance. However, manufacturers make little to no guarantee about their effects on processing speed of individual instructions, which means instructions in one code path may have different execution time than the same instructions in another path.

Control flow–based timing channels thus give constraints on values of secret variables, depending on which code path was executed. If there are multiple conditional branches which depend on secret variables, or if an attacker is able to influence the learned constraints through public program parameters, this information leak may disclose complete secrets to the attacker.

2.1.2 Architecture-based timing channels

Besides control flow–based timing channels, there is a type of timing channels we refer to as architecture-based timing channels. These timing channels are a direct result of the interplay between program code and low-level machine behavior. If this interplay influences instructions' execution time depending on secret values, the timing difference can leak information about those secrets.

One example of non-constant execution time instructions are floating point operations. For most parameters, modern processors can execute floating point arithmetic in constant time. However, if floating point values are really small – known as denormalized numbers –, instructions can take significantly longer to process them. Consider Example 3: a secret floating point variable gets modified through simple arithmetic with a public variable. If an attacker can choose `public`, he can choose it such that the result of the operation is denormalized only if `secret` is smaller than some arbitrary edge value. As processing the denormalized number will take longer than processing a normalized number, there is a timing channel from which an attacker can learn whether `secret` is larger or smaller than the chosen edge value. If `public` can be chosen freely, a binary search will quickly determine the exact value of `secret`. Andrysco et al. [9] demonstrated that such vulnerabilities exist in practice.

```
function (float public, float secret) {
    return public * secret;
}
```

Example 3: Floating point–based timing channel

Coppens et al. [17] discussed other instructions with non-constant execution time on modern x86 processors. They identified that integer division, and division and square root instructions on normalized floating points also exhibit parameter-dependent timing behavior. They also mention that rotation and shift instructions and multiplication have displayed this behavior on older x86 implementations. They did not identify the behavior of denormalized floating point instructions, but successfully demonstrated prevalence of variable-latency

instructions.

Similarly, memory instructions generally exhibit non-constant execution time, due to interference of the cache when data is loaded from memory or written to it. The cache sits between memory and the processor and temporarily stores data from memory for quick reference. The cache is significantly faster than memory, which can create a timing channel which leaks information about the accessed memory locations.

A typical programming structure seen in cryptographic implementations [12, 35, 37], is that of arrays which are accessed by indices that depend on secrets. Thus, the secrets determine which array elements are loaded into the processor's registers from memory. However, this can leak information about the secrets due to cache behavior. If the attacker can measure the timing difference, he knows whether the array element was loaded from main memory or from the cache. Consider Example 4. If an attacker guesses `public` equal to `secret`, only one array element is loaded into the cache instead of two. Thus, if the array was not previously loaded in the cache, the code will execute in approximately half the time it would otherwise execute, thus leaking the secret value.

```
function (int[] array, int public,
    int secret) {
    array[public]++;
    array[secret]++;
}
```

Example 4: Cache-based timing channel

Another source of architecture-based non-constant execution time is the branch-predictor. For many conditional branch instructions, the branch-predictor attempts to predict which code-path will be executed after the branch instruction, so the subsequent instructions can be loaded accordingly. If the branch is wrongly predicted, the loaded instructions need to be discarded and the instructions from the correct path need to be loaded, which results in extra processing time. The branch-predictor bases its predictions on the conditions of previous branch instructions, which means it records processed information. If a branch depends on a secret, branch-prediction is influenced by this secret, and information about it can leak through the timing-difference between correctly and wrongly predicted branches. This means that even if there is no control-flow dependent timing channel, the branch predictor can still leak the information about which code-path is followed.

2.2 Practical Properties

In many applications it is infeasible to completely remove all information leakage about secret variables. For example, in cryptographic software there is a relation between a secret key and the ciphertext, as a different key will result in a different

ciphertext. This relation can theoretically be abused in a brute-force attack if an attacker knows a plaintext and ciphertext pair. The attacker would encrypt the plaintext with different keys until he finds a result matching the ciphertext. He would then know the secret key, and this information has thus leaked through the encrypted result. However, this attack is generally not practical because of the enormous search space of possible keys. Likewise, a system is not necessarily insecure if limited information leaks through a side-channel. Certain properties of side-channels are important to determine their relevance. In this work we focus on dynamic range and maximum information leakage, which are explained below.

As explained in the previous section, exploitation of timing channels requires differentiation between multiple measurements of execution time. If a one bit channel is considered, this means an attacker needs to differentiate between the smallest and largest execution times possible, T_{smallest} and T_{largest} respectively. However, in practice it can be hard to differentiate between T_{smallest} and T_{largest} from actual measurements, because execution time is generally noisy, e.g., due to limited machine resources which the victim process shares with other processes.

Dynamic range is used to express the ratio between minimum and maximum values of a certain signal. It is a commonly applied ratio in the field of image processing, where it expresses the ratio between maximum and minimum image intensity [19]. We apply this concept to timing channels, and define dynamic range as the ratio between T_{smallest} and T_{largest}. This ratio gives an indication of practical differentiability, because it determines the impact of noise on the channel, i.e., the larger the difference between T_{smallest} and T_{largest}, the less impact noise has on differentiability. Because of the wide range of values this metric can have, it is expressed using the logarithmic unit of decibels. We formally define dynamic range as $10 \times \log \frac{T_{\text{largest}}}{T_{\text{smallest}}}$ dB. A dynamic range of 0 dB thus expresses indifferentiability of a channel, which means that no information can be derived from timing measurements.

Whereas dynamic range gives an indication about practical exploitability of a channel, maximum information leakage gives a hard limit on the information that can be compromised through the side channel. It is thus an indicator of the secrecy maintained after exploitation of the channel. Because information leakage is a direct result of differentiability between execution paths, and dynamic range expresses differentiability, information leakage can only exist if the dynamic range between paths is larger than 0 dB. Thus, dynamic range is an important property not only to reason about practical exploitability, but also to determine actual differentiability.

As demonstrated by Păsăreanu et al. [36], it is important to note the difference between maximum information leakage of a single run, and that of multiple runs, as a single run frequently leaks only limited information, whereas multiple runs may leak a secret entirely.

Nonetheless, a side-channel which can theoretically leak a secret entirely may be practically unexploitable due to a significantly low dynamic range. However, assessment of exploitability requires knowledge of the execution environment. For example, network-facing timing channels can have higher noise levels than

channels which can be measured on the machine directly, and may thus be naturally more resilient against timing attacks provided that the dynamic range is low.

Timing channel analysis needs to consider dynamic range to determine differentiability between paths.

Observation 1: Dynamic range

2.3 Timing Channel Mitigation

Techniques that mitigate timing channels can be categorized according to the practical properties they influence, i.e., they limit channel capacity or information leakage. Channel capacity can be limited in either of two ways, one of which is to limit the dynamic range, by making measurable time differences as small as possible. The other method is to make the channel unpredictable by adding extra noise, for example by inserting random delays in case of timing channels [18, 22]. However, effectiveness of decreased channel capacity may be limited, because in some contexts a determined attacker can still acquire information from the channel, given enough measurements.

If information leakage is limited, the side-channel can be either eliminated completely, or residual leakage may be accepted if it is required for functional code. One approach is to write constant-time code for specialized functions that handle secret data, such as cryptographic functions. For other programs, such as web browsers, this approach can be harder because it may not be immediately clear which functions handle secret data. Multiple authors have proposed techniques that transform code in such a way that control flow–based timing channels cannot exist in the transformed code [7, 31, 33]. However, automated code transformations may result in inefficient code, as the goal of the transformations is that all code paths exhibit equal execution time. This is acceptable for minor parts of the code which require a high level of security, but is generally unnecessary and unwanted for all code of a system. Thus, to apply transformations only to code sections that benefit from it, it is important to first identify and quantify the side channels.

Molnar et al. [33] defined a notion of security against control flow–based timing channels called program counter–security (PC-security). This notion asserts that secret variables do not influence the number of executed instructions. Their solution relies on special hardware which guarantees that all instructions execute in equal fixed time. Thus, if secret parameters do not influence the number of executed instructions, program execution time does not depend on secret parameters either.

However, the idealized hardware properties on which they rely are far from realistic for general purpose computers. PC-security thus guarantees observational determinism only for adversaries who can monitor the number of executed instructions, but not actual execution time. In practice, adversaries who can

1. Branch conditions should not, directly nor indirectly, depend on secrets.
2. Array size and referenced indices should not depend on secrets.
3. Instruction execution time should not be influenced by parameters which depend on secrets.

Definition 1: Timing channel prevention framework

measure actual execution time are more common than adversaries who can only monitor the number of executed instructions. Under such an attacker model, any mitigation technique based only on PC-security is necessarily incomplete and unsound.

Timing channel mitigation techniques thus require knowledge about code execution time. However, as some timing behavior is architecture-based, it is impossible to specify a concrete timing model which holds in general. Instead, a qualitative framework can capture the causes of timing channels in general. Such a framework can be concretized in quantitative cost models of instruction execution times, which allows to perform analysis for specific architectures.

Timing channel analysis should use accurate architecture specific cost models based on a qualitative framework.

Observation 2: Accurate cost models

2.3.1 Timing Channel Prevention Framework

Bernstein et al. [13] paid particular attention to avoid timing channels during construction of the NaCl cryptographic library. To prevent timing channels, they explicitly formulated two coding policies[1] that prevent timing channels. The first policy is to avoid branches controlled by secret data. The second policy is to avoid secret data as array indices. The timing channels prevented with these policies are control flow–based and branch-prediction-based timing channels, and cache-based timing channels respectively.

We construct a qualitative framework of timing channel prevention policies, based on the policies used in constructing NaCl. This allows us to reason about causes of timing channels as violations of these policies. However, we observe that the NaCl policies do not prevent the architecture-based timing channels based on secret parameters which we explained in section 2.1.2. We thus extend the framework with a new policy which requires that instructions operating on secret data cannot exhibit different timing behavior depending on secret parameters.

All policies can be found in Definition 1 for reference. If code adheres to

[1] https://nacl.cr.yp.to/internals.html

these policies, and the code is functionally correct, timing channels described in section 2.1 should not be possible. On the other hand, since violations of these policies result in differentiable timing behavior, these violations directly constitute differences in timing traces as described in the beginning of this chapter. In the rest of this paper we refer to violations of policy 1 as type 1 violations, and likewise for violations of the other policies.

Timing channel analysis needs to takes into account all known causes of timing channels.

Observation 3: Holistic view of channel causes

2.4 Timing Channel Identification

As timing channels can be viewed as policy violations, timing channel identification can be viewed as identification of these violations. Due to the different nature of the policies, different forms of analysis are required to identify violations. Data-flow analysis can be used to determine how data from one variable affects different variables and instructions in all stages of program execution. This can identify type 1 and 2 violations, as flows from secrets to insecure instructions assert these violations. In non-concurrent programs this approach can identify type 1 and 2 violations, up to the limits implied by the underlying data-flow analysis technique. However, the dynamic range of the channel may be impractically small and data-flow analysis is incapable of quantifying it.

On the other hand, data-flow analysis is insufficient to prove type 3 violations. Even though data-flow analysis provides an indication of possible type 3 violations, this approach is not sound. A sound approach, again up to the limits of the underlying technique, is achieved by introducing data constraint analysis, i.e., analysis which takes into account data constraints, to identify whether instruction parameters can influence execution time. To the best of our knowledge, no approach to identify type 3 violations has been proposed in the literature before.

Multiple approaches to identify type 1 and 2 violations have however been proposed. Molnar et al. [33] applied fuzz testing to verify that programs satisfy PC-security by comparing instruction-count among multiple randomly tested executions using different public inputs. PC-security violations are caused by branch instructions that depend on secrets, which means policy 1 compliance is a generalization of PC-security. Besides the limitations of PC-security mentioned in section 2.3, fuzz testing has limited code coverage, so it may miss program states which lead to timing channels. The advantage of tests using actual program execution is that effects of processor optimizations such as out-of-order execution may be identified. Of course it is vital in this scenario to test the program in the environment the program will be deployed in.

Multiple authors have proposed the use of self-composition techniques to verify non-interference and absence of timing channels in code [8, 10]. The

concept of self-composition considers a composition of a program with a copy of itself, operating on renamed variables. This allows comparative properties between multiple executions to be verified within a single program.

Almeida et al. [8] transform self-composed code with ghost-code and annotations such as preconditions and invariants, which allows them to store a notion of execution time within the program states. Subsequently, they apply automatic and interactive verification tools to prove non-interference. The security notion considered by Almeida et al. is an extension of PC-security, which includes data memory access patterns, so they can identify a wider range of type 1 and 2 violations. If non-interference cannot be proven, the verification tools show which code causes the violation.

Balliu et al. [10] apply self-composition in combination with symbolic execution to derive trees of observational states, in which each observational state includes the execution time up to that state. Self-composition in this case is not applied on the program itself, but on the observation trees. The trees are linked with a connector which defines the relation between variables in both trees. A verification function is then applied on both trees to assert that the observational states have equal timing behavior so that they do not leak secret information. The cost model applied by Balliu et al. equals the program counter model, so timing channel identification is limited to a subset of type 1 violations. Furthermore, this technique is limited to identify the presence of timing channels, but cannot locate the violating code.

We can make two observations about these techniques. Firstly, none of these techniques take into account type 3 violations. There is thus a gap between known causes of timing channels and causes taken into consideration by identification techniques described in the literature. Secondly, all of these techniques are based on the abstract program counter cost model. This cost model limits both soundness and completeness of test results. These techniques cannot be trivially extended to consider type 3 violations and accurate cost models, because the costs of type 3 violations are conceptually different, i.e., there is no one-to-one mapping between instructions and cost, as the context influences the cost.

Channel identification requires data-flow analysis and data constraint analysis.

Observation 4: Channel identification techniques

2.5 Channel Quantification

Side-channel identification is limited to identification of theoretical side-channels, but gives no information about their practical exploitability. For example, an attack may leak information about a secret key in some system. However, this does not necessarily mean this channel is practically exploitable, as the leaked

information may be insignificantly small and the effort to derive this information may be impractically large. This justifies quantification of practical channel properties such as information leakage and dynamic range.

We illustrate this scenario with example 5. Note that actual practicallity of exploitation depends significantly on execution context, here we approach this example from a high level perspective for brevity. Clearly, the branching condition depends on the secret variable, as it evaluates `public + secret != 0`. This line thus constitutes a type 1 violation.

The information leaked by this channel is a boolean which conveys whether `public` plus `secret` equals zero. If we assume 32 bit integers and an attacker who can choose `public` freely, this channel has a maximum information leakage of the entire secret, i.e., 32 bits. However, if an attacker can determine which path was taken on a single run, he only has a probability of $\frac{1}{2^{32}}$ of finding the secret in one execution, and this probability only increases slowly with the number of measured runs. The information leaked through this channel is thus very small in general.

The branch related to the true condition contains two arithmetic operations: add and divide, whereas the other branch contains no arithmetic operations. Arithmetic operations take only a very small number of clock cycles so if this function is called once in a large program, dynamic range will generally be very small, and it will thus be very hard to differentiate between the branches based on measurements of execution time. To derive information from this channel, an attacker requires a very accurate clock and would need to take a significant number of measurements to filter out noise.

```
function (int public, int secret) {
    if (public + secret != 0) {
        return 1 / (public + secret);
    } else {
        return 0;
    }
}
```

Example 5: Minor timing channel

To the best of our knowledge, the current literature on quantification of timing channels has focused solely on maximum information leakage [28, 36, 45]. These works reduce the problem of information leakage quantification to counting the number of observable execution paths influenced by secret variables. This allows to determine which equivalence classes the concrete value of a secret variable belongs to, thus giving constraints on the secret value.

Like the channel identification techniques described in the previous section, these techniques do not take into account type 3 violations. They would thus benefit from data constraint analysis and an accurate cost model. This allows to perform dynamic range quantification which can accurately determine differ-

entiability of paths, thus improving soundness and completeness of information leakage quantification.

Besides improving information leakage quantification, dynamic range quantification can directly be applied to give an indication about ease of channel exploitability, relevance and effectiveness of channel capacity–reducing countermeasures, and the certainty which an attacker has about information derived from timing measurements.

Channel quantification benefits from data-flow and data constraint analysis. Dynamic range is an important part of channel quantification.

Observation 5: Holistic channel quantification

Chapter 3

Symbolic Execution

Symbolic execution is a static white-box software testing technique, i.e., it tests internal software structures based on the code of the software. It is used to reason about program states and execution paths without executing the program using concrete values. Instead, variables are represented as symbolic expressions, i.e., the expressions determine which values a variable can have in each program state. The symbolic expressions can be used to assert complex program behavior. When symbolic execution identifies a bug, it can leverage constraint solvers to generate concrete test cases which trigger the bug.

The technique works by exploring execution paths, interpreting instructions, and processing them as operations on the symbolic expressions. When conditional branch instructions are encountered, symbolic execution can follow both sides of the branch, and add the required branch condition as a restriction on the relevant variables. When the end of an execution path is encountered, the path can be backtraced to a branch instruction and alternative paths can be followed, thus in theory covering the entire program state space.

Symbolic execution can be used to assert different types of program behavior. As mentioned in Section 2.5, Păsăreanu et al. [36] applied it to quantify information leakage of timing channels. They used symbolic execution to calculate the number of different program states that can be observed, and to determine which observations relate to which possible secret values processed by the program. As an example of completely different usage of symbolic execution, Shoshitaishvili et al. [41] applied symbolic execution to identify execution paths to certain privileged program points, which are not supposed to be executed without authentication. If such paths are found, it may constitute an authentication bypass and the path is further analyzed.

3.0.1 Challenges

Clearly, it is computationally intensive to symbolically execute large programs, as they have a large state space that needs to be explored. Another problem is that execution does not necessarily terminate, as not all programs have a set

end. It may thus be required to process programs up to a maximum execution depth. An alternative approach that may be applied to some problems is to test subsections of programs, such as individual functions like the examples used in this work.

Another significant challenge in symbolic execution is related to loop expressions. If the number of times a loop body is executed depends on a symbolic value – that is, it is not executed a set number of times – a state space explosion occurs. This means that there is an intractable number of execution paths to follow, and it thus becomes infeasible to test all possible program executions.

Secondly, programs can have external calls, e.g., to system functions and external libraries. However, symbolic execution frameworks have trouble parsing these calls because external libraries may be written in another language or the calls are system dependent. This problem may be alleviated by framework extensions, but in practice this frequently is infeasible as it does not work for off-the-shelf products.

Furthermore, constraint solving is an integral part of symbolic execution to generate concrete input which can trigger the bugs discovered during testing. However, constraint solvers cannot solve all constraints, or may take a significant time to do so. This aggravates testing time and may prevent generation of concrete test cases.

These challenges limit test coverage of some programs, and test results may thus be incomplete. That is, the symbolic execution is not incomplete per se, but if it cannot explore the entire state space in practice, the results are incomplete. Furthermore, soundness of results may be limited to soundness and completeness of the constraint solver on which the execution framework relies.

A promising technique which can alleviate some of the problems of symbolic execution is known as *concolic* execution, first introduced by Sen et al. [40]. It uses a combination of both concrete and symbolic execution. Because concrete execution does not suffer from the same problems as symbolic execution, execution speed is significantly increased and analysis can more easily attain a greater depth in a program's execution path. However, due to the concretization of symbols, certain program states may be missed and completeness of analysis results may be limited.

The main focus of our work is timing channel analysis, and not to improve symbolic execution techniques. Having said that, these challenges play an important role in the practical appliance of any technique based on symbolic execution. We consider cryptographic libraries the main target for our analysis, and these naturally have a set execution end, rarely rely on external calls, and rarely contain loops of an unpredictable number of cycles. On the other hand, cryptographic implementations may generate highly complex constraints which can pose a significant challenge for constraint solvers. We accept this as a limitation of the used technique, but later argue how a significantly challenged constraint solver can actually disclose timing channels.

Symbolic execution has practical limitations which can prevent it from exploring entire state spaces.

Observation 6: Limitations of symbolic execution

3.1 Symbolic Quantification

As discussed in Section 2.3, the policies from Definition 1 prevent timing channels on modern machines. Because violations of these policies are the source of timing channels, timing channel analysis is centered around these policy violations. *Data flow analysis* is used to determine how data is propagated through a program, and can thus identify dependencies on secret variables. This can be used to identify type 1 and 2 violations. *Data constraint analysis* is used to determine constraints on variables and parameters in different program states. As such, it can be used to determine when parameters can influence timing behavior of instructions. Combined with data flow analysis, this can identify type 3 violations. These concepts can thus be used to identify possible timing differences in program states.

Symbolic execution can be used both for data flow analysis and data constraint analysis, as this information can be derived from the symbolic program states and expressions on which it operates. This allows symbolic execution to identify all types of policy violations. Data flow analysis can also be performed using other techniques; however, the true strength of symbolic execution for side-channel analysis is that it can automatically characterize and compare program states and code paths by complex relations, as we discuss in the following subsections.

3.1.1 Information Leak Quantification

Symbolic execution can be used to determine constraints on variables in specific program states. Since constraints express information about the values they constrain, symbolic execution can be used to determine what information an attacker can learn through a timing channel. By computing the constraints in program states which are differentiable by timing differences, we can determine exactly what the attacker knows about a secret if he learns that a program reaches these states.

By computing which paths exhibit timing differences, and analyzing the constraints learned through those differences, paths that exhibit the most information leakage can be identified. Furthermore, a constraint solver can be leveraged to generate concrete program input from those constraints, which leads to the measurable program behavior.

However, constraint solvers are not fit to reason about the feasibility set of a constrained variable, i.e., all possible values the variable can take. Thus, given complex constraints on a secret, it is hard to determine the exact amount of information which is expressed about the secret.

3.1.2 Timing Characterization

Differentiation between code paths and instruction behavior requires knowledge about instruction execution time. However, policy violations by themselves are insufficient to determine path timing properties. Instead, one needs to know the instructions in each path, and the timing behavior of each instruction.

Symbolic execution explores code paths step by step, which means it can determine exactly which instructions are executed in each path and which parameters they take. Given these paths, and the constraints on parameters in those paths, one can determine costs for execution paths using a cost model containing execution time for instructions and parameters.

By comparing costs of different paths, one can characterize timing differences between these paths and thus quantify the dynamic range of timing channels. The dynamic range of a timing channel can then be used to predict exploitability of the side-channel, or to perform accurate analysis of which paths can be differentiated, which is required for accurate information leakage analysis.

Even though this technique can give an indication of the time a processor spends executing a program, multiple programs share processor time on modern architectures. Therefore, noise, in the form of perceived extra execution time, is introduced by other processes competing over execution time and other resources. Path timing behavior as characterized by this technique thus cannot be compared to actual measurements directly. Instead, it gives an indication of relative execution time of different path sections.

In this sense, this technique is related to template attacks as described by Chari et al. [16]. Template attacks are based on a characterization of side-channel signal and noise behavior, which can be compared to actual side-channel measurements to derive secret information in a minimal number of runs. Originally applied in power analysis attacks to characterize power-consumption over time, templates may also be constructed for other side-channels. In this case, the dynamic range characterizations can be considered templates of observational points over time. These templates may be used in actual exploitation of a timing channel.

Symbolic execution can characterize path timing behavior in the form of dynamic range, and can furthermore be used to quantify information leakage in the form of constraints.

Observation 7: Quantification options

Chapter 4

ARM Architecture and Instruction Set

Up to this point we have explained the concepts of timing channels and symbolic execution using high-level pseudo-code. However, it is impossible to determine an accurate timing-model for high-level instructions, as compilers can perform optimizations which have a large impact on performance. Thus, in this section we will give a description of ARM Cortex-A7 processor specifics, and the way its instruction set, ARM Assembly, works.

4.1 ARM Assembly

The instruction set for the Cortex-A7 contains many instructions with different encodings, options, and possible operand values. Subsets of the instruction set which the processor can execute are known as ARM, Thumb, ThumbEE, Jazelle, and NEON. In this work, we focus on a set of core ARM instructions, and leave all special instruction subsets, including floating points arithmetic, for future work. The instructions we focus on are those from the *Encoding A1* field of the ARM Architecture Reference Manual for ARMV7-A and ARMv7-R. This includes standard instructions such as arithmetic operations, control-flow instructions, and memory instructions.

Assembly language does not, like high level programming languages, keep track of variables. Instead, it operates directly on 15 processor registers, named `r0` to `r15`. Certain registers have pseudonyms, such as `r13` – the stack pointer – which is also referred to as `sp`. Furthermore, assembly instructions can operate on memory, and the *state flags*, which keep track of the results of special comparison operations. These flags are used to determine whether or not conditional instructions are executed.

Assembly instructions typically take multiple operands, which are either registers, constant values, or memory locations. Most arithmetic operations require an output register parameter, and multiple operands of which at least one is a

register, the other operand can be either a register or a constant value for many operations. However, this format is not a rule, as there are multiple instructions which differ from this format. An example of an arithmetic instruction is `SUB r0, r0, #1`, which subtracts 1 from the value stored in `r0`, and stores the result to `r0`. Shorthand for this instruction is `SUB r0, #1`, because `r0` is used both as input and output register.

Furthermore, there are instructions which accept *shifted registers* as operands, i.e., they take a register, a shift type, and a shift distance operand. The register is then shifted by the distance of the second operand, and the result of that operation is used as the result for the actual instruction. For example, `SUB r0, r1, LSL r2` performs a logical shift left on `r1`. The distance by which it is shifted is acquired from `r2`. Then, the result of this is subtracted from `r0`, and stored back into `r0`.

Assembly instructions can perform different kinds of arithmetic on the contents of registers.

Observation 8: Arithmetic instructions

Memory instructions can move values between registers and memory locations. Memory locations are written between square brackets, and may be composed of multiple operands. For example, `[r0, -#1]` refers to the memory location acquired by subtracting 1 from `r0`. The accepted memory location formats depend on the instruction. An example of a memory instruction is `LDR r0, [sp]`, which loads the value from the the position in memory referred to by the stack pointer, and stores the result in r0.

Values are transferred between registers and memory by dedicated instructions. The memory addresses can be constructed from the content of registers.

Observation 9: Memory instructions

Almost all instructions in the instruction set can be appended by a *condition code*. The condition is computed at run time from the processor flags, and determines whether instructions are actually executed or not. There are sixteen different condition codes, including `AL` for always execute, and `NE` for never execute. The code `AL` is generally discarded for brevity, so that an instruction without condition code is one that always executes. Common condition codes are `EQ` for equals, and `NE` for not equals. `EQ` actually checks whether the Z-flag is set to 1, and `NE` checks whether it is 0.

State flags are set by comparison instructions, which perform simple arithmetic and update status flags based on the result. The result itself is discarded. For example, `CMP r0, #0` subtracts 0 from `r0`, and updates the status flags accordingly. For example, the Z-flag is set to 1 if all bits in the result are zero, and it is set to 0 otherwise. Furthermore, most arithmetic instructions of the instruction set can be appended by an `S`, which means that the instruction should update the status flags.

Branch instructions control execution of the program, by changing the value of the program counter, and thus affecting the executed instructions. This is also known as *jumping*. Branch instructions can jump either to instruction addresses, or to named *labels*, which are prepended to instructions by a colon. Like other instructions, branch instruction can be conditional, so that control flow can depend on the program state. For example `BNE loopstart` will conditionally jump to the label *loopstart*, given that the Z-flag is set to 0. Besides dedicated branch instructions, almost all arithmetic instructions can write their results to the program counter register directly, to form dynamic branches based on the operation.

Program control-flow is determined by branch-instructions which are controlled by the state flags. The state-flags can be set by dedicated comparison instructions as well as by normal arithmetic instructions.

Observation 10: Control-flow instructions

```
LDR r0, [sp];
loopstart:  SUB r0, #1;
...
CMP r0, #0;
BNE loopstart;
```

(a) ARM Assembly

```
do {
--i;
...
} while(i != 0);
```

(b) Pseudo-code

Example 6: Do-while loop

Putting it all together, we get code such as that from example 6(a). For comparison, in high-level pseudo-code this code could be written as example 6(b). The assembly code works as follows; first, a value is loaded from the top of the stack, and stored in register `r0`. Then, 1 is subtracted from the value. "`...`" is actually replaced by a loop body. Then, `r0` is compared to the value zero, i.e., 0 is subtracted from the value in `r0`, and if the result is 0, the Z flag is set to 1. Finally, if `r0` did not hold the value zero, control jumps back to the label `loopstart`, and repeats the loop.

4.2 ARM Cortex-A7 Pipeline

An instruction pipeline is a technique applied in processors to introduce a degree of parallelism in instruction execution. A pipeline is constructed of multiple stages, which each have specialized functions to perform a tiny part of the execution of an instruction. This allows to feed instructions into the pipeline, which move through the pipeline stage by stage, so that at any given time each stage is processing part of a different instruction.

The Cortex-A7 implements an eight to ten stage in-order pipeline, through which instructions typically move by one stage per clock cycle. If a pipeline is

in-order, it means that instructions cannot overtake each other in the pipeline, so they will always be executed in the order that they enter the pipeline. In comparison, out-of-order pipelines allow faster instructions to finish before slower instructions that entered the pipeline earlier. The stages and functionality of the Cortex-A7 pipeline are:

Stages 1 − 3: Preprocessing

Stage 4: Decode

Stage 5: Issue

Stages 6 − 7/10: Execution

Final stage: Result writeback

Figure 4.1 shows a high-level view of the pipeline, where each block typically takes one cycle to process in theory. The execution stage relies on five distinct execution units, which each have specialized functions. The integer, multiply, dual issue, and load/store units each have two stages, and the floating point / NEON unit has 4 stages. The integer unit is used for all simple integer operations. The multiply unit is dedicated to multiplication operations. The floating-point / NEON unit is a unit for complex instructions which operate on floating point and vector operands. The dual issue unit is a unit specifically intended to speed up the pipeline by allowing to execute multiple instructions simultaneously. It is essentially a stripped down integer unit which can execute a subset of simple integer operations. The load / store unit handles memory access.

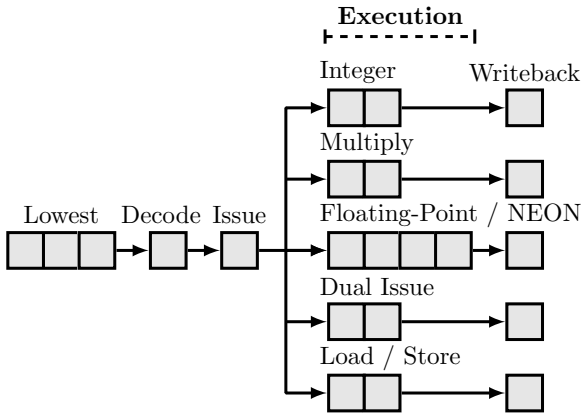

Figure 4.1: ARM Cortex-A7 pipeline based on [21]

In applications, it is common that instructions take results from previous instructions as operands. However, a previous instruction may still be executing

at the moment a following instruction requires its result. If this happens, the latter instruction cannot continue through the pipeline until its operand becomes available. Thus, the pipeline will halt until the previous instruction finishes. This phenomenon is known as a pipeline bubble, because there are stages in the pipeline which are not processing any instructions. Typically, instructions require operands to be available at the start of execution, although there are exceptions that we will discuss later.

There are multiple optimizations implemented in processors to speed up execution wherever possible. For certain instruction combinations, the processor can bypass the results from one instruction to the next instruction, circumventing the wait for the writeback stage to complete. Furthermore, some instructions can issue, execute, and write their results in a single cycle.

The processor uses a pipeline to handle instructions. The instructions are processed in multiple stages and are executed by dedicated units. The speed by which instructions move between stages can vary depending on the context.

Observation 11: Instruction handling

It is important to note that execution time for memory access instructions are not only dependent on the processor, but also of the speed of memory and caches. Since memory and caches are significantly slower than registers, these instructions typically take much longer to execute than the theoretical two cycles of the pipeline model.

Part II

Contribution

Chapter 5

ARM Cortex-A7 Timing Model

In order to compute program execution time at a cycle-accurate level, a model is required that describes how much time a processor takes to execute instructions. As explained in section 2.4, previous approaches relied on the abstract program counter (PC) model, which models execution time by the number of executed instructions. Actual execution time cannot be computed using an abstract model like this, because there are too many differences between processors, and too many factors to take into account for each instruction.

There is obviously a trade-of between high detail models versus low detail models. Highly detailed models can give a more accurate quantification of timing behavior than models of low detail. However, as they are tailored to specific processors, their results cannot be applied to other processors. Furthermore, processing a highly detailed model requires more computational resources than processing a low detail model, thus it is more intensive to model a program's execution time. It should be clear that we do not claim the program counter model has no value; but if one is required to make hard claims about a program's execution time, a more detailed model is of more use.

Because our goal is to model time more accurately, a specific target processor is required. The processor core we chose to model is the ARM Cortex-A7, a ubiquitous processor found in many smartphones. This processor was chosen for the benefits of its simple pipeline and limited instruction set, which makes it relatively easy to create a detailed execution time model, while still leading to results relevant for real applications. Specifically, we focus on the Broadcom BCM2836 implementation of the Cortex-A7, as used on the Raspberry Pi 2 Model B revision a01041. We validate our results on the Allwinner H3 implementation of the Cortex-A7, as used on the NanoPi NEO version 1.2.

5.1 Timing Definition

So far, we have talked about an abstract notion of "execution time". For a process, it is naturally clear what execution time means: the time it takes for a program to run from start to execution. However, when looking at a cycle-accurate level of instruction execution time in a pipeline, things become more complicated. Now that we have explained the pipeline in section 4.2, we introduce two important facets of execution time: issue time and latency. Based on our understanding of the pipeline and the descriptions of these concepts in [3], we constructed our working definitions for issue time and latency as can be found in definition 2 and 3.

Issue time is the number of clock cycles an instruction spends in the issue stage, plus the number of cycles it *halts* the pipeline in execution, but not waiting for an operand.

Definition 2: Issue time

Latency is the number of clock cycles required before an instruction's output register becomes available, counted from the moment the instruction entered the issue stage.

Definition 3: Latency

As explained in section 4.2, instructions typically spend one cycle in each stage. Thus, issue time for most instructions is a single cycle. However, some instructions require more time, thus preventing following instructions from starting execution. When we talk about instruction execution time, we refer to both issue time and latency.

5.2 Contextual Influences

The publicly available information about the Cortex-A7 is very limited. A high-level description of the instruction pipeline has been published by ARM [21], and the GNU Compiler Collection (GCC) source code has a more detailed model of the Cortex-A7 pipeline [2]. ARM has however published more detailed information about instruction execution time for other processors, such as the Cortex-A8 [3], which is a predecessor of the Cortex-A7. Both Cortex-A7 and Cortex-A8 implement the ARMv7-A architecture. These sources discuss several factors which influence execution time depending on the context in which instructions are executed. We perform measurements to identify how these factors influence instruction execution time.

5.2.1 Dual-Issuing

As explained in section 4.2, the dual issue unit can speed up program execution. It can process an instruction simultaneously with another instruction that is being processed by the integer unit. In practice, there are certain instructions which can dual issue *as older*, and those which can dual issue *as younger*. This means that when an instruction which can dual issue as older is followed in the pipeline by an instruction which can dual issue as younger, they issue simultaneously. The first is processed by the normal integer unit, and the latter is processed by the dual issue unit. According to the GCC pipeline description of the Cortex-A7 [2], integer arithmetic and logic instructions which have an immediate operand can dual-issue as younger, and other integer arithmetic and logic instructions can dual-issue as older; this does not include multiply instructions, which are handled by the multiply unit.

5.2.2 Early and Late Operands

Certain instructions for the Cortex-A8 have early and late operands [3], i.e., the register content of some operands needs to be available respectively one cycle earlier and one cycle later than the normal start of execution. This can influence the effect that the latency of an instruction has on a second another instruction following it, thus influencing the combined execution time. We test whether instructions on the Cortex-A7 demonstrate the same behavior.

5.2.3 Bypasses

Under normal conditions, the output of an instruction is written to a result register in the last stage of the pipeline. Subsequently, a following instruction can use this register as an input. However, under special conditions, results can be *bypassed* to inputs of a following instruction, i.e., the result becomes available to the second instruction before it is written to the output register. Specifically, it is claimed that multiply instructions can forward their result to the accumulator operand of a multiply and accumulate instruction following it [2].

5.2.4 Condition-Codes

In the Cortex-A8, conditional instructions which take longer than 2 cycles to execute are known to have variable execution time if the condition is unsatisfied [3]. We test whether instructions on the Cortex-A7 demonstrate similar behavior.

5.2.5 Memory Instructions

Memory accesses are affected by multiple sources. If a memory location is stored in the cache, it can be stored and retrieved significantly faster than when a write to main memory is required. Similarly, the translation lookaside buffer

(TLB) stores a mapping from virtual memory addresses to physical memory addresses. Like with the cache, if a translation of a virtual memory address is already stored in the TLB, memory access is significantly faster. Because of these external influences, it would be an entire work on its own to accurately model the behavior of these operations. Instead, we model a minimum access time, i.e., level 1 cache access. We regard the effects of external factors as noise.

An accurate model of the variable memory access times would be interesting to have. However, there are multiple reasons why we do not consider this necessary for our work. Firstly, our model should ultimately be viewed in the context of timing channel identification. With regard to memory accesses, we consider it more important to model the existence of timing *differences* than to model quantification of the timing differences. This is partially because we consider it impossible to predict exactly how long any memory access takes, because cache and TLB are external factors, influenced by other processes running on the same machine. Lastly, the actual access times depend on the memory chip used, and we consider this outside the scope of this work.

5.2.6 Branch Instructions

When the processor identifies a conditional branch instruction in the pipeline, the branch predictor predicts whether the branch is taken or not. It loads instructions from the predicted address to continue program execution. However, if the prediction was wrong, the wrongly loaded instructions are flushed from the pipeline, and the correct instructions are loaded instead.

5.3 Test Environment

We construct the timing model based on measurements performed on the Raspberry Pi. Specifically, we use the Raspberry Pi 2 Model B revision a01041, running Raspbian 8.0. Timing behavior for the core instructions have been confirmed on the NanoPi NEO version 1.2, running Ubuntu 16.04.2. Measurements are performed by binaries which measure the number of execution cycles which pass while they execute specific instructions. The test programs are written in C, using the processor's performance events interface to measure the number of executed cycles through the PERF_COUNT_HW_CPU_CYCLES counter.

The measured assembly instructions are written in the program using the GCC inline assembly function "__asm__", which allows to directly embed assembly instructions in C code. This setup allows for easy automation of the measurement process, using scripts to construct and compile code for the different instructions to measure. All test programs are compiled with GCC 4.9.2 using the -O0 flag to turn off compiler optimizations. We first describe which instructions have been tested, and then discuss how their execution time is measured.

5.4 Instruction Scope

As explained earlier, we limit our focus on the core instructions of the ARM instruction set. We compiled a set of 351 encoding formats from the *Encoding A1* field of each instruction description in the ARM Architecture Reference Manual for ARMv7-A and ARMv7-R [1].

These instruction formats contain fields for options and operands. There are too many possible options for each field to test every combination. Thus, these fields were initialized according to table 5.1. Operand numbers were chosen semi-arbitrarily, based on the values allowed among all different instructions taking such an operand field. For each instruction format, each possible combination of field values was created and measured.

Table 5.1: Instruction operand fields and tested values

Field	Meaning	Test values
<const>	a constant number	0 and 77
<imm>	different sizes immediate values	0 and 15
<lsb>	a constant in the range 0-31	3
<width>	a constant in the range of 1 to 32-lsb	8
<r*>	different registers	the leftmost register operand r1, and each following register incremented by one
<shift>	a constant-distance shift operation on an operand	LSL #3, LSR #3, ASR #3, ROR #3 and RRX
<rotation>	a bit rotations in the operation	ROR #8
<type>	a register-based shift type on an operand	LSL, LSR, ASR and ROR
<x>	B or T to determine which half of an operand is used for an operation	B
<y>	B or T to determine which half of an operand is used for an operation	T
<+/->	determines whether an operand is added or subtracted to another operand	+ and -
<c>	the condition code of an instruction	the 15 different conditions
<q>	determines whether assembler selects 16-bit or 32-bit encodings of an instruction. Can only be .w when assembling to the ARM instruction set.	.w
{}	can surround any part of an instruction format to make it optional.	each optional part was included and not included

Instructions using the fields <spec_reg>, <endian_specifier>, and <option> were not measured. This has not posed any limitations to our case studies as we did not come across instructions using these fields.

5.5 Methodology

The timing model is constructed by measuring issue time and latency of executed instructions. The instructions are executed in different contexts, to measure the effects and existence of the influences described in section 5.2. To filter out noise, instructions are executed multiple times – typically 512 to 2048 times – and the measured time is divided by that factor. This process is repeated multiple times for each instruction and the quickest execution time is regarded least noisy. This approach leads to results which are over-approximated by no more than a twentieth of a cycle, which we explain by noise and overhead from the cycle-measuring function calls. The results are rounded down to compensate for this over-approximation.

5.5.1 Initial Measurements

A first indication of issue time is acquired by consecutively executing the same instruction. Execution time does not depend on availability of registers, because different registers are used for all source and destination operands. This setup thus measures issue time. However, this setup does not take into account any instruction optimizations which influence execution time by specific instruction interactions. Specifically, this measurement leads to false results for instructions which can dual-issue as younger. Because all instructions which can dual-issue as younger can also dual-issue as older, all such instructions dual-issue and are thus measured at half the actual execution time. However, since all instructions which can dual-issue have an issue time of 1 cycle, all instructions which can dual-issue as younger are known after the initial measurement because the measurements say they take up 0.5 cycles.

5.5.2 Dual-Issue as Older Instructions

We made the assumption that dual-issuing does not discriminate between specific younger and older combinations, i.e., each instruction which can dual as younger, can dual-issue with each instruction which can dual-issue as older. We have found no evidence to disprove this assumption. Based on this assumption, any instruction which can dual-issue as younger can be used to identify all instructions which can dual-issue as older. To do so, the measured instructions are interleaved with an instruction which can dual-issue as younger, again operating on distinct input and output operands, and measured again. One cycle is removed from the measurement result, to compensate for the interleaving instructions. The instructions which can dual-issue as older can then be identified because they are measured as 0 cycles. For the other instructions the actual issue time is measured.

5.5.3 Latency

For the following measurement setups, we refer to *known-timing instructions* by which we interleave instructions we measure. The chosen instructions do not matter specifically, as long as they are not affected by optimizations in specific combinations of instructions. The problem with our measurement setup is that we must rely on machine instructions, to measure other machine instructions. Thus, if we have an incomplete understanding of the instructions supporting the setup, this can negatively influence our understanding of other instructions. Based on trial-and-error, we assume that the issue time of the multiplication instruction `MUL` is not influenced by any specific instructions executed before it. If such behavior is unknown, consistency of results needs to be validated in hindsight. We have not found any evidence that disproves this assumption.

Because instruction issue time is known from previous instructions, latency can be determined by interleaving the tested instructions with a known-timing instruction which takes an output operand of the measured instruction as an input operand. A naive approach is to feed the output from the tested instructions into the input from the following tested instructions. However, this does not take into account timing influences of the input operands, such as early and late operands. However, by interleaving the instructions with a known-timing instruction, the measurements are performed under the same conditions for all instructions.

Again, the execution time of the interleaved instructions needs to be subtracted to compute the latency. Because the interleaved instructions depend on the tested instruction's output, the latency of the instruction is measured. Not all instructions have output operands, so latency does not apply to these instructions. In our measurement results, this demonstrated as measured timed being equal to the instruction's issue time.

5.5.4 Early and Late Operands

To measure which instructions take early and late operands, the setup for measuring latency is reversed. In this setup, the *input* of the measured instructions depend on the *output* of interleaved known-timing instructions. Taking into account the interleaved instructions, computing the difference between the measured time and the time measured under normal conditions reveals how much earlier or later operands are required.

5.5.5 Bypasses

The last operand of the `MLA` instruction can receive bypassed values [2]. To determine which instructions can bypass their results to this operand, the tested instructions were interleaved with `MLA` instructions, taking the tested instruction's output register as the last operand. The measured result was compared to the timing expected from other measurements. To determine which instructions have registers which can receive such bypassed results, the setup was reversed,

i.e., an instruction of which it is known that it can bypass its result, was interleaved by instructions which took the bypassing instruction's output as input.

5.5.6 Condition Codes

To measure the effect of condition codes, two measurements were performed for each condition code. For the first measurement, the flags are set so the condition is unsatisfied, and the measured instruction thus does not execute. For the second measurement, the flags are set so that the condition is satisfied, and the instruction does execute. The execution time of these instructions was first measured separately so that this overhead could be subtracted from the measurements.

5.5.7 Memory Instructions

For the measurements of other instructions, operand fields were chosen without considering the instruction result. However, our setup demands that memory instructions receive more attention. Because our measurement setup is located in user space of a general purpose operation system, we cannot access arbitrary memory locations. However, even if our code was run on the hardware directly, writing to arbitrary memory locations could overwrite the measurement code, and arbitrary memory accesses are thus inadvisable.

To compensate for this, memory addresses were not arbitrarily constructed from the field values of table 5.1. Instead, all memory accessing instructions had their addresses computed relative to the stack pointer, i.e., we claimed a block of multiple bytes on the stack and used their location for our accesses. Furthermore, the multiple store and multiple load instructions were tested with different size register lists.

5.5.8 Branch Instructions

A special setup was required to test branch instructions for two reasons. Firstly, the branch-predictor influences execution time of some branch instructions. Secondly, branch instructions directly influence the executed program paths, thus they can also influence the measurement logic. Therefore, extra logic is required to jump to valid locations and still allow for meaningful measurements of the instructions. Branch instructions were measured in a loop in which they could jump to an address later in the same loop.

The jump instruction which caused the program counter to jump from the loop-end to the start of the loop was implemented using a conditional move, because we found that the branch-predictor is not influenced by this kind of jumps. The conditional branch instructions were first tested while the branch predictor could properly predict whether or not the instruction would branch, i.e., no logic was implemented to disturb the branch-predictor. Since the branch-predictor correctly predicts almost all the branches, the minimum branch time was measured with this setup.

To measure the worst-case branching time, the effect of the branch predictor needs to be mitigated. We implemented a linear-feedback shift register (LFSR) in the measurement loop to serve as a pseudo-random function on which conditional branches depend. The decision to use an LFSR was made because of ease of implementation in a limited number of assembly instructions with constant-time performance. The LFSR we used is $x^9 + x^5 + 1$, which has a period of 511. The LFSR maximally confuses the branch predictor, so that it wrongly predicts 50% of the branches. It thus creates an environment in which the recovery time from branch prediction failures can be measured.

5.6 Results

Our measurements show that all factors discussed in section 5.2 influence instruction execution time in the Cortex-A7. We compiled the most common instructions we characterized and their properties in a table which can be found in the Appendix.We discuss several conclusions that we draw from our measurements.

Issue time Most instructions exhibit an issue time of 1 cycle in the optimal case. Only the branch with link instruction, and instructions which load or store multiple memory addresses take longer. All memory access instructions can take longer based on cache misses.

Latency Integer arithmetic and logic instructions have 1 cycle latency. Latency of multiplication instructions is 2 or 3 cycles.

Conditional execution Issue time is generally not influenced by conditional execution, except for memory instructions. Latency of multiply instructions is reduced by 1 cycle in case of unsatisfied conditions.

Dual-issuing as younger Integer arithmetic and logic operations which operate on a constant value and at most one register can dual-issue as younger. Multiply operations, as well as shift and rotate operations, reverse subtract operations, and the negative move operation, are not included in this group.

Dual-issuing as older Integer arithmetic and logic operations can dual-issue as older. This includes shift and rotate operations, reverse subtract operations, and the negative move operation, but excludes multiply operations.

Early operands All shift instructions and instructions which operate on shifted operands have early operands. Furthermore, all memory-access operations require the registers which compose the memory address to be available early. All early operands are required 1 cycle early.

Late operands Memory store instructions have late operands, which are always required 2 cycles late. Accumulator instructions also have late operands, see the next paragraph about this.

Bypasses We have tested the bypass behavior fo the MLA operation and other instructions with accumulator behavior. However, our measurements show that all instructions have reduced latency when their output is used as an accumulator operand. For memory instructions latency is reduced by 1 cycle, for all other instructions it is reduced by up to 2 cycles, depending on issue time. We think the claim that there is an actual bypass taking place is incorrect, and instead categorize it as late operand behavior, because there is no unique behavior where only the latency of a limited set of instructions is reduced.

However, the measurements have also shown different timing behavior for memory instructions output for many other instructions. Almost all logic and arithmetic instructions have what we call *semi-bypasses*. All operands which except semi-bypassed input, reduce latency of the instruction that produces that input by 1 cycle, if the input is produced by a logic or arithmetic operation.

Memory Execution time of memory instructions depends on cache hits and misses. Furthermore, if a memory access is not properly aligned with memory alignment, an extra memory access is required to process the instruction; e.g., a word aligned memory requires two accesses to process a memory operation on a value which is stored across two words.

Branch instructions There are dedicated branch instructions and dynamic branch instructions, which are arbitrary operations which write their results to the program counter register. Dedicated branch instructions are affected by the branch predictor, and take up to 7 or 8 cycles longer to complete if the branch predictor wrongly predicted the execution path. Dynamic branch issue time is increased by 9 cycles compared to execution of the same instruction on a generic register. Issue time of conditional dynamic branch instructions with unsatisfied conditions is increased by just 1 cycle.

5.7 Limitations & Discussion

This chapter discussed details of the ARM Cortex-A7, it also described how we measure execution time for a subset of its instructions and our findings from those experiments.

As noted before, memory instructions rely not only on the processor, but also on the speed of memory and caches. Thus, for these instructions, the time from our model only applies to the Raspberry Pi 2 model B. Whether similar times can be expected for other systems cannot be said without measuring execution time for the specific systems.

Furthermore, any timing model can only give a limited indication of actual execution time of programs executing in an environment shared with other programs. Operating systems and other programs executing on the same system demand processor time and affect the system state such as cache population and the state of the branch-predictor. Therefore, a certain amount of unpredictable noise is always to be expected in an undedicated environment.

One major obstacle to define a generic methodology to perform measurements of instruction execution time for any processor, is the wide variety of instruction time behavior. The specific types of behavior such as early and late registers, bypasses, and other optimizations, cannot be measured unless they are first identified. Since time behavior information for most processors is only sparsely publicized by producers, it is easy to miss certain behavior. Furthermore, the number of possible instruction and operand combinations is too large to cover exhaustively. Because measuring different facets of instructions ultimately relies on other instructions, unknown timing behavior can always influence measurement results which causes a certain degree of uncertainty for all these measurements, and thus for the entire model.

Our model has been constructed with the hope that we have identified all facets of timing behavior for the Cortex-A7. We have guided our tests by information which was publicly available about the processor's timing behavior. However, due to the reasons mentioned above, there may still be timing properties missing from our model. Therefore, we cannot claim that our model is complete, but it is significantly more detailed than the program counter model. We see that most instructions take 1 cycle issue time, and for most other instructions execution time is in the same order of magnitude. Therefore, we must also conclude that the program counter model is not very far off in execution time prediction.

We have identified differences between our model, and the model used by GCC [2]. Among others, we characterize instruction cycles in a more detailed way. We characterize true and false conditional behavior. We characterized more detailed accumulator behavior, and we identified early and late registers. The work in this chapter has contributed to our understanding of the Cortex-A7 and has identified conditional instructions with non-constant execution time. The model resulting from this work gives us the ability to identify timing channels with greater accuracy.

The Cortex A7 has a relatively simple in-order pipeline and simple execution units, processing a limited instruction set. This has presumably made it much simpler to model execution time for this processor than it is for other processors. We have not attempted to create timing models for other processors, thus we cannot determine how hard this would be exactly. However, we expect that it may be a major hurdle to create a detailed timing model for some processors, e.g. due to multiple pipelines or out-of-order execution.

5.8 Timing Channel Causes

To place the properties of the Cortex-A7 into perspective of timing channels, we shortly identify how the different violation types may occur.

Type 1 violations, based on conditional branches which depend on secret information, occur when secret information influences execution time through a branch instruction. This timing difference can depend on two different factors: After a conditional branch, the processor executes different instructions depending on which address the processor branched to. Thus, a timing difference can be based on the timing difference of instructions in these different paths. Secondly, it can be based on a timing difference between branch prediction success and failure, as a failed branch prediction requires the pipeline to be flushed and new instructions to be loaded. As we explained, the branch predictor does not operate on all forms of branches in the Cortex-A7, thus not all branches can cause timing channels based on branch prediction.

Type 2 violations occur when secret information determines the address of a memory access. All memory accesses in the Cortex-A7 are performed by store instructions and load instructions, which are clear to distinguish from other instructions. Whenever their address operands rely on secret information, a cache-based timing channel can exist and thus a type 2 violation is present.

Type 3 violations occur when instruction operands which depend on secret information determine an instruction's execution time. We identified that both issue time and latency of several conditional instructions depend on whether or not the condition is satisfied. We regard status flags as operands to conditional instructions. Thus, if secret information influences the processors status flags and a conditional instruction with variable timing relies on those status flags, type 3 violations occur.

Chapter 6

Timing Channel Analysis

In this chapter we describe a novel approach to identify timing channels in binary code. Our approach leverages symbolic execution, self-composition proofs, and a model of instruction execution time at any level of detail, to model the relationship between program parameters and program execution time. The technique can be used for identification of timing channels, quantification of timing differences, and to symbolically express how this behavior is caused by program or function parameters.

It differs from previous timing channel identification approaches in several respects. It is the first such approach which can operate completely independent of the abstract PC-security model. We achieve this by working on a detailed symbolic model of the processor's instruction execution time. Therefore, it can analyze programs with much higher accuracy than any approach before. Due to this accuracy, it is the first technique which can identify type 3 timing channels, and give a more accurate indication of channel exploitability.

The technique leverages symbolic execution and self-composition proof obligations to prove non-interference in execution time. Unlike previous approaches that rely on self-composition techniques, we perform self-composition not to prove non-interference of complete program states, but instead focus on Δtime: the time that individual instructions require to execute. By proving non-interference in Δtime, we can identify timing channels *during* symbolic execution, and do not require the analysis to run to completion before giving results. Therefore, our technique can pinpoint exactly which instructions lead to timing channels, and it can easily identify timing channels in non-completing code.

6.1 Symbolic Execution

Binary symbolic execution engines traverse execution paths by symbolic execution of assembly instructions. The machine state, i.e., register and memory content, is recorded in the form of symbolic expressions. The state also contains a path constraint, which expresses constraints on the symbols used in the

symbolic expressions.

Our technique expands the symbolic machine state with timing information. Our machine state contains a symbolic program execution time expression, symbolic time expressions for registers and memory availability, and a symbolic expression about the possibility to dual-issue instructions. An extension of the symbolic execution engine can leverage the timing model to update the timing information in the machine state for each symbolically executed instruction.

Memory access time and branching time are influenced by external factors, i.e., it depends on whether accessed memory is located in the cache and whether the branch-predictor properly predicts the next instruction address. To model timing behavior of those instructions, we introduce a new symbol whenever we encounter one of these instructions. This symbol internally determines whether maximum or minimum execution time is used. This symbol is kept unconstrained to model the external influence, which means the symbolic solver can give multiple values to the time expressions of these instructions.

6.2 Self-Composition Proofs

To prove non-interference and violations thereof, we rely on self-composition. Self-composition is a technique used to analyze properties over multiple program runs of the same program, by transforming the problem to a problem over a single, larger domain, which is a composition of one program run, with a copy of itself, and a connector which describes their relationship. In our scenario, we chose a connector which implies non-interference if it can be proven. Using self-composition to prove the property of non-interference, the problem is reduced to proving a partial correctness property which can generally be solved by off-the-shelf solvers.

6.2.1 Self-Composition

Consider R an abstract set of all the named properties of a single program run. In the combination of symbolic execution and self-composition, this is classically the set of symbolic expressions that express the values of register and memory content, as well as the path constraints associated with the program run. Consider also a well-defined proof obligation function F, that takes a set of properties, and determines whether they are logically coherent, i.e., they do not contradict each other. Thus, $F(R)$ is true if R is a feasible program run. Note that logical coherence of a set of properties requires that each subset of those properties are also logically coherent, mathematically expressed as:

$$\forall X \subset Y : \; F(Y) \implies F(X) \tag{6.1}$$

One can create R', a copy of R, where all properties ϕ are renamed ϕ', which can be seen as the properties of a parallel program run. Because the relations between properties in R' are simply renamed, but still have the same structure, parallel here means the copied program run took the same execution path as

the original run, i.e., the same branches were taken. Because logical coherence cares only about the relations between properties, and not the names of the properties per se, the following holds:

$$F(R) \implies F(R')$$

A union $R \cup R'$ contains all properties of both runs, so that $F(R \cup R')$ determines whether both runs are feasible. Since R and R' are disjunct due to renaming, we have:

$$F(R) \implies F(R \cup R')$$

To create the self-composition C, a connector c is required which relates the properties of R to those of R', e.g., $c = (input\_x \neq input\_x')$. Then, the self-composition is created as:

$$C = R \cup R' \cup \{c\}$$

Because the connector states a relation between R and R', C does not contain two disjunct sets of properties, like $R \cup R'$ does, but instead forms a relationship over the two sets of properties. The self-composition proof obligation $F(C)$ determines whether the two program runs can hold the relation described by c without contradicting itself. Classical non-interference can be proved by self-composition if the connector states that *if all public inputs between runs are equal, the public outputs between runs cannot be different*, i.e., the secret input cannot influence public output.

In our scenario, we are specifically concerned with the output of execution time. In other words, the secret cannot influence execution time, or *if all public inputs between runs are equal, the execution time between runs cannot be different*. For brevity, we explain the proof obligation with a single public input p, a single secret input s, and a single public output time t. However, the logic can easily be extended to any number of variables, by replacing p with P, an ordered list of all public inputs, and likewise for s. R is a program state at the end of a program run, and it contains constraints on p, s, and a relation between p, s, and t. R' is the renamed duplicate of the same state with p, s, and t renamed to p', s', and t'. To prove non-interference, we construct the following connector:

$$p = p' \implies t = t' \tag{6.2}$$

6.2.2 Instruction-Level Timing Channel Identification

Baliu et al. [10] also applied self-composition to prove non-interference of symbolically executed code. However, as described in section 2.4, they define a notion of observations, and apply self-composition proofs on program traces between these points. In contrast, we leave the notion of observations abstract and consider timed program traces in general. Therefore, we can apply self-composition proof obligations on the time that each instruction adds to the

total execution time. We call this Δtime, according to definition 4. By proving non-interference in Δtime, we can identify timing channels *during* symbolic execution, and do not require analysis to first run to completion or to an observation point.

Δtime is the time that individual instructions add to the total execution time of a program.

Definition 4: Δtime

To prove non-interference for Δtime, we create a connector by substitution of Δtime for t in equation 6.2. Thus, we solve the following proof obligation after each instruction:

$$F(R \cup R' \cup \{p = p' \implies \Delta\text{time} = \Delta\text{time}'\}) \tag{6.3}$$

As mentioned in section 6.1, execution time for memory-access instructions and conditional branches depend on external system factors, i.e., cache and branch-prediction behavior, timing of these instructions is modeled by introducing a new symbol in the corresponding Δtime expression. However, non-interference for these instructions does not depend on the Δtime expression directly. For memory-accesses, the information that leaks through the timing channel is that of the memory address. Thus, for memory instructions we perform non-interference on the memory address expression. For conditional branch instructions, the timing channel leaks whether the *condition* is satisfied. Thus, for branch instructions, non-interference proofs are performed for the condition expression on the condition flags. This means that next to a proof obligation with a connector over Δtime, a proof obligation with a connector over these expressions is also used. From here on, whenever we talk about non-interference proofs of Δtime, we also mean non-interference proofs of these expressions where applicable.

The symbol introduced to model the external influence of the cache and branch-predictor is said to be a *public* symbol for self-composition. This models an attacker who can read any such timing differences. If it would not be regarded as a public symbol, self-composition proofs with a connector over Δtime would *always* identify a possible timing difference, even if all publics are kept equal, because the external influence symbol is unconstrained.

Furthermore, because we are not interested in non-interference over symbolic expressions contained in registers and memory addresses, we do not need to model these. Instead, we limit R, the program state, purely to the *path constraints* recorded in the state. We discard all other state properties.

R, the program state used in self-composition, is limited purely to the *path constraints* recorded in the state.

Definition 5: Program state

6.2.3 Pure Control-Flow Timing Channel Identification

Non-interference proof obligations over Δtime fail to capture a very specific type of control-flow timing channels, caused by conditional branches which are unaffected by the branch-predictor. Recall that control-flow timing channels display timing differences because of differences in executed path length. Whenever the branch instruction that determines control-flow of these paths is affected by the branch-predictor, as is often the case, a timing-channel caused by the branch-predictor can be identified if non-interference does not hold. However, if the branch instruction is unaffected by the branch-predictor, the timing-channel is not automatically identified as such.

To account for this, and be able to identify control-flow timing channels as traditionally analyzed by methods based on PC-security, self-composition needs to be applied between the final states of the different program paths. This is a post-processing step which can be applied to catch these timing channels in hind-sight.

Alternatively, one can model the timing behavior of all branch-instructions as insecure, even if they are not affected by branch prediction and if the program paths take equal amounts of time. Onur Acıiçmez [5] has proven that the timing differences of the instruction-cache can form a timing channel, and this class of attacks would be covered by this model. The advantage of this approach is that, like the identification of other timing channels, identification takes place before symbolic execution of an entire program path is finished.

6.3 Analysis

The approach described above takes a binary, a entry point to start execution, and description of public and secret parameters. Possibly, the initial state can be initialized as wished by the analyst, e.g., to put certain values in memory or to call the program with certain concrete parameters.

During symbolic execution of the binary code, the approach can report on timing-channels based on non-interference proofs over the self-composition of the program states. Furthermore, data can be extracted from the symbolic states at these points to help with further analysis of the identified timing channels. Two things we have found very helpful, are the address of the violating instruction, so the violating code section can be inspected, and the symbolic expression of Δtime. This expression can often show a simple relation between the secret and the time, or it may be more complex, in which case further analysis with a constraint solver can lead to a better understanding of the relationship.

After symbolic execution of a program path, the total execution time of the path can be computed. The dynamic range can be computed at this point to determine the size of timing differences.

6.4 Example

We illustrate our technique using example 7. This example implements an S-box lookup function as is commonly used in many cryptographic functions. An S-box is a table of different values, which maps inputs to outputs in a non-linear way. We leave the content of the S-box abstract, by using it as an input to the function. The function takes a data byte and a key byte, and uses a bitwise exclusive or of the two as an index into the S-box. Note that our technique only operates on the assembly code, the pseudo-code is only supplied for understanding of the function.

```
<sbox_lookup>:
EOR r3, r1, r2;
LDRB r0, [r0, +r3];
B r14;
```

(a) ARM Assembly

```
byte sbox_lookup(byte[] sbox,
byte data, byte key) {
  index = data^key;
  return sbox[index];
}
```

(b) Pseudo-code

Example 7: S-box lookup function

First, we briefly discuss the assembly code of the example. The top line, "`<sbox_lookup>:`", is a label which refers to the entry address of the function. The function can be called by a branch to this label. Function parameters are passed in registers `r0`, `r1` and `r2`. The line "`EOR r3, r1, r2;`" corresponds to the pseudo-code line "`index = data^key;`", in other words, it performs the exclusive or of the data and key bytes, and stores the result in register `r3`. The line "`LDRB r0, [r0, +r3];`" corresponds to the pseudo-code line "`return sbox[index];`". It loads a single byte from memory, from the address `r0` + `r3`. `r0` corresponds to the function parameter "`byte[] sbox`": a pointer to the S-box array. `r3` is the just computed S-box entry, and by adding it to `r0` the memory address of the entry is acquired. After loading the S-box entry it is stored in register `r0`, thus overwriting the pointer to the S-box array. Finally, line "`B r14;`" ends the function by branching back to the address contained in register `r14`, also known as the link register, in which the program stored the address from which the function was called.

Our technique is applied on this example as follows. First, one needs to determine which parameters are secret. Because both the plaintext data and keymaterial of a cryptographic function should remain secret, the technique should be instructed that the second and third parameters are secret. The first parameter, the S-box pointer, is not confidential, so it is public. To increase human interpretability of the results, we call the symbols of the function parameters `*sbox`, `data` and `key`. The start of the function is set as the entry point of the binary code, this sets the program counter register `pc` to the starting address. The total execution time of the initial state is initialized to zero. Except for the symbolic content of registers `r1` to `r3`, and `pc`, and the total execution time, the initial state is uninitialized. The overhead for cache misses

is configured at 100 cycles.

After initialization of the technique, symbolic execution is started. The execution engine first processes the first line of code, "`EOR r3, r1, r2;`". The timing module references the timing model for the function `EOR <rd>, <rn>, <rm>`, and finds that it has 1 cycle of issue time and 1 cycle of latency. The initial state is blank in regards to timing availability of registers, so instruction operands `r1` and `r2` are assumed to be immediately available at this point, so no pipeline bubble is modeled. This means Δtime equals the issue time.

$$\Delta\text{time} = 1$$

A self-composition proof obligation is formed to verify non-interference. The path constraints have not been influenced, so R is empty. Thus, the proof obligation according to equation 6.3 is:

$$F(R \cup R' \cup \{p = p' \implies \Delta\text{time} = \Delta\text{time}'\}) \;=\; F(\{p = p' \implies 1 = 1\})$$

Since $1 = 1$ is a tautology, i.e., it is always true, the implication $p = p' \implies 1 = 1$ is always true, and F is thus satisfied. Thus, non-interference holds and there is no timing channel in this instruction. The time of availability of the result register `r3` is updated to current total execution time plus the latency of the instruction; this sets the time of availability to 1. The total execution time of the state is incremented by 1. The instruction is then functionally executed so that the symbolic content of register `r3` in the program state is `data` $\oplus$ `key`.

The execution engine then processes the following code line "`LDRB r0, [r0, +r3];`". The timing module references the timing model for the function `LDRB <rt>, [<rn> ,+/-<rm>]`, and finds that it has a minimum of 1 cycle issue time. Because it is a memory instruction however, a new symbol, `cache_miss_0` is introduced to express the external timing influence of the cache. Δtime is now a symbolic expression:

$$\Delta\text{time} = (\text{if } (\texttt{cache\_miss\_0} \;= 1) \text{ then } 100 \text{ else } 1)$$

The timing module also identifies `r3` and `r0` as early operands, i.e., they are required one cycle early. However, the current total execution time is 1 cycle, and `r3` is only available from this same cycle. Thus, a pipeline bubble of one cycle is formed because `r3` is not available one cycle early. Δtime is updated accordingly.

$$\Delta\text{time} = 1 + (\text{if } (\texttt{cache\_miss\_0} \;= 1) \text{ then } 100 \text{ else } 1)$$

Because `LDRB` is a memory instruction, two proof obligations are formed, one with a connector over Δtime, and one with a connector over the accessed memory address. The path constraint R is still empty. The first proof obligation, over Δtime, according to equation 6.3 is:

$$F(\{{}^{*}\texttt{sbox} = {}^{*}\texttt{sbox}' \wedge \texttt{cache\_miss\_0} = \texttt{cache\_miss\_0}'\}$$
$$\Longrightarrow$$
$$1 + (\text{if } (\texttt{cache\_miss\_0} \ = 1) \text{ then } 100 \text{ else } 1))$$
$$=$$
$$1 + (\text{if } (\texttt{cache\_miss\_0}' \ = 1) \text{ then } 100 \text{ else } 1)))$$

Since `cache_miss_0` and `cache_miss_0'` are the only symbols in the Δtime expression, and these are considered public values which are constrained to the same value, Δtime is always equal to Δtime$'$. Thus, F always holds true.

The second proof obligation is formed over the accessed memory address as follows:

$$F(R \cup R' \cup \{p = p' \implies \text{address} = \text{address}'\})$$

Because R is still empty, we get the following proof obligation if we substitute the symbolic address for address and address':

$$F(\{{}^{*}\texttt{sbox} = {}^{*}\texttt{sbox}' \implies \texttt{data} \oplus \texttt{key} = \texttt{data}' \oplus \texttt{key}'\})$$

Clearly, the implication in this proof obligation is false, as the left hand of the implication is unrelated to the right hand. Because F does not hold, a cache timing channel is identified that leaks the accessed memory address. The execution engine can warn about this address, return the program state, and identify the leaked relation `data` $\oplus$ `key` as a symbolic expression.

Subsequently, availability of the result register `r0`, as well as the total execution time, are updated with Δtime. Both now hold the following symbolic expression:

$$2 + (\text{if } (\texttt{cache\_miss\_0} \ == 1) \text{ then } 100 \text{ else } 1))$$

The instruction is then functionally executed so that the symbolic content of register `r0` in the program state is updated by the memory content. Because the S-box was left uninitialized, this can be a symbolic read, modeled by a new symbol `unconstrained_read_0`.

Finally, the last instruction `B r14;` is executed. The timing module references the timing model and sets Δtime to 1, for which no timing channel is identified by the self-composition proof. Total execution time is updated by 1 cycle, and symbolic execution finishes.

Only one code path has been identified, so no post processing self-composition proofs are required to identify pure control-flow timing channels, because they are not possible if only one code path exists. Total execution time is identified as 4 or 103, depending on the cache behavior of the one identified memory instruction. This corresponds to a dynamic range of 14 dB, according to the approach discussed in section 2.2.

6.5 Discussion

Our approach does not rely on specific features of the timing model, as long as a corresponding implementation to process it symbolically is available. This means the technique can be applied to timing models of any level of detail. Models can even be simple extensions of the PC-model, in which only certain specifically interesting instructions are modeled differently than the PC-model, and all other instructions simply have 1 cycle issue time and no latency. This allows to quickly create test environments to identify specific problems only.

The technique does depend on manual processing in two respects. Firstly, the state needs to be initialized and the technique instructed on public and secret symbols. Secondly, manual interpretation of the results and leaked information is required to determine whether timing channels are actually present. These required steps prevent the technique from analyzing programs completely independently.

Chapter 7

SMArTCAT

In this section we present SMArTCAT: Symbolically Modeled Architecture Timing Channel Analysis Tool, which implements our technique as described in chapter 6. SMArTCAT is built on top of the angr [42] symbolic execution framework, and leverages the timing model for the ARM Cortex-A7 as described in chapter 5. The tool is meant as an aid for security analysis to determine whether, and how, secret parameters influence program execution time.

7.1 angr

SMArTCAT is implemented as a Python package working on top of angr. To properly describe how SMArTCAT works, we briefly explain how angr works. angr is a framework for symbolic execution, that is, it can load binaries, execute them using symbolic values instead of concrete values, and return program results as symbolic expressions.

angr does not operate on binaries directly, instead, it lifts them to an intermediate language called VEX. The lifter uses the Capstone disassembler framework, which can disassemble binaries for multiple machines. Because the lifter can also handle these different languages, angr can operate on binaries of multiple machine types too. However, SMArTCAT is limited to the Cortex-A7 because we only created a detailed timing-model for this architecture.

The VEX instructions are interpreted by angr, operating on symbolic expressions. The program state, i.e., register and memory contents – in the form of symbolic expressions – are maintained as parts of a symbolic state. States contain a symbolic solver instance, which keeps track of path constraints, in the form of symbolic expressions, that apply to the symbolic expressions in this state, e.g., "input_x $>$ 5". Whenever execution paths branch, the state is duplicated and constraints are added to each of those states corresponding to the branch conditions. Concrete values for the symbolic expressions can be generated by the Z3 solver back-end, and the solver can also be used to determine whether constraints are satisfiable; which is used for example to determine

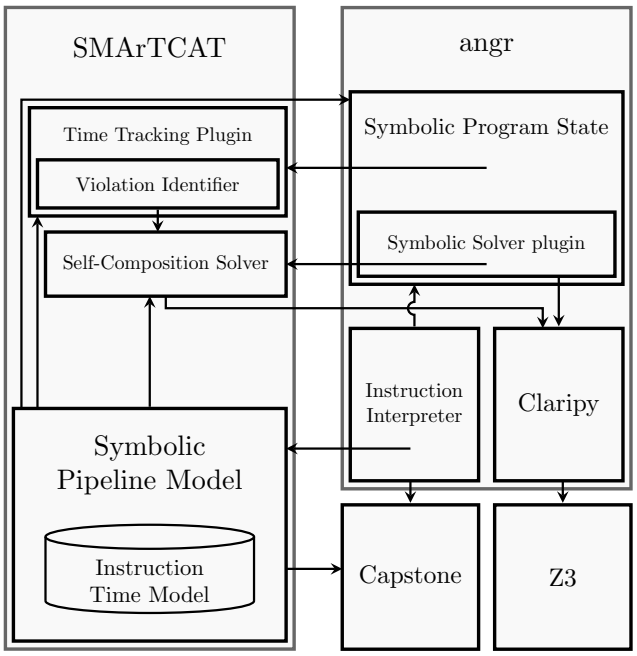

Figure 7.1: SMArTCAT components and interplay

whether both sides of a branch condition are feasible. In angr, Z3 is accessible through Claripy, which provides a python front-end and has multiple optimizations implemented.

angr is not a tool, i.e., it does not perform any analysis itself. Instead, it is a framework which provides symbolic execution. However, as a framework, it can be used to build tools on top of it. This allows us to leverage the symbolic execution and detailed state information during any point of execution to implement our own analysis.

7.2 SMArTCAT Design

The main components of SMArTCAT and angr are displayed in figure 7.1. The arrows demonstrate which components initialize communication with other components. The Symbolic Solver Plugin and Instruction Interpreter of angr have been *monkey-patched* to call SMArTCAT functions, i.e., the original functions have been overwritten with patched code, so that SMArTCAT functions gets called during symbolic execution. The Time Tracker Plugin uses the Symbolic Program State plugin functionality to attach itself to angr.

The Instruction Time Model takes Capstone instruction instances and returns their properties relevant for timing. This includes issue time and latency,

and all properties described in chapter 5, such as register bypasses, early and late registers, and dual issue capabilities. Based on the instruction properties and the state of the program, the Symbolic Pipeline Module symbolically computes Δtime and determines how it is influenced. That is, it computes whether instructions dual issue, bypasses take place, pipeline bubbles form, etc. For each instruction, it queries the Self-Composition Solver whether non-interference holds for Δtime, and then updates the time state and availability of registers and memory locations in the time tracker plug-in. Recall that we consider an attacker who is able to read timed program traces. If non-interference holds for Δtime of each instruction, and no control-flow based timing channels are present, we consider the program secure under our attacker-model and timing model.

The Time Tracker symbolically keeps track of execution time, expressing execution time as a function of program inputs. The Time Tracker can use one of several concretization strategies to optionally concretize execution time up to the point of symbolic execution. Although concretization poses some limits on the completeness of the analysis, this can significantly speed up analysis of programs with complex timing behavior. If concretization is applied, results are an estimate of execution time only. For example, cache hits and misses and branch miss-prediction could be left unmodeled. However, as concretization is applied after timing channel identification, this should not negatively affect timing channel identification, except for pure-control flow based timing channels in the scenario that the instruction cache is not taken into account, as described in section 6.2.3. The Time Tracker is written as a plug-in for the angr program state, so that when the program path branches, it gets automatically duplicated along with the other state information.

Symbolic execution and exploration of program paths depends fully on the angr framework. SMArTCAT users can instruct angr to use different exploration techniques or other functionality offered by angr. However, we make no guarantees about the analysis results as we have not looked into the effects of different combinations of settings beyond the default exploration of all possible paths.

7.2.1 Self-Composition Implementation

Section 6.2 describes the self-composition technique that our tool uses to prove non-interference of instructions. Recall that the technique works on a combination of a path's constraints, a version of the same constraints over renamed symbols, and a connector which relates the renamed symbols to the original symbols. In this section we describe how the technique is implemented in SMArTCAT, and which optimisations have been applied to the implementation.

In angr, a program's path constraints are expressed as symbolic expressions that can be solved by Claripy, the front-end to Z3. If the constraints are coherent, the expression is said to be satisfiable, and Claripy can attempt to generate solutions for symbols which adhere to the expressions. However, Claripy does not have an implementation of the implication relation " $\Longrightarrow$ ". According to

classical logic, $a \implies b$ is the same as $\neg a \vee b$. However, a naive translation of the connector $p = p' \implies t = t'$ to the constraint $p \neq p' \vee t = t'$ does not express the relation which needs to be proven for non-interference to hold. Claripy will search for a solution where either $p \neq p'$ or $t = t'$ is true, but it does not tell whether $p \neq p' \vee t = t'$ is true *for all possible values* of p and t.

Instead, we create a constraint c which is the inverse of the logical formula of the required connector. To determine coherency, we query the solver for *unsatisfiability*. This works because if the solver cannot find a solution to the constraint, this implies that for all possible values the constraint does not hold; thus the inverse of the constraint holds for all possible values. The connector constraint we use for the self-composition is then as follows:

$$c = \neg(p = p' \implies t = t') = (p = p' \wedge t \neq t')$$

The self composition C is thus :

$$R \cup R' \cup \{p = p' \wedge t \neq t'\}$$

Recall that t is Δtime, except when determining non-interference of memory addresses, branch conditions, and complete program paths.

Self-Composition Optimizations

SMArTCAT makes heavy use of self-composition because non-interference proof obligations are formed for every executed instruction. Multiple optimizations have been implemented to speed up this part of the analysis.

Optimization 1 is motivated by the significant overhead that would be generated if all the relevant constraints had to be duplicated and renamed at every self-composition proof-obligation that is constructed. Instead, SMArTCAT constantly keep track of all constraints added to the state, and immediately duplicate and rename the constraints, thus keeping track of a virtual parallel execution path over which self-composition properties can be derived with much less overhead per self-composition proof. For the actual self-composition, both sets of constraints only need to be merged and connected, without any renaming.

Optimization 2 is one described by Naumann et al. [34]. The keen reader may have noticed an interesting aspect of self-composed states: they are constructed from a set of constraints, over several symbols, including public inputs p, and their copies p', which are connected by a constraint $p = p' \wedge t \neq t'$. However, since the connector constrains all public copies p' to the same value as the original p, there is no actual benefit to duplication and renaming of p. Instead, as described by Naumann et al., one can create an integrated self-composition where only unequal symbols are copied and renamed. This creates a smaller self-composition state which is logically equivalent to the original one, but is significantly easier to prove for the solver. The constraint $t \neq t'$ then suffices as a connector, instead of the original constraint $p = p' \wedge t \neq t'$.

For example, if the constraint set R is $\{5p > 10,\ t = 2s + p\}$, then the partially renamed copy R' is $\{5p > 10,\ t' = 2s' + p\}$. The full self-composition constraint set $R \cup R' \cup \{t \neq t'\}$ is then $\{5p > 10,\ t = 2s + p,\ t' = 2s' + p,\ t \neq t'\}$.

Optimization 3 breaks up the self-composition proof-obligation into multiple smaller proof-obligations before solving the full proof-obligation. This is motivated by the observation that non-interference holds for most instructions, and if the workload for these instructions can be reduced, the overall execution time will improve. Note that the self-composition proofs are used to determine non-interference in specific time expressions, which are well-defined expressions over input symbols. Due to equation 6.1 and the well-definedness of F, we can state that if $F(X)$ holds for all $X \subset Y$, $F(Y)$ also holds:

$$(\forall X \subset Y : F(X)) \implies F(Y)$$

Furthermore, for any well-defined function f and any symbols σ and x, we have:

$$F(\{\sigma = x\}) \implies F(\{f(\sigma) = f(x)\})$$

Let t be a well-defined time expression, and let I be the set of all input symbols over which the expression t is formed. Then, based on the previous properties of F:

$$(\forall \sigma \in I : F(R \cup R' \cup \{p = p' \implies \sigma = \sigma'\})) \implies F(R \cup R' \cup \{p = p' \implies t = t'\})$$

This means that if non-interference holds for each of the symbols in the expression, it must also hold for the entire expression. Thus, we can save time by first determining for which symbols in the expression t non-interference might possibly not hold, followed by a step of actual non-interference proofs for each of these symbols. Only if any symbol is found for which non-interference does not hold, a non-interference proof obligation of the entire expression is constructed and evaluated. This significantly reduces time for self-composition proofs for most instructions, for which non-interference holds.

7.3 Evaluation & Discussion

We have tested SMArTCAT on several toy examples as well as real cryptographic functions. The results of the case studies on real cryptographic functions are described in chapter 8. The toy examples proved that SMArTCAT can identify all types of timing channels and identify relations between secret inputs and the timing channel. Execution time of one of the toy examples, a ten instruction linear feedback shift register function, was measured on the Raspberry Pi 2. The measured time matched exactly with the time predicted by SMArTCAT. Complexity of measuring larger programs accurately has prevented us from comparing execution time of large functions with SMArTCAT predictions.

SMArTCAT suffers from multiple limitations imposed on it by the underlying techniques. As discussed earlier, the used timing model is very specific and therefore makes the tool only applicable to very specific platforms. This is a trade-of with the accuracy of security it can guarantee. Any tool can only be as accurate and widely-applicable as the timing-model on which it depends. Other timing models could be implemented for the tool, but for each new platform this requires a significant one-time investment.

SMArTCAT suffers from the same limitations as angr and symbolic execution in general. Firstly, symbolic execution is slow compared to native execution on the target platform. Most of our analyses were run on an Intel Core i7-4800MQ running at 2.70 GHz, on which we symbolically executed 5-30 instructions per second. This speed puts a significant limitation on test program size.

We compared this speed to that of analysis based on the PC-model. Instead of symbolically modeling the pipeline, SMArTCAT was instructed to only count executed instructions, for several real cryptographic functions. This comparison shows that our approach is only approximately 20% slower than symbolic timing-channel identification based on the PC-model.

The literature describes multiple techniques with which symbolic execution speed can be increased. One of the most promising techniques of the moment is called concolic execution [40], which combines concrete and symbolic execution to increase program coverage depth. A significant limitation is that the symbolic relations between secret input and expressions are lost by this technique. A middle-ground could be to only concretize public values. We leave it for future work to determine in which situations this can be beneficial.

One obstacle for symbolic execution is state-explosion, in which program loops cause a significant number of possible execution paths which all need to be executed by the symbolic execution framework. This problem is especially significant for loops of a variable number of iterations. Due to the nature of the specific program instances in which we are most interested, i.e., cryptographic libraries, this does not pose a large obstacle for our work, because most cryptographic implementations do not heavily rely on loops.

SMArTCAT does not directly tackle the problem of timing channels in multi-threaded programs, because angr cannot perform analysis which takes multi-threading into account. We are not aware of any cryptographic implementations which rely on multi-threading, so this is not an actual problem for the program instances in which we are most interested.

Identification of timing channels with SMArTCAT relies heavily on non-interference proofs solved by Z3 through Claripy. However, we have found that Claripy may not find a solution for certain problems, and it can even return false results under certain circumstances. We are unsure about the impact of Claripy's limitations. However, it is always advised to validate results returned by SMArTCAT. Claripy may also not halt on certain complex problems within practical time. However, we note that symbolic expressions in cryptographic implementations generally become complex only when they depend on data or key material. Thus, when Claripy causes SMArTCAT to hang, it always warrants

manual inspection for timing channels at the instruction it hangs at. We have found that this happens often when analyzing cryptographic functions, at various execution depths, but this has not prevented SMArTCAT from producing usable results.

Chapter 8

Case Studies

Like most automatic timing channel identification tools described in the literature, the most important form of validation we applied is by case studies on cryptographic libraries. In this chapter we describe the cryptographic implementations we tested and our findings. All tested binaries were compiled for the Cortex-A7 using GCC (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609

8.1 OpenSSL

OpenSSL [4] is a generic open-source cryptographic library. Multiple timing channels have been identified in OpenSSL before [5, 12, 15]. Thus, we assume that countermeasures against timing channels have not been a priority during initial OpenSSL development. We consider it an interesting target to see whether those findings have led to a timing channel–resistant implementation. We have tested several cryptographic functions of OpenSSL 1.1.0e. Specifically, we have focused on encryption and key-setting functions. We have considered both plaintext and key material as sensitive material which should not leak through timing channels. For each function, we only discuss the first identified vulnerability. For the implementation details of all functions, we refer to the source code [4].

On a side-note, key expansion algorithms are very unlikely functions to be attacked, because they are generally executed only once, after which the result is stored. This prevents an attacker from taking multiple measurements in most situations; we only analyzed these functions for completeness.

8.1.1 Blowfish

Blowfish [39] is a multi-round cryptographic cipher, i.e., it uses a cryptographic function F, which is applied multiple times to achieve the final result of the cipher. The round function F of Blowfish is based on four different S-box

lookups. Recall from section 6.4 that an S-box maps different inputs to different outputs in a non-linear way. The S-box lookups of Blowfish are based on the exclusive or function of output from the previous round, and a round key. This result is split into four bytes which are used as input to four different S-boxes, with four byte output each. The S-box results are combined to produce the result of the round function F. Blowfish is a *Feistel* cipher, which means that each round, half the round-input is used as input to F, and the result of F is combined with the other half of the round-input using an exclusive or function.

In 1998 it was already hypothesized by Kelsey et al. [26] that the design of Blowfish could lead to timing channels due to the large S-boxes it uses. We have tested the Blowfish encryption as implemented in the function `bf_encrypt`. SMArTCAT quickly identifies a timing vulnerability in instruction `ldr r5, [r3, r14, lsl #2]`. Inspection of the source code files bf_enc.c and bf_locl.h clearly shows the S-box function implemented as an array lookup. We reverse engineered the binary code to confirm that the violating `ldr` instruction corresponds to the array access.

SMArTCAT identifies the symbolic expressions which correspond to the values for `r3` and `r14`, which determine the memory access location. One of the expressions includes an exclusive or of a key byte with a plaintext byte, which causes SMArTCAT to identify the timing channel. Line 34 of bf_enc.c – `l ^= p[0];` – clearly determines the first S-box entry, and confirms the identified exclusive or relation between public and secret.

The vulnerability in this implementation of Blowfish is clearly caused by the implementation of the S-box function as an array lookup. This is believed to be the fastest way of implementing S-boxes, and is consequently a very common way to implement them. Unfortunately, array lookups always cause type 2 violations if the lookup depends on a secret value. All S-box lookups in this function are implemented in this way, so all S-box lookups can theoretically be learned by an attacker.

Attackers can learn the accessed memory address from cache-based timing channels, and can thus learn the position of the accessed array element. In the scenario of Blowfish, this means an attacker can learn $\text{key}_0 \oplus \text{data}_0$, where key_0 is the first round key and data_0 the first part plaintext, each 4 bytes long, using four cache-attacks. We propose the following timing attack to demonstrate vulnerability of this implementation. It is a multi-stage attack which has a known-plaintext and chosen-plaintext part.

1. The attacker performs a known-plaintext cache-attack and learns $\text{key}_0 \oplus \text{data}_0$ from the first round. Since the plaintext data_0 is known, key_0 is trivially disclosed.

2. The attacker performs another timing-attack, snooping on encryption of a victim's secret data. He performs this attack on the first two encryption rounds, thus learning the first two S-box lookups:

$$\text{key}_0 \oplus \text{secretdata}_0$$

and:

$$F(\text{key}_0 \oplus \text{secretdata}_0) \oplus \text{key}_1 \oplus \text{secretdata}_1$$

Since key_0 is already known, secretdata_0 is trivially disclosed from the first address.

3. The attacker can now perform a chosen plaintext timing-attack where he chooses $\text{data}_0 = \text{secretdata}_0$. By attacking the second round, he learns

$$F(\text{key}_0 \oplus \text{secretdata}_0) \oplus \text{key}_1 \oplus \text{data}_1$$

Since he chose data_1 himself, this trivially leads to:

$$F(\text{key}_0 \oplus \text{secretdata}_0) \oplus \text{key}_1$$

From this and the data acquired in stage 2, secretdata_0 is disclosed.

Since all data encrypted with Blowfish undergoes the same encryption process, the above attack can be used to learn the entire plaintext. It should be noted that this attack relies on a strong assumption; that is, the attacker can perform cache-attacks from which he learns the accessed S-box entries. In practice, it is not trivial to perform such an attack, which is probably why these kind of vulnerabilities are not being exploited on a large scale in the wild.

One of the challenges in actually learning an accessed register location from a cache-access is the granularity of the cache, i.e., the number of array items stored on one cache line. However, in the case of Blowfish, S-box entries are relatively large, i.e., 4 bytes, and cache line sizes for the Cortex-A7 are relatively small, i.e., 64 bytes. This means deducing the actually accessed array index from an identified cache-miss is easier than it is for algorithms which use S-boxes with smaller entries.

We leave it for future work to actually perform the described attack on Blowfish. However, based on related work on practical cache timing attacks [12, 35], we are confident that a determined attacker can exploit a cache timing channel such as this one under certain circumstances.

Blowfish S-boxes are constructed dynamically from key-material, and this prevents the S-boxes from straightforwardly being implemented as a system of equations. One high-level approach to implement a constant-time lookup for any S-box is by a walk over all entries, such as in algorithm 1 for an 8-bit S-box. Memory accesses in this algorithm are independent of the used S-box index, as all S-box entries are accessed. First, a bitmask m is computed by collapsing all bits in i - index to a single bit, and expanding that to 0xff only if i - index = 0. This is a constant time version of "m = (i - index == 0)?0xff:0". Thus, only when the walk over all S-box indices arrives at the correct index, it adds the S-box entry into the result.

However, this implementation is rather inefficient, as for each S-box lookup, it requires a number of memory accesses in the order of the size of the S-box, instead of a single memory access in the non-constant time implementation. Instead of relying on an inefficient constant-time implementation of Blowfish, it is strongly advised to use modern algorithms which can be more efficiently implemented in constant time solutions.

```
byte lookup(byte[] sbox, byte index) {
        result = 0;
        for (i=0; i<length(sbox); ++i) {
                unsigned byte m;
                m = i - index;
                m |= (m>>4);
                m |= (m>>2);
                m |= (m>>1);
                m = 0xff * (m^1);
                result += m & sbox[i];
        }
        return result;
}
```

Algorithm 1: Generic constant-time S-box lookup solution

8.1.2 AES

AES [43] is a block-cipher based on a substitution-permutation network. Like Blowfish, it relies on S-boxes to provide a source of non-linearity. The key schedule algorithm, which turns a short key into separate round keys for every round, relies on the same S-box functions. We tested both the encryption and key setting parts of the algorithm, respectively implemented in the functions `aes_encrypt` and `aes_set_encrypt_key`. The AES implementation of OpenSSL 0.9.7a has been found vulnerable before By Bernstein [12]. Constant-time versions of both the cryptographic function as well as the key-expansion function have been implemented by Käsper et al. [24].

Running SMArTCAT on `aes_encrypt`, it identifies a type 2 violation at instruction `ldr r4, [r4, #0xc28]`. The memory address expression identified by SMArTCAT includes an exclusive or of a key byte with a message byte. Source code inspection reveals that, like in the case of Blowfish, the AES S-box has been implemented as an array. Once again, we reverse engineered the binary code to confirm that the violating instruction corresponds to the array access. If an attacker can learn accessed S-box entries, he can mount a known-plaintext attack to recover the key, which he can subsequently use to deduce plaintext bytes from timing attacks in which the attacker snoops at the encryption of a victim's plaintext.

Running SMArTCAT on `aes_set_encrypt_key`, it identifies a type 2 vulnerability at instruction `ldr r10, [r2, #0x28]`. The memory address expression identified by SMArTCAT includes a byte of key material. Thus, an attacker who learns the accessed S-box entry, also learns the key byte. Once again, source code inspection reveals the use of the S-box implemented as an array.

Normally, AES operates on S-boxes with 256 entries of 8 bits each. Given a cache line size of 64 bytes, the entire S-box is spread over just four cache-lines. This means significantly more effort from an attacker is required for

him to deduce the accessed S-box entry from a cache-miss. However, OpenSSL supports a function to increase performance by operating on extra large S-box tables, with 8 *byte* entries. Obviously, significantly less effort is required to attack a system that uses these tables.

8.1.3 Camellia, CAST, 3DES, SEED

OpenSSL contains multiple other algorithms. To prevent this section from becoming too repetitive, we shortly summarize our results for the following algorithms. We tested key setting and encryption functions of the algorithms Camellia [32], CAST [6], 3DES [11] and SEED [30], which are all based on S-boxes. For all these functions SMArTCAT was able to identify type 2 violations. We reverse engineered the library binary, to confirm that all identified timing channels corresponded to S-box lookups. Furthermore, the memory access expressions all show a clear relation between key and plaintext material, and the accessed memory location.

Camellia was already found vulnerable in OpenSSL-1.0.0-beta3 in 2009 by Zhao et al. [46]. As far as we can tell, the S-box implementation has not changed since. CAST has been hypothesized to be vulnerable by Kelsey et al. [26].

8.1.4 RC2 and RC4

Furthermore, we tested RC2 [38] and RC4 [25], which are two algorithms not based on S-boxes. However, these algorithms work with several indexed lookups into the key material itself, effectively turning these lookups into dynamic S-box transformations as is the case with Blowfish. Furthermore, RC2's key expansion operation uses a *Pi-table*, which is affectively an S-box based on the digits of Pi. For both key-setting functions and the RC4 encryption function, SMArTCAT was able to identify type 2 violations. When analyzing the RC2 encryption function, SMArTCAT hung at instruction `ldr r7, [r1, r7, lsl #2]`, which can happen when symbolic expressions become too complex to process automatically. However, with cryptographic algorithms this is an indication that the instruction warrants further inspection.

Comparison of the binary with the source code of the functions confirms that the identified instructions are all related to the array accesses. The memory address expressions identified by SMArTCAT demonstrate a clear relation between key bytes and the address for both the RC4 encryption and key-setting function, and the RC2 key-setting function. The relation has only been significantly obfuscated for the RC2 encryption function, due to the mixing rounds which precede the memory access.

8.2 TweetNaCl

NaCl is a cryptographic library specifically intended to circumvent timing channels. During implementation, type 1 and 2 violations were taken into account

and carefully circumvented. [13] However, the library has been implemented in the C programming language, not in assembly. Its C-code has been validated for timing channels. TweetNaCl [14], also implemented in C, is a version of the library which has been written to fit into a hundred tweets.

TweetNaCl is an interesting case study because of multiple reasons. Firstly, type 3 timing channels were not taken into account during implementation of this library. Secondly, our tool can perform timing channel analysis at a more detailed level than we believe has been used to analyze TweetNaCl before. Lastly, the Cortex-A7, as any other non-x86 processor, was probably not directly kept in mind during implementation of the library. It is interesting to know whether specific timing behavior for this processor could introduce a timing channel in the library.

We tested TweetNaCl version 20140427[1]. Branching behavior between C-code and assembly is very closely related, and we have no indication that compilation for the Cortex-A7 would introduce extra branches based on secret parameters; thus, we have no reason to believe that any possible timing channels exist due to conditional branches. Instead, we focus on type 3 vulnerabilities only. As an initial step we located all conditional instructions in a binary compiled with the TweetNaCl. Secondly, we identified all functions on which they depend, and executed SMArTCAT to analyze those functions. We repeated this process for binaries compiled with the compilation flags Os, Og, and O3; which are intended for minimal code, debugging, and fastest execution respectively.

Analysis was performed until the conditional instructions were executed. SMArTCAT did not identify timing channels for any of the tested functions. This leads us to believe that TweetNaCl is secure with respect to timing channels for the ARM Cortex-A7. Furthermore, it gives an indication that compiler optimizations do not tend to introduce conditional instructions in binaries compiled from code which avoids branches that depend on secret values. However, since type 3 violations may exist for wildly different instructions for other processors, we cannot conclude that TweetNaCl is secure for all processors.

8.3 Discussion

SMArTCAT was able to identify multiple cache-based timing channels in the OpenSSL library. For all identified timing channels it is easy to identify the logical steps required to deduce secret information from an accessed memory-location. Furthermore, SMArTCAT may hang at instructions that create a relationship between secret and time which is too complicated to solve automatically.

Many attacks and hypotheses about vulnerabilities in the algorithms implemented in OpenSSL have been known for a long time. Despite this, constant-time implementations have not been adopted to solve the vulnerabilities. It is clear to us that the library cannot be trusted to perform cryptographic functions in constant-time on arbitrary platforms.

[1]The latest version at the time of writing, released through https://tweetnacl.cr.yp.to

Although theoretical attacks on these algorithms exist, practical exploits may still be burdened by other properties such as S-box size, which influences the detail by which a timing channel leaks information. However, determined attackers have frequently been able to circumvent practical difficulties [12, 37, 46]. To increase the security of ciphers which rely on S-boxes, the S-box functions could be implemented by means of a system of equations, as is done in the AES implementation of TweetNaCl.

Alternatively, the relationship between secrets and accessed S-box entries can be obfuscated, as demonstrated by RC2, which performs a preprocessing step on the secret data, before it is used to determine an array lookup. If this relationship is sufficiently obfuscated, an attacker has a significantly harder time deducing secret information from timed cache accesses. Especially in the case of Feistel ciphers, we do not see a reason to not obfuscate secret data, before it is used as an index to an S-box, as the structure of Feistel ciphers does not demand that the round function is invertible.

Part III

Reflection

Chapter 9

Related Work

There have been multiple automated timing channel analysis techniques described in the literature before which are related to our work. These approaches explore different analysis techniques of have different analysis goals.

Molnar et al. [33] first defined PC-security, and created an automatic technique to verify it on binary code. Their approach is based on fuzz-testing, i.e., they execute a program many times using different program parameters, and analyze the executed code paths of the program. Fuzz-testing is a crude approach and completeness of program coverage can be limited, because there is no full exploration applied as is with symbolic execution. On the other hand, this approach is fully automated which circumvents the requirement of manual initialization.

As described earlier, PC-security is intended for hardware which executes each instruction in the same constant time. As this model cannot be applied to general purpose processors, our approach is more complete because it can identify type 1, 2 and 3 violations, whereas the approach by Molnar et al. can only identify type 1 violations.

As described in section 2.4, Almeida et al. [8] transform self-composed code with ghost-code and annotations such as preconditions and invariants, which allows them to store a notion of execution time within the program states. Subsequently, they apply automatic and interactive verification tools to prove non-interference. The security notion considered by Almeida et al. is an extension of PC-security, which includes data memory access patterns, so they can identify a wider range of type 1 and 2 violations. If non-interference cannot be proven, the verification tools show which code causes the violation.

For programs with complex program loops, this approach suffers from the same problem as symbolic execution, i.e., state explosion. Furthermore, the technique is based on automatic and interactive verification solving. This looks a lot like our approach, except that our technique determines non-interference automatically, and we require interactive behavior only to further analyze the actual information leak. The biggest difference between this approach and ours is that Almeida et al. operate on C source code, and they assume that the

compiler does not violate the security policy. Thus, they cannot identify type 3 violations and the compiler needs to be trusted under their security model.

As described in section 2.4, Balliu et al. [10] apply self-composition in combination with symbolic execution to derive trees of observational states, in which each observational state includes the execution time up to that state. Self-composition in this case is not applied on the program itself, but on the observation trees. The trees are linked with a connector which defines the relation between variables in both trees. A verification function is then applied on both trees to assert that the observational states have equal timing behavior so that they do not leak secret information. As far as we understand, this technique is limited to identify the presence of timing channels, but cannot locate the violating code.

Although both this approach and our approach are based on symbolic execution and self-composition, they are applied in different ways. Balliu et al. apply self-composition on full observation trees. On the other hand, we apply it during execution, so we can identify timing differences on the fly and identify differences in timed program traces instead of on observational states only.

The cost model applied by Balliu et al. equals the program counter model, so timing channel identification is limited to a subset of type 1 violations, whereas our approach can also identify type 2 and type 3 timing channels.

As discussed in section 3, Păsăreanu et al. [36] apply symbolic execution to quantify information leakage of timing channels, and to determine which program inputs lead to maximum information leakage. They open up interesting territory by expanding analysis to information leakage over multiple runs. Furthermore, they automate a large part of post-analysis of identified timing channels by automatically determining program parameters which maximize information leakage. However, their current approach focuses solely on type 1 timing channels, so considering timing channel identification this technique does not cover as wide a range of timing channels as our approach.

Chapter 10

Conclusions

In this work we set out to research the impact of detailed timing models on timing channel analysis. We conclude our work based on multiple pillars: feasibility, cost, completeness, soundness, and practical benefits.

We have demonstrated that it is feasible to construct a detailed timing model, at least for a relatively simple processor such as the Cortex-A7. This was possible despite limited information about the processor's internals, and in the presence of false information about the timing of certain instruction couples.

We have proven the feasibility of symbolic execution techniques for timing channel identification that use detailed timing models, to create an accurate estimation of execution time of binary functions. Practical problems concerning the complexity of symbolic time expressions can partially be overcome by strategical concretization of the symbolic expressions, which is a necessity when executing large cryptographic functions.

The cost that our technique induces on the running time of symbolic analysis seems limited: by our estimations running time is roughly 20% slower than when only counting instructions with symbolic execution. If we factor in the fact that our technique can identify timing channels on the fly, i.e., before the tool has run to the end of analysis, timing channel identification in vulnerable code can often be performed faster using our technique.

Recall from section 2.3 that PC-security was never intended as a security scheme for general purpose processors. Instead, it is intended for a processor which executes all instructions in the same time. Thus, it is clear that for general purpose processors, our approach can identify more timing channels than approaches which rely solely on PC-security. Our approach is thus more complete for this type of processors.

Furthermore, approaches that use the program counter model may identify timing channels based on a number of executed instructions in different paths, whereas both paths may actually require the same number of cycles to execute, due to differences in instruction execution time. In reality, these cases are likely very limited, and we conclude that our approach offers only marginally more sound results for general purpose computers.

An important facet of any approach is the practical benefit it offers. However, as what is practical may be highly subjective, this can be one of the hardest facets to clearly define. On one hand, our case studies have demonstrated the approach; however, only type 2 vulnerabilities have been identified in the cryptographic functions which we analyzed, and type 2 vulnerabilities do not require a detailed analysis like we used. On the other hand, it proves that many cryptographic functions do not suffer from type 3 timing channels when compiled for the Cortex-A7 core instruction set.

We consider our work as a proof that automated timing channel analysis based on detailed timing models is feasible. It can improve analysis results without substantial overhead. Future work must demonstrate the relevance for other processors and code bases.

Chapter 11

Future Work

Guided in part by timing channels identified by Andrysco et al. [9], this work makes a first step towards improved accuracy of timing channel analysis. However, there is still significant work to be performed in this field to ultimately determine its relevance.

Firstly, our work has focused very specifically on a subset of instructions of the Cortex-A7. Besides conditional instructions, no other causes of operand-dependent instruction execution time have been identified. If we want to improve our grip on timing channels caused by this behavior, it is important to improve our understanding of the relationship between instruction operands and execution time, for different modern processors. It would greatly benefit the applicability of tools like SMArTCAT if accurate timing models are constructed for multiple processors.

Secondly, the relationship between high-level code patterns and instructions which exhibit operand-dependent timing is not always clear. Our case studies have once again confirmed the relation between naive implementations of S-boxes and cache-based timing channels. However, as we have not identified any type 3 timing channels, it leaves to wonder whether they are actually common in cryptographic software. Obviously, this fully depends on the types of instructions which are identified to exhibit operand-dependent timing behavior for different processors.

Lastly, our analysis technique is limited by several challenges that generally limit symbolic execution, such as the complexity of solving symbolic expressions, as well as complicated initialization of program states and function parameters. Our approach would greatly benefit from steps made in these fields which alleviate these challenges. Specifically, if an approach could be developed to automatically initialize program states and estimate which parameters should remain secret, our approach could operate automatically without manual initialization. This would be a big step towards intelligent automated software testing for timing channels.

Bibliography

[1] ARM architecture reference manual ARMv7-A and ARMv7-R edition, issue c. infocenter.arm .com/help/index.jsp?topic=/com.arm.doc.ddi0406c Accessed: March 2017.

[2] ARM Cortex-A7 pipeline description. gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/config/arm/cortex-a7.md;hb=HEAD Accessed: March 2017.

[3] Cortex-A8 technical reference manual revision r3p2. infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf Accessed: March 2017.

[4] OpenSSL: Cryptography and SSL/TLS toolkit. www.openssl.org. Accessed: March 2017.

[5] O. Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[6] C. Adams. The CAST-128 encryption algorithm. 1997.

[7] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53. ACM, 2000.

[8] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796–812, 2013.

[9] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[10] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1080–1091. ACM, 2014.

[11] W. C. Barker and E. B. Barker. Sp 800-67 rev. 1. recommendation for the triple data encryption algorithm (TDEA) block cipher. 2012.

[12] D. J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[13] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.

[14] D. J. Bernstein, B. Van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer, 2014.

[15] B. B. Brumley and N. Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, pages 355–371. Springer, 2011.

[16] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[17] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[18] J.-S. Coron and I. Kizhvatov. An efficient method for random delay generation in embedded software. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 156–170. Springer, 2009.

[19] J. M. DiCarlo and B. A. Wandell. Rendering high dynamic range images. In *Electronic Imaging*, pages 392–401. International Society for Optics and Photonics, 2000.

[20] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Security and Privacy, 1984 IEEE Symposium on*, pages 75–75. IEEE, 1984.

[21] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[22] W.-M. Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[23] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 13–pp. IEEE, 2006.

[24] E. Käsper and P. Schwabe. Faster and timing-attack resistant aes-gcm. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 1–17. Springer, 2009.

[25] K. Kaukonen and R. Thayer. A stream cipher encryption algorithm "arc-four", 1999.

[26] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Computer SecurityESORICS 98*, pages 97–110, 1998.

[27] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[28] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*, pages 564–580. Springer, 2012.

[29] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1055–1062. ACM, 2013.

[30] J. Lee, J. Park, S. Lee, and J. Kim. The SEED encryption algorithm. *SEED*, 2005.

[31] H. Mantel and A. Starostin. Transforming out timing leaks, more or less. In *European Symposium on Research in Computer Security*, pages 447–467. Springer, 2015.

[32] M. Matsui, S. Moriai, and J. Nakajima. A description of the Camellia encryption algorithm. 2004.

[33] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[34] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *European Symposium On Research In Computer Security*, pages 279–296. Springer, 2006.

[35] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[36] C. S. Pasareanu, Q.-S. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 387–400. IEEE, 2016.

[37] C. Percival. Cache missing for fun and profit. http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf, 2005.

[38] R. Rivest. A description of the RC2 (r) encryption algorithm. 1998.

[39] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast software encryption*, pages 191–204. Springer, 1994.

[40] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[41] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[42] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[43] Standard, NIST-FIPS. Announcing the advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:1–51, 2001.

[44] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29–43. IEEE, 2003.

[45] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 595–606. ACM, 2010.

[46] X.-j. Zhao, T. Wang, and Y. Zheng. Cache timing attacks on camellia block cipher. *IACR Cryptology ePrint Archive*, 2009:354, 2009.

Appendix A

ARM Instruction Times

Legend	
D	Dy: Dual issue as younger *and* older; Do: Dual issue as older only
I	Issue time
L	Latency
M	Memory instruction: time depends on cache hit
FI	Condition false issue time
FL	Condition false lock time (latency)
SB	Can receive a semi-bypass
ER	Early Registers (1 cycle early)
LR	Late Registers: number of cycles late

Instruction	D	I	L	M	FI	FL	SB	ER	LR
`adc<c> <rd>, <rn>, #<const>`	Do	1	2		1	2	+		
`adc<c> <rd>, <rn>, <rm>, <shift>`	Do	1	2		1	2	+	rm	
`adc<c> <rd>, <rn>, <rm>`	Do	1	2		1	2	+		
`adc<c> <rd>, <rn>, <rm>, <type> <rs>`	Do	1	2		1	2	+	rm, rs	
`add<c> <rd>, <rn>, #<const>`	Dy	1	2		1	2	+		
`add<c> <rd>, <rn>, <rm>, <shift>`	Do	1	2		1	2	+	rm	
`add<c> <rd>, <rn>, <rm>`	Do	1	2		1	2	+		
`add<c> <rd>, <rn>, <rm>, <type> <rs>`	Do	1	2		1	2	+	rm, rs	
`add<c> <rd>, <rn>, #<const>`	Dy	1	2		1	2	+		
`add<c> <rd>, <rn>, <rm>, <shift>`	Do	1	2		1	2	+	rn	
`add<c> <rd>, <rn>, <rm>`	Do	1	2		1	2	+		
`and<c> <rd>, <rn>, #<const>`	Dy	1	2		1	2	+		
`and<c> <rd>, <rn>, <rm>, <shift>`	Do	1	2		1	2	+	rm	
`and<c> <rd>, <rn>, <rm>`	Do	1	2		1	2	+		
`and<c> <rd>, <rn>, <rm>, <type> <rs>`	Do	1	2		1	2	+	rm, rs	

Instruction	D	I	L	M	FI	FL	SB	ER	LR
asr<c> <rd>, <rm>, #<imm>	Do	1	2		1	2		rm	
asr<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
bic<c> <rd>, <rn>, #<const>	Dy	1	2		1	2	+		
bic<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
bic<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
bic<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
blx<c> <rm>		2	-		2	-			
bx<c> <rm>		1	-		1	-			
bxj<c> <rm>		1	-		1	-			
clz<c> <rd>, <rm>	Do	1	2		1	2	+		
cmn<c> <rn>, #<const>	Dy	1	1		1	1	+		
cmn<c> <rn>, <rm>, <shift>	Do	1	1		1	1	+	rm	
cmn<c> <rn>, <rm>	Do	1	1		1	1	+		
cmn<c> <rn>, <rm>, <type> <rs>	Do	1	1		1	1	+	rm, rs	
cmp<c> <rn>, #<const>	Dy	1	1		1	1	+		
cmp<c> <rn>, <rm>, <shift>	Do	1	1		1	1	+	rm	
cmp<c> <rn>, <rm>	Do	1	1		1	1	+		
cmp<c> <rn>, <rm>, <type> <rs>	Do	1	1		1	1	+	rm, rs	
eor<c> <rd>, <rn>, #<const>	Dy	1	2		1	2	+		
eor<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
eor<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
eor<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
ldm<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldm<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldm<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldm<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmda<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmda<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmdb<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmdb<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmib<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldmib<c> <rn>{!}, <registers>		2*	1*	+	2*	1*		rn	
ldr<c> <rt>, [<rn>, #+/-<imm>]		1	1	+	1	1		rn	
ldr<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldr<c> <rt>, [<rn>, +/-<rm>, <shift>]{!}		1	1	+	1	1		rn, rm	
ldr<c> <rt>, [<rn>, +/-<rm>]{!}		1	1	+	1	1		rn, rm	
ldrb<c> <rt>, [<rn>, #+/-<imm>]		1	1	+	1	1		rn	
ldrb<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldrb<c> <rt>, [<rn>, +/-<rm>, <shift>]{!}		1	1	+	1	1		rn, rm	
ldrb<c> <rt>, [<rn>, +/-<rm>]{!}		1	1	+	1	1		rn, rm	
ldrbt<c> <rt>, [<rn>], #+/-<imm>		1	1	+	1	1		rn	
ldrex<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldrh<c> <rt>, [<rn>, #+/-<imm>]		1	1	+	1	1		rn	
ldrh<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldrh<c> <rt>, [<rn>, +/-<rm>]{!}		1	1	+	1	1		rn, rm	
ldrsb<c> <rt>, [<rn>, #+/-<imm>]		1	1	+	1	1		rn	
ldrsb<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldrsb<c> <rt>, [<rn>, +/-<rm>]{!}		1	1	+	1	1		rn, rm	
ldrsh<c> <rt>, [<rn>, #+/-<imm>]		1	1	+	1	1		rn	
ldrsh<c> <rt>, [<rn>]		1	1	+	1	1		rn	
ldrsh<c> <rt>, [<rn>, +/-<rm>]{!}		1	1	+	1	1		rn, rm	
ldrt<c> <rt>, [<rn>], #+/-<imm>		1	1	+	1	1		rn	
lsl<c> <rd>, <rm>, #<imm>	Do	1	2		1	2		rm	

Instruction	D	I	L	M	FI	FL	SB	ER	LR
lsl<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
lsr<c> <rd>, <rm>, #<imm>	Do	1	2		1	2		rm	
lsr<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
mla<c> <rd>, <rn>, <rm>, <ra>		1	3		1	2		ra: 2	
mov<c> <rd>, #<const>	Dy	1	2		1	2	+		
mov<c> <rd>, <rm>	Dy	1	2		1	2	+		
mul<c> <rd>, <rn>, <rm>		1	3		1	2			
mvn<c> <rd>, #<const>	Do	1	2		1	2	+		
mvn<c> <rd>, <rm>, <shift>	Do	1	2		1	2	+	rm	
mvn<c> <rd>, <rm>	Do	1	2		1	2	+		
mvn<c> <rd>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
mov<c> <rd>, <rm>	Dy	1	2		1	2	+		
orr<c> <rd>, <rn>, #<const>	Dy	1	2		1	2	+		
orr<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
orr<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
orr<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
pkhbt<c> <rd>, <rn>, <rm>, lsl #<imm>	Do	1	1		1	1			
pop<c> <registers>		2*	1*	+	2*	1*		sp	
pop<c> <registers>		2*	1*	+	2*	1*		sp	
push<c> <registers>		3*	3*	+	2*	2*		sp	<registers>: 2
qadd<c> <rd>, <rm>, <rn>	Do	1	3		1	1	+		
qadd16<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
qadd8<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
qasx<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
qdadd<c> <rd>, <rm>, <rn>	Do	1	3		1	2			
qdsub<c> <rd>, <rm>, <rn>	Do	1	3		1	2			
qsax<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
qsub<c> <rd>, <rm>, <rn>	Do	1	3		1	1	+		
qsub16<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
qsub8<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
rev<c> <rd>, <rm>	Do	1	2		1	2			
rev16<c> <rd>, <rm>	Do	1	2		1	2			
revsh<c> <rd>, <rm>	Do	1	2		1	2			
ror<c> <rd>, <rm>, #<imm>	Do	1	2		1	2		rm	
ror<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
rrx<c> <rd>, <rm>	Do	1	2		1	2			
rsb<c> <rd>, <rn>, #<const>	Do	1	2		1	2	+		
rsb<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
rsb<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
rsb<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
rsc<c> <rd>, <rn>, #<const>	Do	1	2		1	2	+		
rsc<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
rsc<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
rsc<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
sadd16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sadd8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sasx<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sbc<c> <rd>, <rn>, #<const>	Do	1	2		1	2	+		
sbc<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
sbc<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sbc<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
sel<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
shadd8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
shadd16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
shasx<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
shsax<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		

Instruction	D	I	L	M	FI	FL	SB	ER	LR
shsub16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
shsub8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
smlal<c> <rdlo>, <rdhi>, <rn>, <rm>		1	3		1	2		rdlo: 2	
smull<c> <rdlo>, <rdhi>, <rn>, <rm>		1	2		1	1			
ssat<c> <rd>, #<imm>, <rn>, <shift>	Do	1	2		1	2	+	rn	
ssat<c> <rd>, #<imm>, <rn>	Do	1	2		1	2	+		
ssat16<c> <rd>, #<imm>, <rn>	Do	1	2		1	2	+		
ssax<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
ssub16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
ssub8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
stm<c> <rn>{!}, <registers>		2*	3*	+	2*	3*		rn	<registers>: 2
stmda<c> <rn>{!}, <registers>		2*	3*	+	2*	3*		rn	<registers>: 2
stmdb<c> <rn>{!}, <registers>		2*	3*	+	2*	3*		rn	<registers>: 2
stmib<c> <rn>{!}, <registers>		2*	3*	+	2*	3*		rn	<registers>: 2
str<c> <rt>, [<rn>, #+/-<imm>]		1	2	+	1	2		rn	rt: 2
str<c> <rt>, [<rn>]		1	2	+	1	2		rn	rt: 2
str<c> <rt>, [<rn>, +/-<rm>, <shift>]{!}		1	2	+	1	2		rn, rm	rt: 2
str<c> <rt>, [<rn>, +/-<rm>]{!}		1	2	+	1	2		rn, rm	rt: 2
strb<c> <rt>, [<rn>, #+/-<imm>]		1	2	+	1	2		rn	rt: 2
strb<c> <rt>, [<rn>]		1	2	+	1	2		rn	rt: 2
strb<c> <rt>, [<rn>, +/-<rm>, <shift>]{!}		1	2	+	1	2		rn, rm	rt: 2
strb<c> <rt>, [<rn>, +/-<rm>]{!}		1	2	+	1	2		rn, rm	rt: 2
strbt<c> <rt>, [<rn>], #+/-<imm>		1	2	+	1	2		rn	rt: 2
strex<c> <rd>, <rt>, [<rn>]		1	2	+	1	2		rn	rd: 2
strh<c> <rt>, [<rn>]		1	2	+	1	2		rn	rt: 2
strh<c> <rt>, [<rn>, #+/-<imm>]		1	2	+	1	2		rn	rt: 2
strh<c> <rt>, [<rn>, +/-<rm>]{!}		1	2	+	1	2		rn, rm	rt: 2
strt<c> <rt>, [<rn>], #+/-<imm>		1	2	+	1	2		rn	rt: 2
sub<c> <rd>, <rn>, #<const>	Dy	1	2		1	2	+		
sub<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rm	
sub<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sub<c> <rd>, <rn>, <rm>, <type> <rs>	Do	1	2		1	2	+	rm, rs	
sub<c> <rd>, <rn>, #<const>	Dy	1	2		1	2	+		
sub<c> <rd>, <rn>, <rm>, <shift>	Do	1	2		1	2	+	rn	
sub<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
sxtab<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
sxtab16<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
sxtah<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
sxtb<c> <rd>, <rm>	Dy	1	2		1	2	+		
sxtb16<c> <rd>, <rm>	Dy	1	2		1	2			
sxth<c> <rd>, <rm>	Dy	1	2		1	2	+		
teq<c> <rn>, #<const>	Dy	1	1		1	1	+		
teq<c> <rn>, <rm>, <shift>	Do	1	1		1	1	+	rm	
teq<c> <rn>, <rm>	Do	1	1		1	1	+		
teq<c> <rn>, <rm>, <type> <rs>	Do	1	1		1	1	+	rm, rs	
tst<c> <rn>, #<const>	Dy	1	1		1	1	+		
tst<c> <rn>, <rm>, <shift>	Do	1	1		1	1	+	rm	
tst<c> <rn>, <rm>	Do	1	1		1	1	+		
tst<c> <rn>, <rm>, <type> <rs>	Do	1	1		1	1	+	rm, rs	
uadd16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uadd8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uasx<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uhadd16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		

Instruction	D	I	L	M	FI	FL	SB	ER	LR
uhadd8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uhasx<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uhsax<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uhsub16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uhsub8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
umaal<c> <rdlo>, <rdhi>, <rn>, <rm>		2	3		2	2			rdh: 1, rdlo: 2
umlal<c> <rdlo>, <rdhi>, <rn>, <rm>		1	3		1	2			rdlo: 2
umull<c> <rdlo>, <rdhi>, <rn>, <rm>		1	2		1	1			
uqadd16<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
uqadd8<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
uqasx<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
uqsax<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
uqsub16<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
uqsub8<c> <rd>, <rn>, <rm>	Do	1	3		1	1	+		
usad8<c> <rd>, <rn>, <rm>		1	3		1	2			
usada8<c> <rd>, <rn>, <rm>, <ra>		1	3		1	2			ra: 2
usat<c> <rd>, #<imm>, <rn>, <shift>	Do	1	2		1	2	+	rn	
usat<c> <rd>, #<imm>, <rn>	Do	1	2		1	2	+		
usat16<c> <rd>, #<imm>, <rn>	Do	1	2		1	2	+		
usax<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
usub16<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
usub8<c> <rd>, <rn>, <rm>	Do	1	2		1	2	+		
uxtab<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
uxtab16<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
uxtah<c> <rd>, <rn>, <rm>	Do	1	2		1	2			
uxtb<c> <rd>, <rm>	Dy	1	2		1	2	+		
uxtb16<c> <rd>, <rm>	Dy	1	2		1	2			
uxth<c> <rd>, <rm>	Dy	1	2		1	2	+		

* Depends on length of reglister list, indicated time based on list of three registers.
Memory instruction execution times are influenced by cache hits and misses.