Andreas Hechl, BSc

Design and Implementation of a Model-based Symbolic Execution on Constrained Devices

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor
Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor
Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr.techn. Andreas Sinnhofer

Graz, May 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

Computer programs processing confidential data, amongst other data, are hard to be proven secure. It is hard to check such programs for unwanted or even malicious publication of confidential material.
This thesis introduces the concept of a secure program. It is a collection of environmental software and hardware measurements, including a high security device, where the secure program should be executed on. Additionally, restrictions regarding the execution environment itself are defined to prevent a security breach from happening.
The main part of this thesis is the design and implementation of a security guard. The security guard is an abstract model derived from the actual implementation of the secure program's execution environment on a high security device and runs before the actual execution of the secure program. The security guard checks against pre-defined rules and executes operations on symbolic data to approve a secure program that does protect all secrets, or reject a (not so) secure program failing to fulfil all checks. To verify whether a given secure program is secure the security guard symbolically executes the program on the basis of the given instructions, parameters, and references and then traces the program's output data for leaking secret material. Once the secure program is proven secure it is allowed to be executed with actual data. The implementation of the security guard was done in the programming language C, aimed to be memory efficient and having a short execution time to be executable on a hardware restricted high security device.
Finally, a detailed example of a simple execution of the security guard will be presented and a discussion of security considerations and performance is made.

Kurzfassung

Computerprogramme, die geheimes Datenmaterial verarbeiten, auf deren Sicherheit zu überprüfen ist ein schweres Unterfangen. Eine Überprüfung solcher Programme auf versehentlich, oder sogar böswillig, eingebaute Fehler, mit der Veröffentlichung von geheimen Datenmaterial als Folge, ist nur mit großem Aufwand möglich.
Diese Masterarbeit beginnt mit der Vorstellung eines Konzeptes für ein sicheres Programm (secure program). Ein sicheres Programm ist eine Zusammenstellung aus Software- und Hardwarevorkehrungen um einerseits die Möglichkeiten eines Angriffs gering zu halten und andererseits eine Analyse des sicheren Programms einfacher zu gestalten. Zu den Vorkehrungen gehören, unter anderem, ein so genanntes High Security Device, auf dem das sichere Program ausgeführt wird und Einschränkungen der Laufzeitumgebung.
Im Hauptteil dieser Masterarbeit wird der Entwurf und die Implementierung eines Sicherheitswächters behandelt. Der Sicherheitswächter ist ein abstraktes Modell abgeleitet von der tatsächlichen Implementierung der Laufzeitumgebung eines sicheren Programms und führt ein sicheres Programm vor seiner tatsächlichen Ausführung aus. Der Sicherheitswächter überprüft bestimmte, vorher definierte, Regeln und führt symbolische Operationen aus, um ein sicheres Programm, das geheime Daten geheim hält, freizugeben, oder ein (nicht so) sicheres Programm, das Sicherheitslücken aufweist, abzulehnen. Zur Überprüfung eines sicheren Programms wird dieses, mit Hilfe der gespeicherten Instruktionen, Felder und Parameter, symbolisch ausgeführt, um dann die ausgegebenen Daten auf sicherheitsrelevantes Material zu untersuchen. Sobald die Sicherheit eines sicheren Programms nachgewiesen ist, darf es mit tatsächlichen Daten ausgeführt werden. Die Implementierung des Sicherheitswächters wurde mit dem Ziel speichereffizient und laufzeitoptiemiert zu sein und somit auf einem hardwarebeschränkten High Security Device ausführbar zu bleiben in der Programmiersprache C durchgeführt.
Am Ende der Arbeit wird ein umfangreiches Beispiel einer einfachen Ausführung des Sicherheitswächters erläutert, Sicherheitsbedenken besprochen und die Leistungseigenschaften analysiert.

Acknowledgement

This master thesis was carried out at the Institute for Technical Informatics at the Technical University of Graz during the years 2016/2017.

I want to thank all people from the Institute for Technical Informatics who supported me during the work on my master thesis. Especially I want to thank my two supervisors Ass. Prof. Dipl.-Ing. Dr.techn. Christian Steger and Dipl.-Ing. Dr.techn Andreas Sinnhofer, who had always time whenever I ran into a trouble spot or had a question about my research or writing.

Finally, I must express my very profound gratitude to my parents and to my spouse, Viola, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Graz, May 2018 Andreas Hechl

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives a short introduction to the general preconditions and motivations of this thesis. It introduces the concept of secure programs, which processes, among other data, confidential data material. Then it introduces the purpose and motivation of the security guard, which will furthermore be the focus of this thesis. Finally, a quick outline of the rest of the thesis will be given.

1.1 Secure Programs

This thesis focuses on computer programs, processing confidential data material, among other data. Meaning at some time in the execution process the computer handles plain confidential data, i. e. unencrypted keys and certificates, used for cryptographic operations, but also related data to it, like checksums or meta data. This intermediate data should not be readable, or observable, by arbitrary persons and should only be published in a secure way, for instance encrypted for a specific user group.

In order to prevent the unwanted publication of confidential material, which will be abbreviated "secrets" in the rest of this thesis, the concept of a secure program has been developed and is the starting point of this thesis. The concept of a secure program is a conglomeration of environmental software and hardware measurements and systematical restrictions to prevent a security breach from happening. These measurements and restrictions will be explained in the next paragraphs.

The first measurement is to run a secure program exclusively on a special hardware, called high security device. The concept of a high security device is to hinder adversaries from hardware or software related attacks, e.g. side channel attacks, or trying to get control of the operating system of the device, and ensures that only the output can be obtained by the calling user, intermediate data is not accessible. The exact definition of a high security device should not be the focus of this thesis but it can range from at least a PC in a high secure environment with restricted access to a hardware security module.

Secure programs are meant to be executed multiple times on different input data. Therefore, as the next measurement, a static input/output-slot system was developed to separate non-secret data, secret data, and the secure program itself on defined inputs of the high security device. The parameter count for secret and non-secret input as well as output fields is fixed with respect to a specific secure program. This configuration of a secure program, together with secret and non-secret input data, can be seen in figure 1.1. Users can provide the signed and encrypted secure program with their personal secret and non-secret input parameters to the high security device. The high security device processes the data and publishes the user specific data, usually encrypted, to the output.

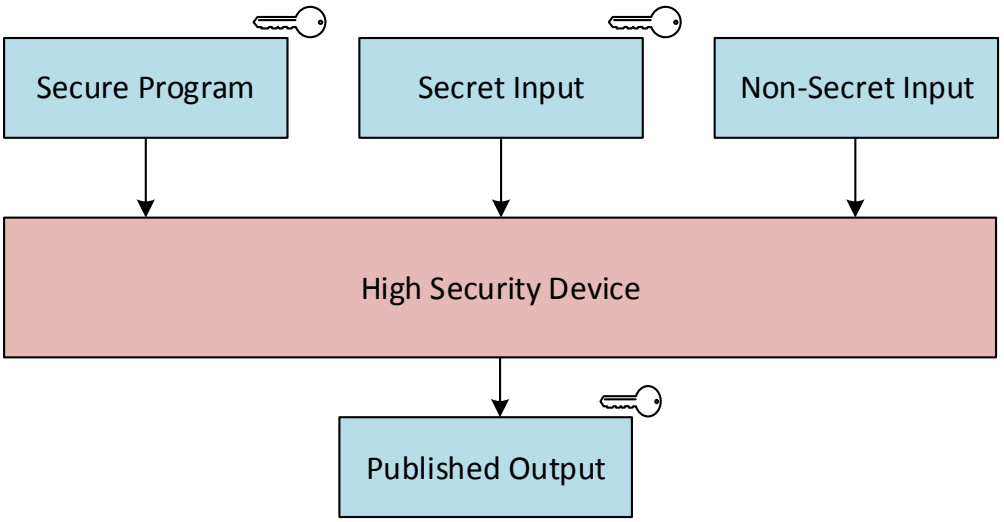

Figure 1.1: Configuration of a secure program with secret and non-secret input with a high security device

For simplicity and security reasons the next measurement to prevent secrets from being published, is a restriction of the basic building blocks of the program, the instruction-set. Only a specific set of these instructions, managing and processing the data, is supported. The instructions have a defined set of input and output parameters and are supported by the high security device. Additionally, the program is not able to process branches or loops, which are not unrolled.

A more thorough explanation about the concept and structure of a secure program can be found in section 3.1.

The final measurement is a program acting as a security guard. This program checks specifically the given secure program for security breaches and is introduced in the next section.

1.2 Motivation

Because of the sensible data processed and handled by the secure program, an author of such a program can, deliberately or unintended, make mistakes while writing a program, which can lead to leaking parts or all of the secret material. This can be done, for instance, by using an insecure sequence of operations or publishing parts or whole secrets to the output of the high security device directly without previous encryption.
To prevent such security violations a security guard is introduced. The security guard can be run before or during the actual execution of the secure program on the high security device on an symbolic data set derived from the actual data. A discussion of secure program exectuion order can be found in section 3.4. The security guard should detect wrong operations on confidential material and ensure that data, which leaks confidential material or whole secrets, is not published outside the high security device. It can also spot misuse of instructions and parameters configuring these instructions in the secure program to prevent “bugs”, which otherwise would be detected later during actual execution on a high security device, leading to a higher impact on resources and financials.

Chapter 2

Related Work

This chapter covers the related work regarding the basic topics of this thesis. Section 2.1 introduces information flow security and its different categories. Subsequently, section 2.2 gives an overview on program analysis methods and its two major sub-categories *static program analysis* and *dynamic program analysis*. Finally, section 2.3 explains the current knowledge in cryptography analysis and basic concepts of it.

2.1 Information Flow Security

This section introduces the idea of information flow security. After a short introduction to the topic the categories, side channels and examples for better understanding are described in section 2.1.2. Then in section 2.1.3 information flow in function sequence is discussed and a short example of a practical attack is given. Finally in section 2.1.4 *static information flow control* as a way to detect declassification of secure data is presented.

2.1.1 Introduction

Unlike the content of most scientific research that considers how secret data can be stored or transferred in a confidential way, so that not authorised entities cannot figure out the content of these secrets, information flow security describes the process of transferring information between different variables in processes, like computer programs. Most of the time this information, which was implemented and then executed by the program, transfer is the desired process. But when dealing with partially or fully transferring secret data information from one variable to another, a leak to other, not secret variables can happen and eventually leads to a security breach to not authorised entities. In the worst case this security breach happens without notice [1].

2.1.2 Categories and Examples

There are two major categories of information flow: *direct flow* and flow through a *covert (side) channel* [1].

In the following code listing an example of direct information flow is given.

```
int function(int h){
  int l = h;
  return l;
}
```

Listing 2.1: Explicit information flow

This function contains two variables, h, whose content is secure information, and l, which can be observed by anyone. Now a direct assignment from h to l is been made and all information about h is now part of l. In most cases this information flow is desired behaviour in an implementation and can be detected easily.
Information flow through a side channel has many variants, structured by the *covert channels* they use [1]. One is the implicit information flow. The following listing shows leakage of secure information by implicit information flow through a non-secure variable.

```
int function(int h){
  if(h == 32){
    l = 1;
  } else {
    l = 0;
  }
}
```

Listing 2.2: Implicit information flow, based on Fig 1. of [1]

In this function a part of the data stored in the secure variable h is leaked by assigning the value 1 to the non-secure variable l if h is exactly 32. With this knowledge it is explicitly known whether h is 32 or not and part of the information stored in h is leaked. This process is called declassification of sensitive information by Sabelfeld and Sands [2].
Another more subtle covert channel is the *timing channel*. The following listing shows an example of code vulnerable to a timing attack.

```
int function(int h){
  if(h == 32){
    // do some stuff
  }
}
```

Listing 2.3: Code vulnerable to a timing attack

The execution time of this function is longer if the secret variable h is exactly 32. And the same information as in the listing before can be considered as known.
Other known covert channels are [1]:

- **Termination channel:** Information about secure data is gathered by analysing whether a program terminates or not.
- **Probabilistic channel:** By changing the probability distribution of observable data an attacker can make recurrent computations and observe stochastic properties.
- **Resource exhaustion channels:** By limiting memory, processing time, disk space, or another resource of the used computer information can be gathered.
- **Power channel:** By measuring the power usage of the computer or specific hardware (CPU, GPU, RAM, HDD) on the computer information can be gathered.

2.1.3 Information Flow in Function Sequences

Sometimes a single function does not declassify information but a set of these functions can. Especially if the functions are used in a sequence they are not designed for or the input is not checked for consistency. These attacks cannot be directly analysed or prohibited by the methods used in cryptography analysis (section 2.3): although cryptography analysis uses a mathematical approach to ensure that no information is released by a system it does not consider information propagation.
Example attacks exploiting information flow in function sequences by calling security APIs from secure systems are discussed by Mike Bond[1] and Ross Anderson.
One of these attacks was a known-key attack [3, Section 2.3]: A given security module's (SM) API can generate key shares (KS) and returns this key share encrypted with a master-key (MK) not known by the user: $(KS)_{MK}$. Another function of the API is the combination of two key shares to get a full fledged key, the combination itself was a simple XOR-operation. The API gets the key shares encrypted with the master key $(KS_1)_{MK}$, $(KS_2)_{MK}$ and returns the resulting key again encrypted with the same master key $(KS_1 \oplus KS_2)_{MK}$.
The attack consisted of generating just one key share and combining this key share with itself using the API. The key share was XORed with itself which resulted in a key with all zeroes: $(KS_1 \oplus KS_1)_{MK} = (0)_{MK}$. The key now was known by the attacker and can be used for malicious behaviour.

2.1.4 Static Information Flow Control

One way of preventing declassification of secure data by using static program analysis is static information control. The used type system of the program is extend by a security type system, which consists of two parts: an ordinary data type (int, float, char) and a

[1] `https://www.cl.cam.ac.uk/~rja14/#Protocols`

label describing the content of the variable and how it may be used. This label is static, which means it does not change during runtime. When compiling a program the compiler checks the labelled variables and reports a security violation if an improper assignment is taken.
To be able to track the security of implicit information control a *program-counter label* (pc) is introduced which marks the execution parts of the program with specific requirements the variables have to meet. Looking at the example in listing 2.2 the program counters of all conditions of the if-statement get a label marking high security. As a result the compiler would report a security violation because a low security variable is used in a high security labelled area when compiling this section [1].
However, this approach can only reveal information declassification through implicit information flow, but cannot reveal information declassification on other covert channels like the timing channel.

2.2 Program Analysis

This section gives a short overview on program analysis methods and its two major subcategories *static program analysis* and *dynamic program analysis*. The two program analysis concepts *abstract interpretation* and *symbolic execution* are discussed more accurately with examples, fields of applications and problems and limitations within these concepts.

2.2.1 Introduction

Program Analysis describes the method of generating approximations of how a program behaves when executed [4]. The goal is to either optimise the program, for instance to reduce memory usage or increase performance, or to ensure the correctness and stability of a program, e.g. behaviour or bug testing.

2.2.2 Static and Dynamic Program Analysis

Program analysis is divided into two major categories: static program analysis and dynamic program analysis which will be explained in more detail in section 2.2.2 and section 2.2.2. The two categories will be then compared in section 2.2.2.

Static Program Analysis

Static program analysis does not actually execute the program, but analyses the program for all paths of execution at compile time[5]. It can be used to detect possible flaws during development of a program or for code review. The typical usages of static program analysis are [6]:

- **Type checking** is the most used and known form of static analysis. Variables have to be assigned to a type (int, long, string, object-name) to detect false assignments and other faults.
- **Style checking** enforces a ruleset to the usage of whitespaces, naming, commenting, or program structure. Usually it does not detect faults in the program but improves the code's readability, understandability, and maintainability.
- **Program understanding** tools help a user to understand the code more easily and gain insight in programs, which were not written by this user. Most integrated development tolls (IDEs) support such analysis. Some examples are "find the declaration of this variable", "find all uses of this method", or refactoring of variables.
- **Program verification** is checking the code against a specification in a tool which verifies the correct implementation. Because the specification must be very detailed for this check, it is only used in areas where small implementation errors can cause high impacts and are difficult to fix, like hardware design.
- **Property checking** is, in contrast to program verification, checking the code against a partial specification. Tools implementing property checking are mostly based on applying logical inference or model checking.
- **Bug finding** tools are additional static analysis tools to help finding common coding mistakes which are not determined by the compiler. Bug finding tools reveal mistakes like double free of a pointer or variable assignment in conditions.
- **Security review** tools use some of the same techniques the other analysis tools use but with a security focus. They are mostly a combination of property checkers and bug finders. An exemplary function is the detection of buffer overflows.

The advantage of static program analysis is that possible errors and programming faults can be detected early in the development stage of the program and therefore is much cheaper to fix these errors.

Dynamic Program Analysis

Dynamic program analysis observes a program during a concrete execution to gather information about it. In many cases the program analysis tool may by executed first, then the program to be analysed gets called by the analysis tool. Dynamic program analysis performs best if most, or, preferably, all aspects and features of the analysed program, are observed, this property is called *code coverage*. The more parts of the code are executed during the analysis the better an assessment can be made for the whole program. Some dynamic program analysis tools' purpose is to check this very property, analysing the code coverage of, for instance, a testing framework.
A widely known example for a dynamic program analysis tool would be Valgrind. Valgrind

is a runtime memory usage analysis tool and runs while executing a C or C++ program to find possible memory leaks, violations, or other bugs and errors [7]. Another big application for dynamic program analysis is testing for functional safety, which not only tests for program failures within the program, but also how a program reacts if the inputs are defective, e. g. inputs are out of range or not updated any more, or some parts of the software or hardware gets faulty. Functional safety analysis is widely used in the aviation industry or companies working in the field of other means of transportation, but also, for instance, in power plants.

Comparison

Gosain and Sharma [8] and Michael D. Ernst [9] defined the following important characteristics regarding static analysis:

- Static Analysis **does not require program a to be executed**, in most implementations it builds an abstract model of the program and analyses this model.
- Because the program is abstracted, an abstract analysis is **less precise** and does not consider all possible run-time states of the program.
- Static analysis **holds for all the executions**, but has a more conservative, more approximate result.
- It **lacks in handling run-time programming language features**.
- Because there is no actual program execution during analysis, it **incurs less overhead**

And on the opposite the characteristics of dynamic analysis:

- Dynamic analysis **requires a program to be executed** to get a result.
- The result is **more precise** than the result of a static analysis because no abstraction has to be done.
- The result does only **hold for the particular execution** of the program, executions with other parameters are not considered.
- Dynamic program analysis is **best suited to handle run-time programming language features** like polymorphism, dynamic binding, threads etc.
- In contrast to static analysis, dynamic analysis **incurs large run-time overheads**.

2.2.3 Soundness of Program Analysis Tools

Soundness of an analysis tool is a guarantee for the validity of the results, produced by the tool. It means that a tool is sound if a potential error or violation is reported correctly, but

does not mean that every encountered error is an actual violation. These false errors are called *false positives*. Unsound tools on the other hand not only produce false positives, they also can report false negatives, which means an actual error or violation is stated as safe or is not recognised, which may have much bigger consequences when releasing the software. [6]

2.2.4 Abstract Interpretation

This section gives an introduction into abstract interpretation, some examples, projects and software using abstract interpretation, and a discussion of limits of it.

Explanation

Most of the time not all aspects of the output of a program are necessary to check for a specific behaviour. Therefore not all properties of the input are needed. Abstract interpretation takes just these aspects of the actual input objects and executes them on an abstract model. The model is derived from the actual program with regard to specific features, it represents a derived set of functionality of the actual program operations but is sound with respect to the correctness of the abstraction.
The most crucial phase of abstract interpretation is during the generation of the abstract model: is this model wrong or implemented in a sloppy way with regards to the feature set of the actual program, the interpretation by the model can produce faults and wrong assumptions of the behaviour from actual program [10].

Examples

Rule of Signs

A simple example for abstract implementation would be the rule of signs [10] [11]. A given program computes the multiplication of two signed integers, the result of the computation is not needed but the sign of the output is needed. Therefore it is just necessary to save the sign (+1 or -1) of the input and derive an abstract model, which obeys the same rules of the actual implementation with respect to the limited information. In this example the simple model would be: same signed input causes positive output, different signed input causes negative output:

$$+1 * +1 = +1$$

$$-1 * -1 = +1$$

$$+1 * -1 = -1$$

$$-1 * +1 = -1$$

With this abstract domain all multiplication operations of positive and negative real numbers are supported. An abstraction expression α is defined to obtain an abstract value from an actual value.

$$\alpha(22) = +1$$

$$\alpha(-44) = -1$$

But this primitive first model is not yet complete regarding all parameters the actual implementation supports. There is one special number which is not yet handled with: zero. Without the implementation of this status the analysis cannot be correct. By extending the abstract domain with a new symbol (0) and some new rules, zero is supported too.

$$+1 * 0 = 0$$

$$-1 * 0 = 0$$

$$0 * -1 = 0$$

$$0 * +1 = 0$$

Now all real numbers are supported by the introduced abstract model for the multiplication operation. Then again, for the abstraction of the addition function some rules can be determined easily:

$$+1 + 0 = +1$$

$$-1 + 0 = -1$$

$$-1 + -1 = -1$$

$$+1 + +1 = +1$$

But for the addition of different signed values the abstract domain has to be extended once more by another value, T, which represents an unknown value. Here are some, but not all, operations to define the T value:

$$+1 + -1 = T$$

$$T + +1 = T$$

$$T + T = T$$

Similar operations can be defined for the multiplication functions to fully support all calculations with the abstract model.

Dimension Calculus

Another approach may be an abstract domain based on dimension calculus from elementary physics with values like *length*, *surface*, *time*, *speed*, *acceleration* and their relationship to each other [10] . Some exemplary abstract operations would be:

$$length + length = length$$

$$length * length = surface$$

$$length/time = speed$$

$$speed/time = acceleration$$

To get the abstract interpretation of a concrete expression it has to be derived by a defined operator and then the abstract expression can be evaluated taking the abstract operators, see [10] for more information.

Problems, Issues, and Limitations

Abstract interpretation means a loss of precision. With this trade-off in mind a result in a feasible amount of time can be accomplished and the results can be interpreted more easily to get an approximate solution. The soundness of these solutions is ensured when certain mathematical constraints between abstract and the actual domain are fulfilled. If the implementation is not done correctly, an infeasible amount of time is needed to compute all iterations to reach a fixed point or the soundness of the approximation is not given any more, like in the example in section 2.2.4 when the "0" input parameter is not yet handled. This has to be considered during development of the abstract model [12].

2.2.5 Symbolic Execution

This section gives an introduction into symbolic execution, some examples, projects and software using symbolic execution, and a discussion of limits of it.

Explanation

Symbolic execution changes the input parameters from actual real data objects, for instance integer values or strings, to arbitrary symbols. The program executions semantic has to be extended to accept these symbols providing the normal execution as a special case. Variables in the program are not assigned the values but these input symbols operations and are saved constantly in addition to other properties in the execution state of the program. The control flow (loops, conditions) also is analysed by defining *path conditions* which trigger specific paths and are logged additionally in the execution state. In the end

an extensive breakdown of the symbolically executed program is given. Possible output formulas dependent from input values and sets of input data, which trigger different paths of the program in addition to the output data of the relative paths, are defined [13].

Example

In code listing 2.4 a simple function is given which takes an integer x, makes some computations and finally decides with a fixed condition which further actions will be taken.

```
int function(int x)
{
  int y=x*2;
  if (y < 7) {
    return y;
  } else {
    fail();
  }
}
```

Listing 2.4: Symbolic Execution Example

For the symbolic execution the function will not be executed with a concrete value but with the symbol α. At the beginning the input value α is assigned to the variable x, the current execution state is $x = \alpha$. In the next line x is multiplied with two and assigned to y, the next execution state is $y = \alpha * 2$. Now the condition is analysed, the symbolic value of y is known so the condition can be described with $\alpha * 2 < 7$ converting to $\alpha < 4$. In conclusion it is known that if α is smaller than 4 the if condition is fulfilled and the function terminates with the return value smaller then 4, if α is bigger than or equal to 4 the else branch is executed and the program fails. By using symbolic execution not only the dependency of the condition from the input value has been examined, but also the possible return values have been formalised.

Application

One of the biggest applications of symbolic execution is static test generation, where all inputs to cover all program paths are generated for full test coverage [14]. With this method most if not all possible execution paths can be reached and potential programming faults, e.g. null pointer exceptions, can be found. A list of papers and projects related to symbolic execution can be found on Github[2] .

Problems, Issues, and Limitations

Symbolic execution has some disadvantages in its application. Two known limitations are regarding path explosion and environment interaction which are discussed in this section.

[2] https://github.com/saswatanand/symexbib

Path Explosion

With increasing program size the conditional paths can explode exponentially, for example the same function is called under different conditions with similar but distinct values and the function itself has conditional paths. Another challenge are unbounded loop iterations [13]. Solutions are parallelisation of independent paths, merging similar paths or manually deselecting unwanted paths.

Environment Interactions

More complicated programs interact with their environment in other ways than just by calling internal functions. Conditions may be dependent on system calls and signals and therefore are more difficult to be estimated. One solution to this problem is to model the needed interactions inside the symbolic execution. The downside of this is a more complex system and more effort.

2.2.6 Summary

In summary the two analysis concepts, symbolic execution and abstract interpretation, may seem similar but are two completely different approaches in program analysis:
Symbolic execution is a more static and theoretical approach of program analysis, the input parameters are symbolic and will be defined depending on the desired path within the analysed program.
Abstract interpretation on the other hand is a more dynamic approach using fixed abstract values derived from actual values or value ranges with a specified abstraction function as inputs. These abstract inputs are executed by an abstract control flow to again get an abstract output, which finally may be simplified but still has necessary information to conclude the behaviour of the actual program from its abstract implementation.

2.3 Cryptography Analysis

This section gives an overview of symbolic analysis in cryptography, why it is needed, and what research has been done. It explains the concept of ciphertext indistinguishability, cryptographic primitives and protocols, symbolic models, and computational models and the relationship between them.

2.3.1 Introduction

Cryptography is omnipresent in the modern society and many older implementations of cryptography have been proven to be broken (GSM[3], DECT[4]). It is very difficult to prove a cipher to be secure. It is even more difficult to prove several ciphers, which are used in

[3] `https://en.wikipedia.org/wiki/GSM#GSM_security`
[4] `https://en.wikipedia.org/wiki/Digital_Enhanced_Cordless_Telecommunications`

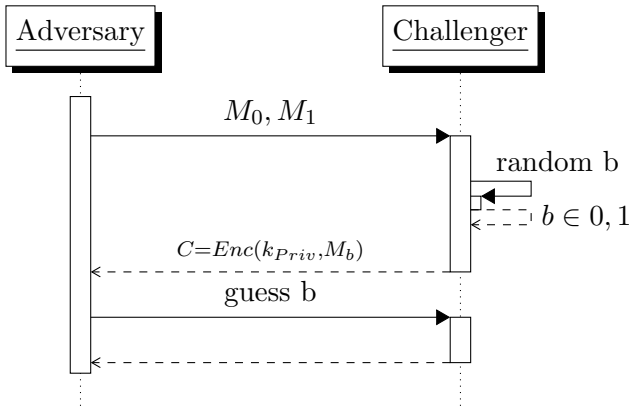

Figure 2.1: Sequence diagram of IND-CCA

complex computer programs or libraries like cipher suites, to be secure. If there is no proof of broken security now, there is no guarantee that there will be no proof in the future. So there is a need of strong security guarantees.
The best way to get the highest assurance of the security of a cipher is open competition and proof of security by as many independent people/institutes as possible. The problem is such proofs of security are very time consuming and there is no standard for it, even if it is machine assisted. Another approach for easier analysis is making the proof modular, which means trying to divide it in individual smaller tasks [15].

2.3.2 Ciphertext Indistinguishability: IND-CPA and IND-CCA

To properly analyse the security of a cryptographic operation and its produced ciphertext some requirements have to be defined that have to be met during the analysis. These requirements are not only related to the ciphertext but also to its relationship with the plaintext. The first definition to be made is the *Indistinguishability* (IND) of the ciphertext, which means an adversary cannot get any information of the plaintext when given the calculated ciphertext, except the adversary guesses randomly, which requires an unlimited amount of computational power and is therefore allowed.
One step further the adversary is allowed to choose the plaintext and can calculate its ciphertext. When analysing public-key cryptography this is fulfilled if the adversary obtains the public key. The adversary now chooses two messages freely and submits them to a challenger. The challenger encrypts one of the two messages and returns it to the adversary, if the adversary can calculate the correct message, out of the two he has given the challenger, related to the ciphertext more than half the time the algorithm is not **indistinguishable under chosen plaintext** (IND-CPA). This is the first property of security of encryption schemes, a sequence diagram can be seen in figure 2.2.

Another harder property is **indistinguishability under chosen ciphertext attack** (IND-CCA). The adversary is, additionally to his capabilities as defined in IND-CPA, able

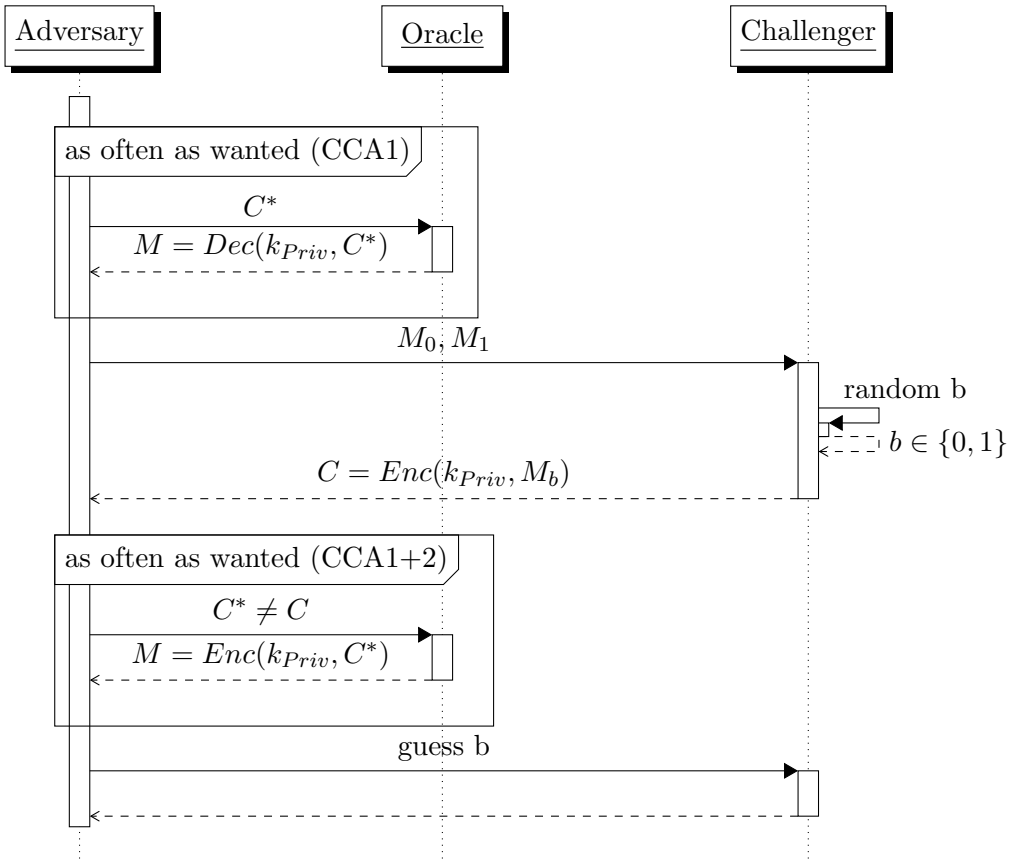

Figure 2.2: Sequence diagram of IND-CPA

to use a decryption oracle which can decrypt any ciphertext except the one given by the challenger. This oracle cannot be analysed by the adversary, and therefore the private key cannot be extracted. This attack is further separated into two sub-categories, dependent on the time the adversary can use the oracle. It is divided into **indistinguishability under chosen ciphertext attack** (IND-CCA1) and **indistinguishability under adaptive chosen ciphertext attack** (IND-CCA2). The modifier sequence diagram can be seen in figure 2.1 [16].

Cryptographic Primitives and Cryptographic Protocols

There is a differentiation between cryptographic building blocks (or cryptographic primitives) and cryptographic protocols:
Cryptographic primitives are basic cryptographic algorithms like simple ciphers, symmetric encryption, public-key encryption, hash algorithms, or signatures.
Cryptographic protocols are implementations which use a combination of the primitives as building blocks to implement a secure way of communication or storage of data. There is no guarantee that the protocol is secure if the primitive is proven secure.

To analyse the security of the combination of the used building blocks in the cryptograhic protocols an abstract model is introduced, which is called **symbolic model**.

Symbolic Models and Computational Models

Research defines two informal views of cryptography, the symbolic and the computational model.

Symbolic Model

The symbolic model sees all cryptographic operations as formal, the cryptographic primitives are seen as blackboxes. The operations are represented by expressions, for example a ciphertext may be represented as $(M)_k$, where M and k are formal expressions, which cannot be extracted from the ciphertext $(M)_k$. Other functions can be applied to these expressions resulting in other expressions, for instance there can be a function which computes M from $(M)_k$ and k. The security properties are modelled and not defined.
This formal perspective is very easy for designing protocols and understanding them. It leads to simple models which can be checked easily by an automatic process, but does not guarantee a secure implementation [17].

Computational Model

The second view of cryptography is the computational model, these models are more accurate defining a lower level of abstraction and adversaries with defined tools trying to break secret protocols, the definitions of the abilities of adversaries are explained in section 2.3.2. The messages are defined as bit-strings, the security is defined by difficulty an advisory has to obtain secret data using computational tasks like factoring or solving discrete algorithm challenges.
Proofs using the computational model are much more realisticthan the symbolic model but cannot be obtained automatically, therefore have to be made manually [17].

The relation between this computational and the symbolic model has been researched by Abadi and Rogaway [17] and is called **computational soundness**. To widen the limited application of computational soundness, which can only be applied on a small number of primitives, **deduction soundness** has been introduced by Cortier and Warinschi [18] and further advanced by Böhl, Cortier and Warinschi [19].

Chapter 3

Concept and Design

This chapter gives an extensive introduction to the concept of a secure program in section 3.1. It then introduces the security guard in section 3.2, explains the detailed motivation behind it, and the requirements which are to be met during design. Afterwards a discussion of the the specific ideas to fulfil the requirements defined in can be read. At first it will be discussed where in the execution flow of the secure program the security guard should be implemented. Then fundamental design elements are introduced, like *secret levels* in section 3.5, the *symbolic execution* in section 4.7, or *data types* in section 3.9. Additionally, some challenges, which occurred during the design phase, are discussed, like the uniqueness of symbolic fields in section 3.7. Lastly, the requirement of a guaranteed termination with defined results in section 3.12 is discussed.

3.1 Secure Programs

As already explained in section 1.1, a secure program acts as an executable record to a high security device. It contains all instructions and memory organisation information to be able to process the instructions it contains and provide the desired result in a specified output. It has several sections, which can be split into the three major groups: fields, instructions, and output. The fields section (marked blue in figure 3.1) declares predefined parameters, e.g. constants, which are read only, defined fields for saving intermediate data, and secret fields with external secret data material, for instance keys. For more details see section 3.1.1. The instructions section (marked orange) is holding a set of references to instructions supported by the high security device implementation and references to fields as input and output data, which are executed on the high security device, for more information see section 3.1.2. The output section (marked green) is defining the data which is returned to the caller of the secure program and therefore leaves the more secure environment of the high security device, for more information see section 3.1.3.
Figure 3.1 shows the structure and relationships of the different sections of the secure program which is explained in more detail in the next sections.

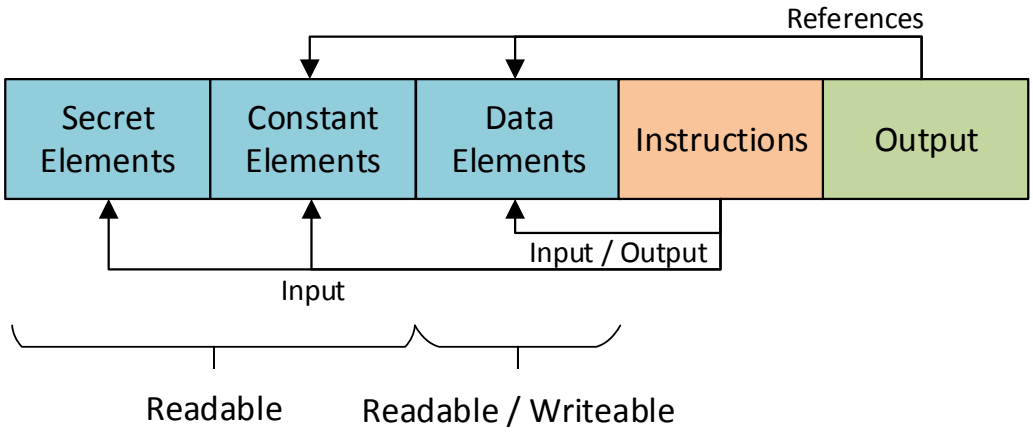

Figure 3.1: Structure of a secure program

3.1.1 Fields

Fields are the defined memory structure in the secure program. One field has a fixed length, a fixed field type and an identifier so the content can be accessed by the defined instructions in the secure program. They are allocated before the execution of the secure program and some are filled with needed predefined constant data.
There are three different kinds of fields:

- **Constants:** Are initialised and filled during the creation of the secure program. These fields can be read from only and are classified as "no secret" (see section3.5).
- **Data Fields:** These fields are for storing all data, that is produced during the execution of the secure program. Intermediate and final results are written to and read from these fields by the instructions. Data fields with final results are additionally referenced in another section, called output section of the secure program. These fields are allocated before the execution of the secure program and are initialised with "0"-bits.
- **Secrets:** Are initialised during secure program's creation with basic information about the secrets' data, i.e. the type of the secret (Public/Private Key, Symmetric Key, etc.) or algorithm type. The actual secret will be provided to the high security device during execution in an additional record. The data of this record is encrypted for and can only be obtained by the high security device. Each secret field can be hierarchically classified by a pre-defined secret-level (see section 3.5). Secret fields are read-only and therefore cannot be altered.

3.1.2 Instructions

The high security device has a fixed set of functions, which are called by the secure program. These functions are called instructions and have a unique identifier and a strict set of input and output fields or parameters in a static order. Input values can either be data parameters, like integers or strings, or references to any field (secret, data, or constant) in the fields section in the secure program. Output values are always references to data fields where the (intermediate) output data gets written to. An input or output field reference can address a whole field or just a sub-range of a field with an offset and a size parameter. Some instructions may prevent the usage of sub-range references. The instructions are executed in a linear order on the high security device, this means there exists no control flow in the instructions (i.e. labels, branching or loops).

3.1.3 Output Section

After the execution of all instruction of the secure program, results are specified by a specific output section, referencing the fields, which are returned by the high security device. This section is also defined in the secure program and can reference only data or constant fields. Because only the last data written to data fields at the end of the execution is returned, intermediate data does not need to be considered published and therefore can be ignored. At the end of the execution the referenced fields are encrypted and published to an output.

3.2 The Security Guard

From the explanation given in the previous sections it was shown that a secure program is a powerful tool to process sensible and confidential data. But with great power comes great responsibility: the author of such a secure program can, deliberately or by mistake, reveal confidential data to the output. The security guard has the purpose to detect such disclosure of confidential data. As a positive side effect the security guard can also reveal possible mistakes from the author of the secure program, for example, using a key for a cryptographic operation it is not intended for. Such a mistake would be detected much later during usage of the output data, if at all.

Here are some requirements the security guard is designed for:

- Every unique secret key must be identifiable and copy of keys must show their relationship to each other.
- No direct reveal of secret data by publishing parts or all of it on the output section.
- Correct usage of intermediate data output of instructions, which are input parameters to other functions. For instance, when generating elliptic curve keys the type of

the elliptic curve the key is generated for is also saved and can be checked when given to an elliptic curve encryption operation against the curve type of the operation.

- No leakage of secret data: indirect information flow, as described in chapter 2.1, can cause that an adversary is able to get information about secret data. An example would be calculating a CRC value of a key and publishing all or parts of the resulting CRC value without encrypting it before.
- Introduction of a "secret hierarchy" called secret level, where a secret can only be encrypted by another secret with the same or higher secret level.
- Every key which encrypts another key needs to be added to some type of dependency structure in order to prevent cycle encryption (see section 3.10)
- Keys calculated via key agreement with the same parameters must get the same unique identifier because it is basically the same key and should be handled like a copy.
- Sub field support, for instance, to generate a certificate in a field build from other fields.

3.3 Concept of the Security Guard

To securely process confidential data in an automated way a high security device gets a special secure program, which has a fixed set of *instructions* and a determined set of memory, partitioned and defined by *fields*. These fields and instructions were explained in section 3.1. The secure program, together with encrypted secret key material generated from another source, called Long Term Secrets, is loaded on a high security device where it then decrypts the needed data and processes it according to the instructions defined in the secure program. See figure 1.1 for an illustration of the described concept. To ensure the secret data from any involved party stays protected during execution on the high security device and the final processed data of the finished script is protected in a secure way, a verifying tool, called security guard, is introduced. This security guard implements a symbolic representation of the actual environment executing the secure program, and checks for possible security violations. A violation can be caused by an implementation mistake or a deliberate attack of an adversary. The security guard acts as a safeguard and hinders a possible revelation of security relevant data when the secure program is executed on real data.
The security guard could be applied in parallel to the actual execution of the secure program, or during the initial generation of the secure program before its actual execution on real data. For a discussion of the advantages and drawbacks of both implementations see section 3.4. The decision in this implementation was in favour of a one time security guard execution before the actual execution with real data.

3.4 Order of Execution

The first considerations during the design of the security guard were regarding the order of the execution: A secure program is designed to be executed more than a million times and on many high security devices in parallel. In the end two possible approaches, with their own advantages and drawbacks, exist to conveniently check the security of the secure program. Both of them are discussed in the next two sections.

3.4.1 Parallel Check during Execution

The first variant is to check the secure program parallel to the actual execution. The execution environment of the high security device gets extended with the implementation of the concept of the security guard, meaning for every instruction supported by the secure program the environment is extended with an additional symbolic execution and further control mechanisms. If a violation of usage or a security breach is detected the execution gets aborted, all intermediate data gets deleted, and no data at all gets returned to the output. An illustration of the concept of a parallel check can be seen in figure 3.2. Green operations are already implemented in the environment for the secure program execution. Yellow operations must be newly implemented and integrated.
The advantages of this design is that the high security device can be designed stateless. The effort for an adaptation is negligible with a low impact on the existing infrastructure and processes. Furthermore, changes in instructions require editing in just one part of the execution environment. Additionally, if checks on the actual data, in contrast to the symbolic data, are required, it can be accessed easily.
On the other side the drawback of this design is the performance of the executions of the secure program in general. As long as no checks in run-time values are needed, the security guard must check the secure program just once. Therefore, given the script does not change, the execution of the security guard on the same secure program becomes redundant. This means with 10,000 executions, 9,999 executions of the security guard on an unaltered secure program are not necessary. In the performance analysis in section 5.2 it is shown that the execution time of a secure program for a complex product stays below 100 milliseconds, this is a high estimation for demonstration purposes. With this estimation, one million executions of the same secure program would take additional 100,000 seconds, or more than 27 hours.
The effect on memory usage is not as high when using parallel checks, the memory will be freed after every single check and therefore will not stack up. But the entire execution, symbolic and actual, needs more memory, which can lead to problems on high security devices with restricted memory.

3.4.2 Separate Check and Execution

The second variant is a separate loading of the secure program with a one-time check of the security guard with symbolic data which then, if no rule is violated, gets distributed and

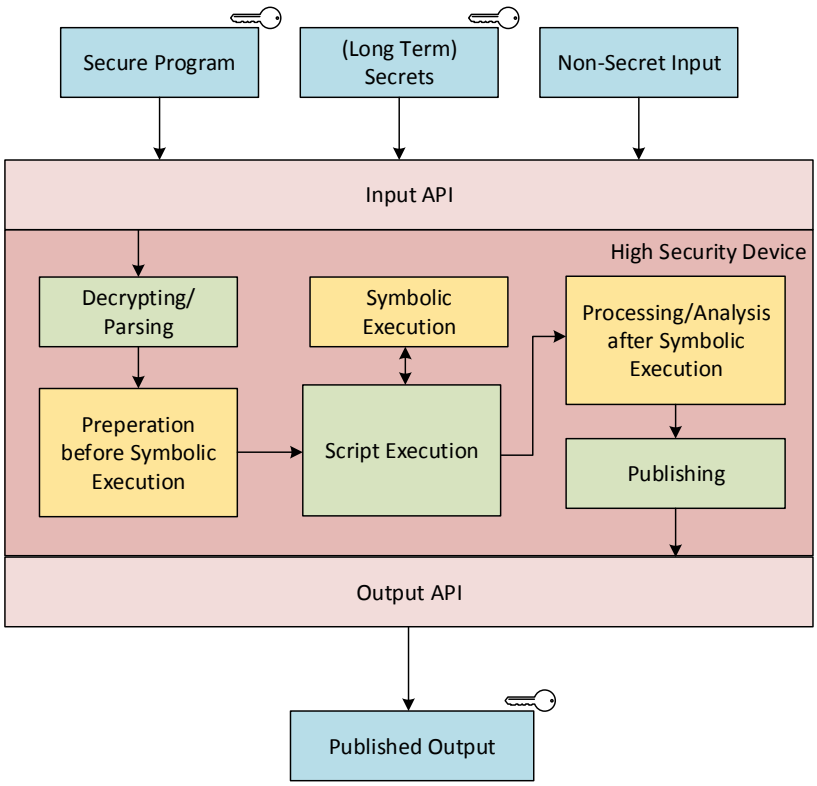

Figure 3.2: Parallel check during execution

executed by the high security devices with real data. The high security device making the check can, but not necessarily has to be the same, physical module. Figure 3.3 illustrates the concept of a separate check with a subsequent execution.
This approach has the advantage of having the security guard executed just once, an impact on performance is therefore negligible.
A drawbacks is an alteration of the existing, already established, processes. Especially the design of a stateful implementation of the execution environment of the high security device to get a comprehensible script, which can be executed symbolically beforehand. Lastly, checks on real (intermediate data) is more complicated, if not impossible.

In favour of performance the second design has been chosen and is discussed in this thesis.

3.5 Secret Levels

Another consideration is to achieve a specific hierarchy within secret (key) material. For instance, a secret can be generated by other parties and imported from outside the high security device, called Long Term Secrets, or generated during execution of the secure program, called Generated Secrets. To get a hierarchy into the different secret origins, **secret levels** are introduced. In this example, two different secret levels are in place, but it can be extended to as many as there are needed.
One of the main targets accomplished by introducing secret levels is that a secret with a higher secret level cannot be encrypted using a secret with a lower secret level. Secrets which are not generated during the execution of the secure program on the high security device, but come from the outside, have to have an according (higher) secret level assigned, and therefore cannot be encrypted by a generated secret during secure program execution, because of it's lower secret level.

3.6 Symbolic Execution

Before the high security device may use a secure program it needs to check its security properties through a symbolic execution. A symbolic execution is a mechanism to check high-level properties (e.g., on data flow) of a program without actually executing it. The fact that the secure program is a linear, meaning non-branching, program and that the set of instructions is relatively small makes a complete symbolic execution feasible on a high security device.
The symbolic execution implements all instructions of the secure program in an abstract way considering data flow and surveying confidential data. These symbolic instructions consist of three major parts:

- A **requirement check** verifies the inputs of the instruction, whether data types are correct and if the given parameters are sound with regard to the given input fields especially when handling key fields (see section 3.10 for more information).

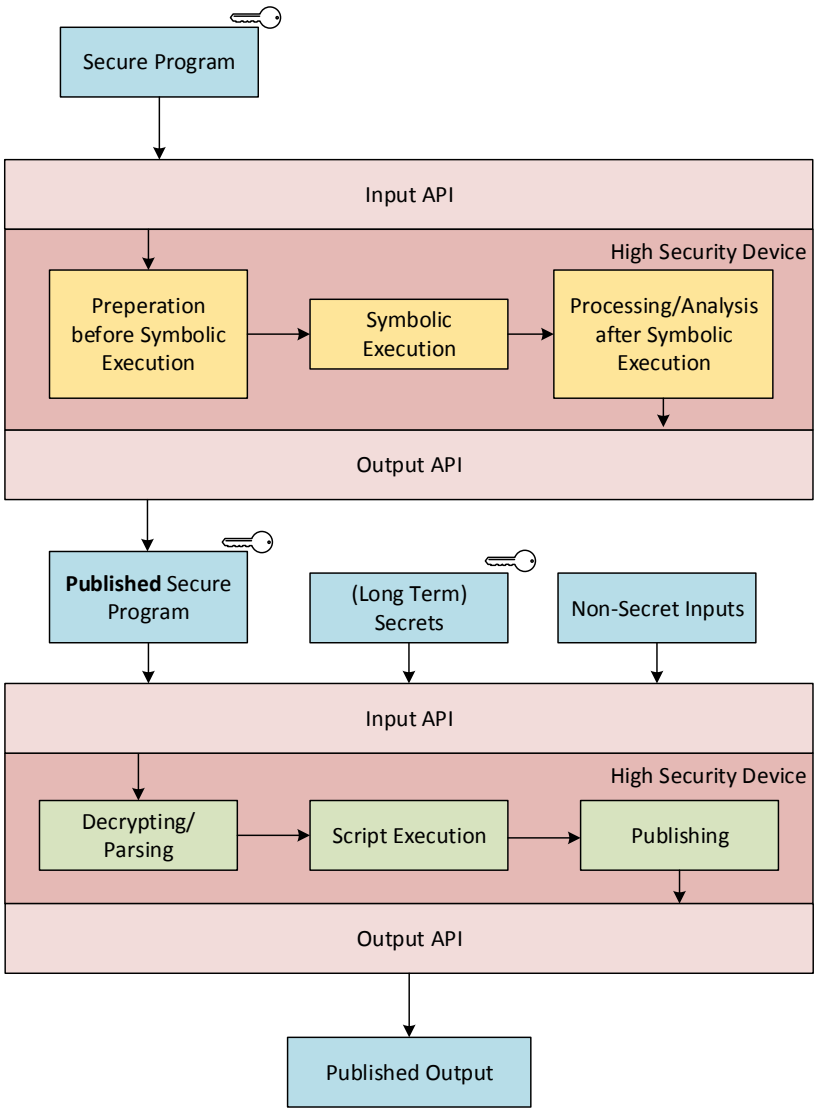

Figure 3.3: Separate check and execution

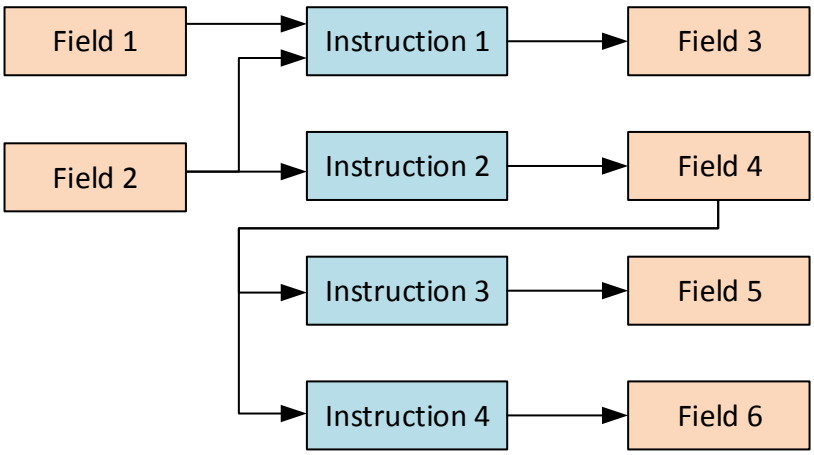

Figure 3.4: Structure of the symbolic execution

- The actual **symbolic execution** of the instruction. It determines the correct data types of the output and specifies additional data in the output fields which may be needed later in the process. Additionally keys of cryptographic instructions will be added to a hierarchical structure according to the usage (see section 3.10 for more information).
- The **Confidentiality** functions and parameters are set accordingly in the output fields. These are used at the end of the symbolic execution of all instructions in the confidentiality check (see section 3.8.6).

Figure 3.4 gives an idea of the structure of the symbolic execution of a secure program with 4 instructions and 6 fields. The fields can be constant, secret, and data fields of the secure program defined in section 3.1.1. Each instruction is executed in linear order, from first to last, and may take any of the already existing fields as a parameter. Furthermore, each instruction writes to at least one data field, the output. The fields during the symbolic execution can be read (used as a parameter) multiple times, but, in contrast to the actual execution of the secure program, only written to once. The challenges caused by this restriction and solutions introduced can be found in section 4.6.

3.7 Uniqueness of Symbolic Fields

When executing the secure program in a real scenario, the instructions have references to fields, that are specified in the secure program itself and moreover are re-using fields and sub-fields. It is allowed to overwrite the specified fields, or sub-fields, with previous information. Unfortunately the symbolic execution cannot adapt this behaviour and every instruction in the symbolic execution must have its own unique output field in order to

have a comprehensible history of fields when checking confidentiality levels (see section 3.8) after the symbolic execution.

In other words, during real execution of the secure program an instruction can write a result into an output field, some other instructions then use this output field as input and then another instruction overwrites parts or the whole output field with new information, which then gets used by other instructions. But during the execution of the security guard on the secure program all fields and their symbolic information have still to be available after symbolic execution while publishing the output as explained in section 3.8.6.
For example, a field A gets written with a secret key, then field A gets copied to another field B. Field B is now dependent from field A with its confidentiality functions. Now field A gets overwritten with random data. If the fields are not unique and the content of the old field A gets lost when overwriting it, the whole concept of the confidentiality check does not work.

3.8 Confidentiality Levels

Secret data can be disclosed, may it be willingly or by accident, in two different ways. One way is to encrypt it in an insufficient way, for instance using inappropriate key lengths or weak encryption algorithms. The other way is by publishing data derived from the secret material by leaking it through indirect information flow. To ensure secret data is appropriately encrypted and no indirect information is leaking by publishing derived data without an appropriate encryption also, confidentiality levels are introduced.

Confidentiality levels are numbers with a specific range set between 0 and a maximum value. The value of the confidentiality value provides a lower bound on the security measures in bits, which is the expected number of operations required for breaking a specific cryptographic cipher text depending on the encryption operation used. For example, if the confidentiality level is 100 an adversary has an expected effort of 2^{100} operations to figure out the secret data. The determination of the confidentiality levels, dependent on the used cryptographic operation, is shown in section 3.8.3. These initial confidentiality levels are assigned to the field with the secret while executing the security guard. Secrets coming from outside the high security device must have specific meta data about the type of the secret, for instance the algorithm type and characteristics of the algorithm, in order to determine the correct confidentiality levels. This meta data is then used to calculate the confidentiality value before the execution. When generating a new secret during the execution of the security guard, the confidentiality level is determined during the instruction and assigned to the output field of the instruction.

The confidentiality level of a field is actually the minimum of two confidentiality levels: The confidentiality level considering direct leakage and the confidentiality level considering threshold leakage. These two concepts and the specific difference between both are

explained in section 3.8.1 and section 3.8.2.
During symbolic execution the output fields of the instructions get specific confidentiality update functions, defined by the symbolic instructions, assigned. If the field gets published (meaning written to the output of the high security device) at the end of the symbolic execution of the secure program, these confidentiality update functions get called as explained in section 3.8.5.

In the next sections a more mathematical way is used to describe the manipulation and usage of confidentiality values to recognise leakage of secret data.

3.8.1 Confidentiality Level considering Direct Leakage

The confidentiality level considering direct leakage ($conf_{DL}$) of a field F is the confidentiality level affected by data which is derived (and therefore leaks data) from this field and lowers the confidentiality level of F by a certain amount L, the leakage. The operation which is given to the derived field is denoted as Leak(F, L) in the symbolic execution. This kind of leakage can also be called **"direct leakage"** since it directly helps the adversary. For instance, publishing the CRC-checksum computed over F directly leaks 32 bits of information about F's value in a worst case scenario if the calculated checksum is written to the output at the end of the symbolic execution. The confidentiality level considering direct leakage has a lower bound of 0, meaning once it reaches 0 all further leakage is ignored. In contrast to the direct leakage, there is also threshold leakage as explained in section 3.8.2.

3.8.2 Confidentiality Level considering Threshold Leakage

The confidentiality level considering threshold leakage ($conf_{TH}$) of a field F is the confidentiality level which gives an upper threshold of the security on published information that adds a lower bound B for the confidentiality level of F. Hence it is denoted as AddConfLB(F, B) in the symbolic execution. Note that if publishing more than one field with the same secret's content, setting a new lower bound to the secret's field, AddConfLB(F, B) does not change the value of the confidentiality level considering threshold leakage if the old value is already below that, since it only adds an additional lower bound. This kind of leakage is called **"threshold leakage"** because it directly helps the adversary only after performing an expected amount of $2^{conf_{TH}}$ operations; however, the assumption is that the adversary learns the complete value of F once he invests at least $2^{conf_{TH}}$ operations. For instance publishing the output of encrypt_symmtric(K, F) for a key K that provides a security level of 80 bits sets the threshold leakage of F to 80 (AddConfLB(F, 80)). $conf_{TH}$ has a natural lower bound of 0.

After the symbolic execution each output field of the secure program will get a confidentiality level of 0. For each field the confidentiality update section of the instruction that produced the output in that field dictates how the confidentiality level is propagated

to the inputs, i.e., how the confidentiality level of the inputs changes based on a change of the confidentiality level of (one of the) outputs.

3.8.3 Initial Confidentiality Levels

The confidential level ($conf$), as already described in section 3.8.2, is defined as the number of calculations an adversary has to perform to get to know confidential data. This section defines the confidentiality levels of specific widely used security algorithms. The figures are based on the current recommendations from established government organisations such as NIST (US) and the BSI (Germany) [20].

The defined initial confidentiality levels are:

- **AES128:** 128
- **AES192:** 192
- **AES256:** 256
- **2TDEA:** 80
- **3TDEA:** 112
- **ECC:** For elliptic curve cryptography the confidentiality level is the bit length of the modulus of the underlying field (also often referred to as the size of the curve).
- **RSA:** Dependent from the modulus length, the confidentiality levels are as follows:
  - $0 \leq$ Modulus Length < 1024: 0
  - $1024 \leq$ Modulus Length < 2048: 80
  - $2048 \leq$ Modulus Length < 3072: 112
  - $3072 \leq$ Modulus Length < 7680: 128
  - $7680 \leq$ Modulus Length < 15360: 192
  - $15360 \leq$ Modulus Length: 256

3.8.4 Getting the Confidentiality Level of a Field

Every Symbolic Field F holds a ConfDL(F) and ConfTH(F) operation and therefore every instruction specifies a ConfDL(F) and ConfTH(F) operation for each of its output fields F. A field is an output field of an instruction if the instruction writes to it. The ConfDL(F) and ConfTH(F) operations provides the current confidentiality level (considering direct leakage or threshold leakage) of the corresponding output field. This value may depend on the confidentiality level of other fields, e.g. inputs of the instruction (which, in turn, may be output fields of other instructions; to learn their current confidentiality level, the corresponding function is called for them). Some exemplary symbolic instructions and it's confidentiality functions can be seen in section 4.10.7.

3.8.5 Updating Confidentiality Levels and Confidentiality Checks

Similarly to section 3.8.4, every instruction defines a Leak(F, L) and a AddConfLB(F, B) operation for each of its output fields. These operations describe how direct leakage / threshold leakage is propagated to the inputs of the instruction.

3.8.6 Confidentiality Check

Once all instructions have been executed, the symbolic execution performs the confidentiality check. It calls for each output field F (i.e., a field referenced in the output section) a specific publishing function called Publish(F). This Publish(F)-function itself calls Leak(F, Max), where Max is the maximum confidentiality level and AddConfLB(F, 0), and thus effectively sets the confidentiality level of the Field F to 0. If one of these fields is the output of an instruction, this often leads to further invocations of Leak and AddConfLB. In this way the confidentiality level of zero from the published fields is propagated to other fields. Once the confidentiality level of all output fields is set to zero, all guards collected (see 3.8.7) must be executed until the confidentiality levels of all fields do not change any more. The symbolic execution has now computed the final confidentiality levels for all fields.

A confidentiality check can fail in two pre-defined ways:

- For external secrets, which are static and can be reused in other executions of the secure program, the confidentiality check fails if the confidentiality level is lowered at all (in this case the confidentiality check can be aborted immediately). The reason for this is because the confidentiality level can get lower each time the secure program gets executed giving an adversary more information about the secret data and there is no possibility to check how often a secure program gets executed.

- For generated secrets during execution of the secure program the confidentiality check fails if the confidentiality level drops below a defined threshold. Generated secrets are randomly generated during each run of the secure program and therefore its confidentiality level cannot get lower than calculated during one execution, even if the secure program gets executed more than once. If the secure program gets executed again, a completely new secret is generated.

3.8.7 Guard Functions

For some instructions, the confidentiality level of the output depends on future/final confidentiality levels of other fields. For example, for a symmetric encryption operation the confidentiality level of the message depends on the confidentiality level of the key, which may be altered after the encryption in the symbolic execution. For scenarios like this, guards are needed.
A guard is a function that needs to be called once a value changes (it monitors that value).

Hence, a guard always is defined on the alteration of specific values of secret fields, for instance every time the confidentiality of a key changes it's guards must be called. A guard monitors the output field(s) of the instruction it belongs to in the sense that it keeps their properties up to date (usually their confidentiality level, to be more specific).
Optionally, a guard may also be called if the value it monitors did not change.

3.9 Data Types

The symbolic execution assigns a data type to every field in the secure program. This allows it to check the type requirements of instructions.
The main purpose behind the data types is to prevent misuse of data with an impact on security. The most important points are:

- Keys on elliptic curves must only be used with the curves they have been generated for. If they are interpreted as points on a different curve and used for cryptographic operations it is not clear whether security guarantees still hold.
- Symmetric keys must only be used with the block cipher (currently supported are AES and TDEA) they are intended for.
- Only key material can be used as the key input for cryptographic operations.
- Decryption works only on ciphertexts of proper type. Otherwise, encrypt(K, decrypt(K, D)) for arbitrary data D might be falsely considered as a proper ciphertext of D under key K while, e.g., for AES, it is just D.

If a supertype is required from a symbolic instruction, all its subtypes of the supertype are accepted as well. Some data types have additional information to be saved, for instance the curve type of an elliptic curve key or the algorithm type of a symmetric key. The symbolic execution sets specific values for these parameters once it assigns such a data type to a field. Parameters allow a more concise description of requirements. The alternative would be to have additional subtypes corresponding to all possible parameter values.
The introduced data types to implement the symbolic execution of the actual secure program are:

- **Secret Data (SData):** See section 3.11.2 for more information. SData has the subtypes:
  - **Long Term Secret Data (LTSData):** Secrets coming from outside the secure program
  - **User Generated Secret Data (UGSData):** Secrets created during execution
- **Data**

  - **Public Key**
    - **RSA Public Key**
    - **ECC Public Key:** with the curve type as additional parameter
  - **Private Key**
    - **RSA Private Key**
    - **ECC Private Key:** with the curve type as additional parameter
  - **Symmetric Key:** with the algorithm type (TDEA or AES) as additional parameter
  - **Hash:** with the algorithm type (for instance SHA-3) as additional parameter
  - **Signature**
  - **Symmetric Cipher Text:** This is data which was encrypted during execution of the secure program. Additional data saved with this data type is the key encrypting this cipher, the encryption mode used, and the input field, which got encrypted.
  - **Asymmetric Cipher Text:** With the same additional data as Symmetric Cipher Text
  - **Random**

3.10 Key Identification

Keys for cryptographic algorithms, both symmetric and asymmetric, must have a secret level and a key ID. The key ID helps to uniquely identify the key and to determine copies of a key, which must have the same key ID as the original key. With help of the key ID a copied key can be assigned to the original key and its secret specific values can be determined and altered if necessary. If a leak happens on a copied key, the confidentiality data of the original key has to be updated and if the key or any copied key is published to the output the original and all copied keys must be considered on possible leakage. Additionally, if a key encrypts another key, the encrypted key is saved as a lower key of the encrypting key to identify and omit cycle encryption.
The symbolic field of a key holds the key specific data type and may also hold data type specific additional information about the key, for instance elliptic curve parameters or symmetric encryption type. If a key is coming from outside of the high security device, the data has to be given via some type of additional data to the security guard.

3.11 Sub Fields

Instructions may address a whole field or, if permitted, a sub range of a referenced input or output field. A field itself, however, can only have one data type, key ID and other field specific information. The idea of a complex system which oversees the exact data of

sub-fields has been discarded to have a simpler symbolic model which is easier to maintain and bugs and errors can be analysed and handled in a more convenient way.
Therefore, some restrictions have to be defined to still have a safe and sound symbolic execution. The next section introduce these restrictions.

3.11.1 Full Secret Field References

From a security perspective there is no reason to use just specific parts of a key. Therefore, every secret field, which is defined as key with a key ID, and every Secret Data (SData) with its subtypes is only allowed to be referenced as whole field, meaning sub-field references to this fields are prohibited.

3.11.2 The SData Type

To separate normal data from data, that has secret material in it, the SData type was introduced in section 3.9. This special secret data has no key ID, and thus no additional information like original key field or secret level, therefore it cannot be used for most cryptographic functions. It is composed of two subtypes for data of secrets created during execution of the secure program called User Generated Data or **UGSData**, or data coming from outside the high security device called Long Term Secret Data or **LTSData**. If non-secret data is concatenated with a secret data field, the new data type of the output is, depending on the secret level described in section 3.5, either UGSData or LTSData. When an UGSData field and an LTSData field are concatenated, the output field must be an LTSData field.

As a summary, an enumeration of the hierarchy of concatenated secret data fields is given:

1. **LTSData**
2. **UGSData**
3. **SData**
4. **Data**

A concatenated field can only go upwards, but not downwards in this hierarchy. Additionally, the secret level of a key field has always to be considered, i.e. the output field of an UGSData field concatenated with a long term secret key must be an LTSData field.

3.11.3 Sub Field Output

If an instruction during the execution of the real secure program has a sub-field reference to an output field with previously written data to, this referenced field gets partially overwritten. It may be the case that, if sensible data was present in this field, this sensible

data got overwritten, but in the worst case the previously written sensible data is still in the field. To cover this case the worst case scenario always is expected during symbolic execution, which means the new field must be the "virtual" concatenation of both, the symbolic output field of the current instruction, and the symbolic field with the information of the previous data in the field which gets overwritten. A sophisticated system will be introduced in section 4.10.6 which implements this design criteria.

3.12 Termination and Result

The symbolic execution terminates as soon as it has reached a stable state, meaning all confidentiality levels (see section 3.8) of all output fields are stable. Stable means that the confidentiality level of all output fields has been set to zero (as stated in 3.8.6), the changes are propagated to all dependent fields, and the guards collected during the symbolic execution finally finished changing the confidentiality level of any field.
The symbolic execution of the secure program always terminates: the number of instructions in the secure program is finite and every instruction operates only on a finite number of fields, therefore the number of fields in the symbolic execution is finite. Since the confidentiality level of a secret field is confined to a value between zero and the maximum confidentiality level, the overall state space is finite. Because confidentiality levels can only drop, the symbolic execution cannot return to an earlier state with different confidentiality levels. Or, in other words, the sum of all confidentiality levels can only decrease. The number of guards is finite, therefore the state of the symbolic execution can only remain unchanged for a finite number of steps without causing the symbolic execution to terminate, the state must eventually change (or the symbolic execution terminates). The sum of confidentiality levels can only drop and has a lower bound of zero, as a result the symbolic execution always terminates.

Chapter 4

Implementation

This chapter introduces the implementation of the security guard. The first sections are arranged by the actual phases of the security guard. Afterwards, in section 4.10, advanced concept details are explained.

4.1 Development Environment

The implementation was done as part of a firmware module for a hardware security module (HSM) as high security device and written in the C programming language. Because of the restrictions of the HSM platform and security considerations, no external libraries or code were utilized. During development the Eclipse integrated development environment (IDE) with the C/C++ Development Tools (CDT) extension and to compile the resulting C-source code the GNU Compiler Collection were used.

4.2 Structure of the Implementation

The principle concept of the security guard is described in chapter 3. Only if the checks and requirements during the execution of the security guard are met, the secure program is published and can be actually executed by a high security device. Therefore the security guard acts as an acceptance gate for the creation of a valid and executable secure program. In figure 4.1 the simplified implementation concept can be seen.

The implementation has the following structure and is explained more thorough in the next sections:

- **Preparation:** Properties of external secrets, which are coming from outside the high security device and are imported by the secure program, are parsed and saved into the *imported field structure*. For simplification and convenience reasons this parsing happens before the actual execution of the security guard in parallel to the parsing of the imported secrets for the actual execution of the secure program.

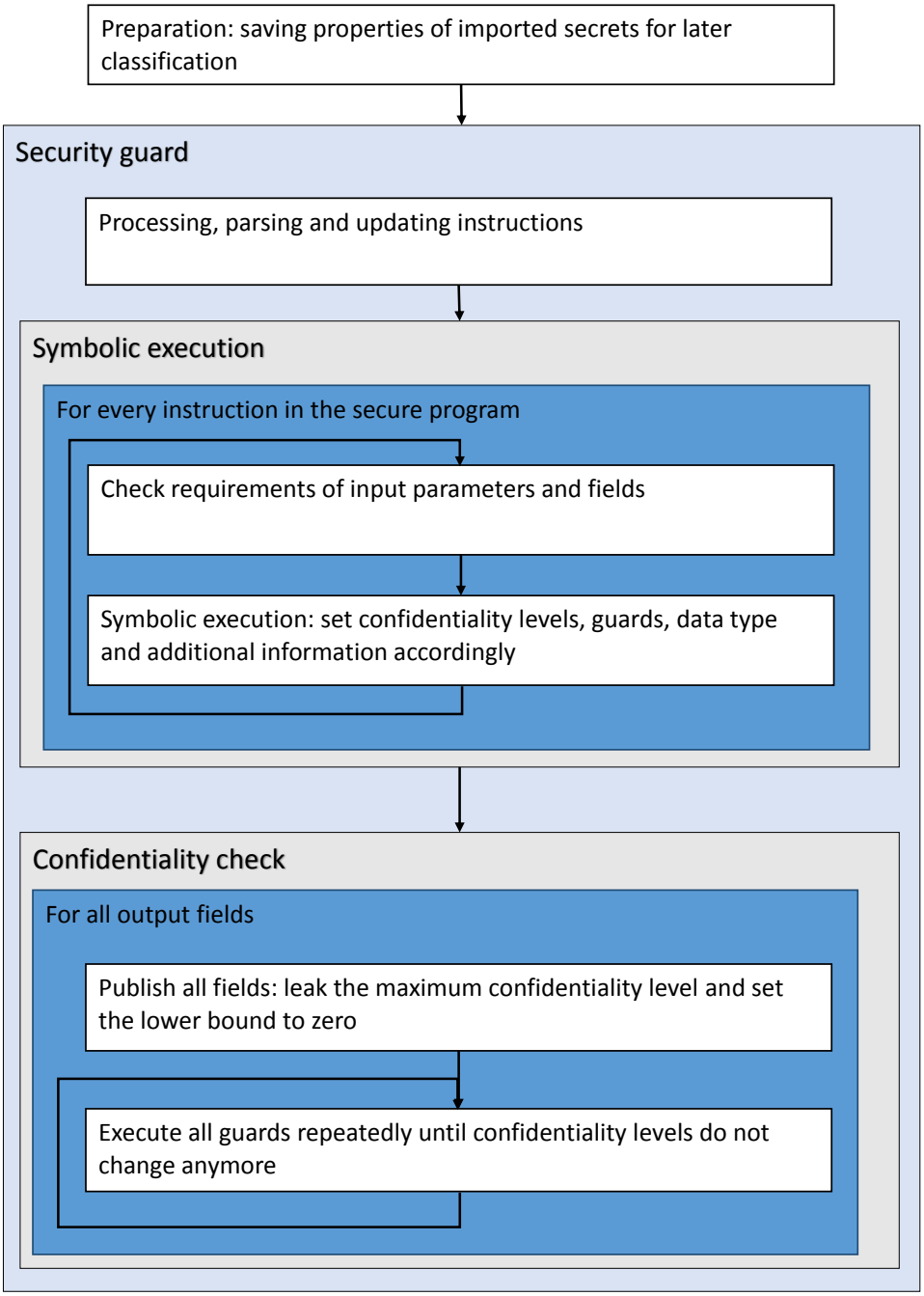

Figure 4.1: Basic structure of the program

- **Secret Data Processing:** This and subsequent operations are part of the security guard implementation: the imported secrets, which were parsed while preparation, are processed and classified, getting correct data types, confidentiality and secret levels.
- **Parsing:** Symbolic data from fields, instructions, keys, and output fields are parsed.
- **Instruction Updating:** As explained in section 3.7 all symbolic fields are, in contrast to the actual execution of the secure program, only allowed to be written to once. In order to ensure the uniqueness of fields referenced output fields of symbolic instructions, which were already written to, are updated with unique output fields for a comprehensible history of output fields to the secrets these outputs are depending from.
- **Symbolic Execution:** For every instruction a check is performed whether all requirements are fulfilled and then the instruction is symbolically executed, updating fields and other structures.
- **Confidentiality Check:** Every output field is published through its confidentiality functions, which were formerly set during symbolic execution. Then the guards, which were also set by the symbolic execution, are processed until no confidentiality change takes place any more. At last, the confidentiality levels of all secrets are checked if still valid.
- **Clean-up:** Freeing of allocated structures and other data used during the execution of the security guard.

The following sections explain the implementation structure of the security guard more deeply.

4.3 Secret Import Preparation

When the high security device parses the secrets imported from outside the high security device, preparing for the actual execution of the secure program, new functionality was added to extract and keep the data needed for classification of the secrets during the security guard execution. A structure is used to collect needed data which will be given to the security guard function call. Storing the algorithm type, and, if needed, additional information about the algorithm, for instance the curve type when an elliptic curve encryption algorithm is used.

4.4 Process Imported Secret Data

The secret data, which is given by the structure explained in section 4.3 is processed and classified in order to store it in the secret field collection with all its information. For

simplicity and maintainability of the implementation secret fields are generated the same way as they are generated during symbolic execution when a secret field is written to an output field of the processed instruction. During generation of a secret field its secret level is determined, the data type is assigned, and, if needed, additional information is also saved in the symbolic field.

4.5 Symbolic Parsing

After importing the symbolic information of the external secrets the secure program itself is parsed. At first, the key collection structure is initialised, then the field and instruction elements are parsed and lastly the output field management is initialised.

4.5.1 Initialising the Key Collection

The symbolic key collection, which is explained thoroughly in section 4.10.4 is initialized. key IDs will be added during parsing the secret field list and symbolic execution in section 4.7.

4.5.2 Parsing Field Elements

A symbolic field collection structure is used during the whole execution of the security guard, managing all constant, external secret, and data fields. Additionally other field related data, like guarded fields, are managed too. It would be impractical to use a normal array to store the fields and guards respectively. An array is not intended to be dynamically extended frequently, for instance while updating the instruction, or when adding single guarded fields during execution of the security guard. As a consequence, a basic linked list implementation has been used to get a simple, dynamically extensible data structure managing all symbolic field elements.

These linked lists managing the symbolic field elements are prepared before filling them with the symbolic field elements in the secure program. The structure of a symbolic field element is explained in detail in section 4.10.1. All constant and data fields are initialised accordingly and added to the fields list. External secret fields are initialized with the parsed data from section 4.4, getting the correct data type, secret level, and, if needed, additional data type dependent information like ECC curve parameters. The key IDs are added to the according secret key list in the key collection.

4.5.3 Parsing Instruction Elements

In the same way the symbolic field elements collection is needed, a symbolic instruction element collection is needed as well to manage the instructions executed by the secure program. Because the instructions do not need to be extended dynamically, a simple onetime allocated array is used holding all symbolic instructions. For more information about the

exact details about the symbolic instruction element see section 4.10.2.

In order to parse the individual instructions in the secure program, the security guard has its own function for every instruction code supported by the secure program to assign the correct references to input and output fields as well as parameters needed for the symbolic execution later. Field references to the symbolic field collection to input and output fields are generated and parameters (algorithm, curve, etc) are saved accordingly.

4.5.4 Output Field Management Initialisation

The output section is a collection of references to data or constant fields which are handed back to the caller at the end of the execution of the secure program. These references are parsed from the secure program and symbolic references are generated in the same way as input or output references are generated during parsing the instructions.

4.6 Update Instructions

After the initialisation and parsing of all needed data (fields, instructions, output) from the secure program, the uniqueness of all symbolic fields, as explained in section 3.7, has to be ensured. Before the symbolic execution all symbolic fields are proven unique, because no field manipulation has happened yet. The only time fields get manipulated during the execution of the security guard is while executing the symbolic instructions when the output fields are getting written to field elements. To ensure the uniqueness of fields is given the following procedure is introduced:
Before the symbolic execution of the security guard, the output field references of every instruction will be resolved and checked if the referenced field elements were used before, e.g. if one ore more output fields already have been written to. This can be detected with a “not used”-tag in the referenced field element. If an output field already has been written to, a new virtual data field will be generated. After generating the new, virtual, data field the following instructions of the symbolic execution, both input and output field references, will be updated with the new data field reference. When the last instruction is checked and, if needed, updated, the uniqueness of every field during the symbolic execution is ensured. This procedure is just applicable to ensure the uniqueness of full output fields. If a sub-field is written to a field, which was previously written to, some additional actions are to be made. These needed actions are explained in section 4.10.6.

4.6.1 Example

Let there be a simple set of instruction. For simplification no instruction number is given and every instruction has one input and one output field. The numbers are the field indices in the reference. Every reference is a full field reference. The example instructions with its references can be seen in table 4.1. To have another representation figure 4.2 shows the instructions with its references in a more graphic way.

Instruction 1	Input: 1	Output: 3
Instruction 2	Input: 2	Output: 3
Instruction 3	Input: 3	Output: 3
Instruction 4	Input: 3	Output: 4

Table 4.1: Tabular representation of instructions before the instruction update

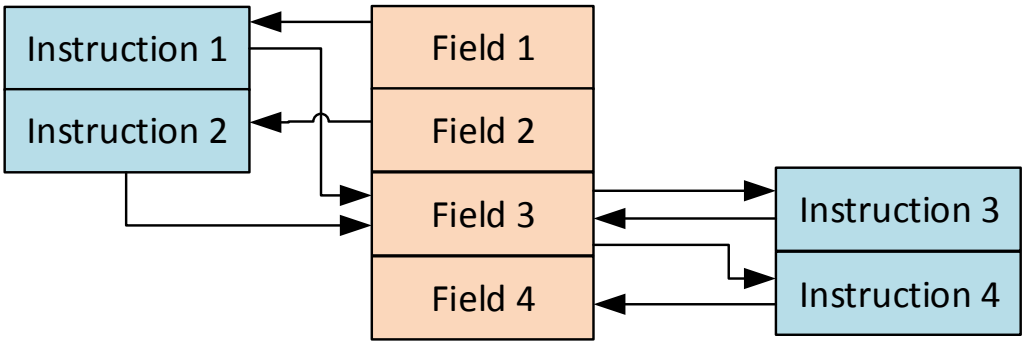

Figure 4.2: Graphic representation at the beginning

The field references of the output of the instruction 1 and instruction 2 are pointing to the same field index 3. When looking at the first index, the "not used"-tag of the referenced field is removed. When looking at the second instruction the referenced output field does not have the "not used"-tag anymore so the reference has to be updated. A new data field will be generated and added to the symbolic field collection. In this case the new data field has the index 5. The output field reference of instruction 2 is updated with the new field index. Then every input and output field reference of the following instructions are investigated, and, if a field has field index 3, it will be updated with the new field index 5. In table 4.2 the changes can be seen.

The new set of instructions, which will be used now is in table 4.3. In the next step instruction 3 is investigated. The "not used"-tag of field 3 is already removed, so again a

Instruction 1	Input: 1	Output: 3
Instruction 2	Input: 2	Output: ~~3~~ 5
Instruction 3	Input: ~~3~~ 5	Output: ~~3~~ 5
Instruction 4	Input: ~~3~~ 5	Output: 4

Table 4.2: Tabular representation of instructions during the first update

Instruction 1	Input: 1	Output: 3
Instruction 2	Input: 2	Output: 5
Instruction 3	Input: 5	Output: 5
Instruction 4	Input: 5	Output: 4

Table 4.3: Tabular representation of instructions after the first update

Instruction 1	Input: 1	Output: 3
Instruction 2	Input: 2	Output: 5
Instruction 3	Input: 5	Output: ~~5~~ 6
Instruction 4	Input: ~~5~~ 6	Output: 4

Table 4.4: Tabular representation of instructions during the second update

new data field has to be generated and added to the symbolic field collection. The output field of instruction 3 is updated with the new field index and all following field indices, input or output, too. In table 4.4 the updates to the fields can be seen.
The final set of instructions with whole unique output fields can be seen in table 4.5. Now every instruction has its own, unique output field, meaning that every field gets only written to once. Nevertheless it is ensured that every input of an instruction has the correct data in relation to the time of execution of the related instruction.
In figure 4.3 the fields and instructions after the update are represented. Newly added fields are represented in red. These updated fields and instructions can now be used for the symbolic execution of the secure program.

4.7 Symbolic Execution

Finally, after all precautions have been done, the main functionality of the security guard can be performed: the symbolic execution. Every instruction in the symbolic instruction collection gets executed step by step in the same way as the implementation of the instruction parsing functions, by executing its own function during symbolic execution. A symbolic instruction consists of the following steps:

- Dereferencing the input and output fields by looking up the index in the reference and fetching it in the symbolic field collection in order to get the correct symbolic

Instruction 1	Input: 1	Output: 3
Instruction 2	Input: 2	Output: 5
Instruction 3	Input: 5	Output: 6
Instruction 4	Input: 6	Output: 4

Table 4.5: Tabular representation of instructions after the second update, output fields are now unique

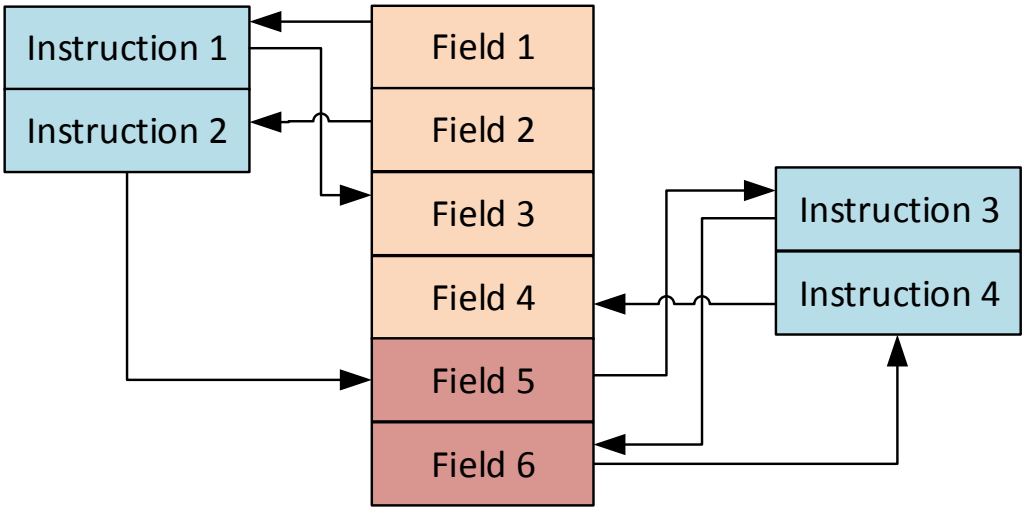

Figure 4.3: Graphic representation with unique fields

field element.

- Checking requirements of the input fields and parameters with regard to the given instruction.
- Updating the symbolic output field according to the symbolic instruction, meaning assigning it the correct data type with, if necessary, additional information and adding it to the key collection. Additionally, some other manipulations can be made, like adding a key as a lower key to an input field, if that input is encrypting another key.
- Setting the function pointers for the confidentiality level considering direct leakage as well as threshold leakage and determining the initial confidentiality values as specified in section 3.8.3. More information about the implementation of the confidentiality functions can be found in section 4.10.5.
- Adding the output field to the guarded fields if it has a guard

Some symbolic instructions are explained in section 4.10.7. These instructions are used in the example in section 5.1. Not all instructions are discussed here, because that would go beyond the scope of this thesis.

4.8 Confidentiality Check

During symbolic execution of the instructions, every symbolic output field gets function pointers to pre-defined *confidentiality functions* assigned. These confidentiality functions characterise a possible leakage of parts or all of the field and add a lower bound to the confidentiality of the field.

At the end of the symbolic execution every output field reference in the symbolic output structure will be called with a *publish* function. This publish function calls the leak function-pointer of the published field with the maximum confidentiality level first, then the lower bound confidentiality function-pointer with 0. This means everything of this field will be leaked and the lower bound of the confidentiality is set to zero. In case the confidentiality level considering threshold or direct leakage falls below threshold of the given secret a security violation will be triggered. See section 5.1 and especially section 5.1.4 for an extensive example of this concept.

Keep in mind that there are two thresholds for the confidentiality level: one for long term secrets and a general threshold. The threshold for long term secrets must always be higher than the general threshold. Basically a long term secret never should change confidentiality, however, if the confidentiality considering threshold leakage of a long term secret changes and is still above the pre defined threshold for long term secrets, a warning will be displayed.

4.8.1 Execution of Guards

After the actual confidentiality check, executing all set confidentiality functions, the guards are called one after another. These guards are assigned to the output fields of symbolic instructions in the same way as confidentiality functions, but, in contrast to the confidentiality functions, just to specific output fields and not to all. The *guarded fields* are additionally added to a structure to save them for the following execution of guards after the confidentiality check.

With a special return parameter system, which is explained in section 4.10.5, it can be recognised if an error occurs or a confidentiality level of a field has changed. If a confidentiality level has changed, all guards must be called again to ensure that this change does not affect the guards of other fields triggering an additional confidentiality level change. All guards are executed as often as confidentiality levels, while executing, are changing. At some point either no confidentiality level changes any more or the confidentiality of one field falls below its threshold and the security guard exits with an error. Either way it is secured that the security guard terminates and cannot come to an endless loop.

4.9 Result

After the publishing functions, including the executions of the guards, were done and no violation has been triggered, the secure program is good to go and can be loaded. But before that the needed memory for all the data and structures has to be freed in a stable way. This is the last step of the security guard execution. For a discussion of the general stability of the design see section 3.12 in the design chapter.

4.10 Used Concepts and Data Structures

This chapter illustrates the implementation details of some concepts defined in the preceding sections.

4.10.1 Structures used for Symbolic Fields

Symbolic fields are holding the key data of the security guard. The structures used to manage and safe all symbolic fields are the most important ones and are holding most of the information needed by the security guard.

A symbolic field element consists of the following elements:

- The **length** of the field.
- The **index** of the field in the actual secure program, which can be different from the symbolic field because of field uniqueness (see section 3.7). This index has no actual use during the execution of the security guard but is handy for bug tracking and troubleshooting in case of an error or security violation.
- The **previous symbolic field index** of the field in case the original field gets overwritten by a subsequent instruction.
- The **data type** represented as explained in section 4.10.3.
- **Additional data** dependent from the data type.
- **Confidentiality function pointers**, considering threshold and direct leakage. Meaning the publishing functions *leak()* and *add_conf_lb()*, and, if needed, the *guard*.
- **Confidentiality related data** like the actual values of the confidentiality level considering threshold and direct leakage, and some additional data dependent from the instruction. For more information see section 4.10.5.
- If the field element's content is the output of an instruction with confidentiality functions, which itself is dependent on input fields the field element can store those **input fields**. For more information regarding confidentiality functions see section 4.10.5.

- The **key ID**, if the field is a key. See section 4.10.4 for more information.
- A list of **keys** this field depends on to detect cycle encryption.

Symbolic fields are stored in a linear linked list, where the field id, used for identifying the field in references, is the index number in the linked list itself.

This field list, holding all symbolic fields needed during execution of the security guard, is stored in a symbolic field collection structure which additionally stores the size of the field list and another list of the guarded fields. The guarded fields list is needed during execution of the guards as explained in section 4.8.1.

4.10.2 Structures used for Symbolic Instructions

Symbolic instructions are stored in the symbolic instruction element structure which is held by the symbolic instruction collection. The symbolic instruction collection consists of a normal array, in which the instruction elements are stored, together with an integer holding the number of instruction elements. There is no need for a linked list to save the instruction elements because the instructions are parsed at the beginning (see section 4.5.3) and never change in count, therefore no dynamic saving structure is needed.

Symbolic instruction elements consist of:

- The **instruction code** defined by the implementation of the secure program. This instruction code is used to identify the function that executes this instruction element during symbolic execution.
- An array of references to the **input fields** and an integer saving the quantity of references.
- An array of references to the **output fields** and an integer saving the quantity of references.
- An array of **parameters** used during symbolic execution and an integer saving the quantity of parameters. Parameters can be the type of curve, when generating an ECC key pair, or the type of algorithm, AES or Triple DES, when encrypting something with a symmetric cipher.

References to symbolic field elements consist of three values:

- The **index** of the symbolic field element in the linked list stored in the symbolic field collection.
- The **length** of the reference in the field. This length can be less than the actual size of the field, in which case a sub-field has been referenced.

Nibble	1	2	3	4	5	6	7	8
Type	0	0	SData	0	Subtype	Subtype	Subtype	Subtype

Table 4.6: Data type system

Nibble	1	2	3	4	5	6	7	8
Type	0	0	SData (1)	0	0	0	UGSData	LTSData

Table 4.7: Secret Data-subtypes

- The **offset** in the referenced field. If this offset is a different value than 0, a sub-field is referenced. When dereferencing such a field, a check is made whether the offset and the length of the reference exceeds the length of the referenced symbolic field element.

For more information about sub-fields see section 4.10.6.

4.10.3 Data Types

To support the data types and their sub data types defined in section 3.9 a bit-based system is introduced. Every symbolic field element, as discussed in section 4.10.1, owns an integer member defining the data type, which has the following structure: The first four nibbles are to determine whether the type is Data or SData. If it is SData the 3$^{\text{rd}}$ nibble is set to one, if it is Data the 4$^{\text{th}}$ nibble is set to one. The four lower nibbles are for subtypes. In table 4.6 the basic structure can be seen.

SData has the subtypes Long Term Secret Data (LTSData) and User Generated Secret Data (UGSData). When having an SData Secret, the 3rd nibble must be set to one and the 7$^{\text{th}}$ and 8$^{\text{th}}$ nibble determines whether it is LTSData or UGSData, as seen in table 4.7. As an example, if the data type is LTSData, the 3$^{\text{rd}}$ and the 7$^{\text{th}}$ nibble is set to one. The representation in hexadecimal code is 0010 0010.

The Data datatype has a more complex subtype system: the 6$^{\text{th}}$ nibble specifies whether it is a symmetric key, asymmetric key, or other data, like hash, cipher text, or random data. If it is an asymmetric key the 7$^{\text{th}}$ nibble specifies the security type of the key, whether it is a public, private, signing, or verification key, and the 8$^{\text{th}}$ nibble specifies the elementary key algorithm type, ECC, RSA, or part of an RSA key. If it is a symmetric key, the 6$^{\text{th}}$ nibble is set to two and the the 8$^{\text{th}}$ nibble specifies the specific key type, whether it is an encryption key, a mac key, or both. In table 4.8 you can see the structure of a symmetric and an asymmetric key data type, the values in brackets are fixed values for this type.

In table 4.9 are the data types, which were defined in section 3.9 and are therefore supported by the security guard. All numbers are the representation of one nibble of the integer in hexadecimal.

Nibble	1	2	3	4	5	6	7	8
Type	0	0	0	Data (1)	0	Asymmetric Key (1)	Security Type	Key Algorithm
Type	0	0	0	Data (1)	0	Symmetric Key (2)	0	Key Type

Table 4.8: Symmetric and asymmetric key subtypes

Nibble / Type	1	2	3	4	5	6	7	8
SData	0	0	1	0	0	0	0	0
LTSData	0	0	1	0	0	0	0	1
UGSData	0	0	1	0	0	0	1	0
Data	0	0	0	1	0	0	0	0
Public Key	0	0	0	1	0	1	1	0
Private Key	0	0	0	1	0	1	2	0
RSA Public Key	0	0	0	1	0	1	1	1
RSA Private Key	0	0	0	1	0	1	2	1
ECC Public Key	0	0	0	1	0	1	1	3
ECC Private Key	0	0	0	1	0	1	2	3
Symmetric Key	0	0	0	1	0	2	0	0
Hash	0	0	0	1	0	4	1	1
Signature	0	0	0	1	0	4	1	2
Symmetric Cipher Text	0	0	0	1	0	4	2	1
Asymmetric Cipher Text	0	0	0	1	0	4	2	2
Random	0	0	0	1	0	4	4	1

Table 4.9: Representation of all supported data types

```
1 // check if data type is Data
2 (X & 0x0001 0000)
3 // check if data type is RSA Key
4 ((X & 0x0001 0101) == 0x0001 0101)
5 // check if data type is Public Key
6 ((X & 0x0001 0110) == 0x0001 0110)
7 // check if data type is RSA Public Key
8 X == 0x0001 0111
```

Listing 4.1: Different data type checks

Data Type Checking

The benefit of the structure explained in this section is to check for specific properties of a data type of a symbolic field element with simple bit-wise operations.

An exemplary summarisation of data type checks can be seen in listing 4.1. Line 2 shows a check whether the data type is Data with a bit-wise AND-operation, it will always determine the correct supertype without considering any sub data type.
Line 4 shows a check whether a field is an RSA Key, while ignoring the key type. It is checked if the 4^{th} (Data/SData), 6^{th} (symmetric/asymmetric key), and 8^{th} (key algorithm) nibble is set to one. In contrast to the Data-data type check this bit-wise AND-operation hast to be compared with the same value to ensure that all three bits are set, because the bit-wise AND-operation would also return a TRUE value if just one of these three nibbles, and therefore bits, are set to one (TRUE). The Data-data type check just checks for one bit and therefore does not need to be compared with the value itself.

Line 6 on the other hand checks for a public key without considering the algoritm type of the public key, be it RSA or ECC. And finally, for completeness, line 8 shows a check for one specific data type. In this example it is for an RSA public key.

4.10.4 Key Management and Identification

Secret key fields, of long term secrets or generated secrets, have a unique Key Identification (key ID) parameter to determine, for instance, which key is a copy of another key, the public key of the private key, or the key part of an RSA key. In the implementation the key ID is a pointer to a symbolic_secret_key struct which has the following members:

- A list to other key IDs which are dependent from this key.
- The secret level of this key.
- The actual symbolic field element of the key. It is a reference to the field of the first (virtual) field element the key has occurred.

4.10.5 Confidentiality Functions

Confidentiality functions are implemented as independent C-functions with fixed input and output parameter types. Four of these functions are held by every symbolic field of the secure program: One for getting the confidentiality level considering direct leakage (named ConfDL(F), see section 3.8.1) and one for getting the confidentiality level considering threshold leakage (named ConfTH(F), see section 3.8.2), both of them were discussed in section 3.8.4. The other two functions are used to update the confidentiality levels during publishing the fields at the end of the security guard execution with the functions Leak(F, L) and AddConfLB(F, B), as discussed in section 3.8.5. Another special set confidentiality function are the guard functions. One of these functions, already explained in section 3.8.7, is only assigned to output fields of specific symbolic instructions during symbolic execution. Every other field does not need to have a guard function.

The confidentiality functions are getting assigned to every field via function pointers. When called, the functions get the field (F) they are assigned to as a function parameter to be able to access field related values. Additionally, the Leak-function gets the value the field gets leaked (L) and the AddConfLB function gets its new lower bound (L). Most of the field values, as well as the function pointers themselves, are getting assigned to the field during the symbolic execution or when initialising the fields. When initialising the fields during parsing the secure program, constant fields are holding non security relevant data and therefore can be initialised with data, same for data fields, which, at this point, are holding no information at all. Additionally, data and constant fields contain the following confidentiality functions:

- **ConfDL(F):** RETURN 0.
- **ConfTH(F):** RETURN 0.
- **Leak(F, L):** No operation.
- **AddConfLB(F, B):** No operation.
- **Guard(F):** None.

Secret fields on the other hand contain secret material from the beginning. To be more specific: they always contain secret keys. While the exact content of fields is not known, the symbolic data has been determined before the execution of the security guard (see section 4.3) and then processed (see section 4.4) and therefore is known. With these values the correct data type as well as additional data and the secret level is known, a key ID can be generated and the confidentiality levels could be determined. The confidentiality level considering direct leakage and the confidentiality level considering threshold leakage are the same if set during generation of a secret field, but saved in different variables for further manipulation. The confidentiality functions of a secret field is set as follows:

- **ConfDL(F):** return the determined confidentiality variable considering direct leakage.
- **ConfTH(F):** return the determined confidentiality variable considering threshold leakage.
- **Leak(F, L):** If L is greater than 0, abort the Symbolic Execution indicate a failed verification of the secure program. External secrets can be reused multiple times by other executions of the secure program and therefore it is not known how often a leak happens.
- **AddConfLB(F, B):** If B is less than ConfTH(F), abort the Symbolic Execution and indicate a failed verification of the secure program. Same explanation as in Leak(F, L) applies.
- **Guard(F):** Assign none.

A discussion of confidentiality functions assigned during the symbolic execution of instructions can be found in section 4.10.7.

Return Value System

Confidentiality functions can have three different return values, apart from a return value indicating nothing has happened: One return value that informs if a security violation has happened, another one to inform if an error has happened, in both cases the symbolic execution is aborted and the secure program is rejected, and a third one for an alteration of a confidentiality value of a field. A detection of an alteration is needed if the confidentiality function is called by a guard, because, as explained in section 3.8.7, an alteration of confidentiality values during guard execution means to execute all guards again until no confidentiality values change any more.

Some confidentiality functions can call more than one other confidentiality function, therefore return values must be handled in a special way. One not so elegant way would be to store all return values in some type of array or list. Another more elegant way is to store the return value in a simple integer, not with an assignment, but rather with a bit wise operation: The three return values, security violation, error, and alteration set different bits in the return value. If a confidentiality function calls more than one other confidentiality function the return values get combined with a simple bit-wise OR-operation. With this concept no return value can be lost or overwritten.

4.10.6 Sub Field Support

An instruction has references to its input and output fields. These references can, as already discussed in section 3.11, reference to just a part of a field. This is accomplished with a size and an offset parameter. To accomplish the goal of no security violations, the following actions will be taken.

Full Secret Field Reference

As already specified in section 3.11.1, fields are not allowed to be referenced just in parts but must be referenced as the whole field. To accomplish this goal every field reference is checked during differentiation whether the field is a secret and if the reference to the field is a sub-field reference. If this is the case a security violation will be declared and the program is terminated with a security violation error.

Sub Field Output

This concept is rather complicated and therefore firstly explained, before a simple example, for better understanding, is given in section 4.10.6. The reason why this action is performed this way is to avoid to update the following symbolic instructions again with the newly generated concatenated output field.

After every symbolic instruction, its output fields will be checked whether a sub-field of the actual field has been referenced. If this is the case, the former symbolic data field, which was generated during updating the instructions (see section 4.6), of the actual secure program data field will be detected. This is necessary because it is the actual field which partially will be overwritten and hence there is still information of it existent. Then a new virtual data field will be created and the symbolic output field of the instruction will be copied (virtually with the instruction copy) to the new field. The actual output field, not the new one, will be cleared and reset. Next, a virtual concatenate instruction will be called with the new data field and the former data field as inputs and the actual current instruction output field as output. This ensures that every data and secret, which was in the actual field, will not be ignored and the following symbolic instructions and the confidentiality check publishing the output fields still operate correctly while no updating of any other fields or instructions is necessary.

Example of Sub Field Output Handling

Let there be two instructions. The first instruction has the output field 1, the second instruction writes in a sub-field of the same output field 1.
Figure 4.4 represents the memory part from the field in which the real execution of the secure program writes the output. At first the output of instruction 1 writes to the whole field DATA 1, marked as red. Then the output of instruction 2 partially overwrites the field DATA 1, marked in blue.
As explained in section 4.6, instruction 2 already has a new, unique, symbolic output field and the following instruction is updated with a new output field, to not overwrite the output field of instruction one. Let the field be called VDATA 2. Instruction 1 now writes to VDATA 1 and instruction 2 writes to the independent field VDATA 2.
Figure 4.5 represents the symbolic field collection after the execution of the second symbolic instruction.

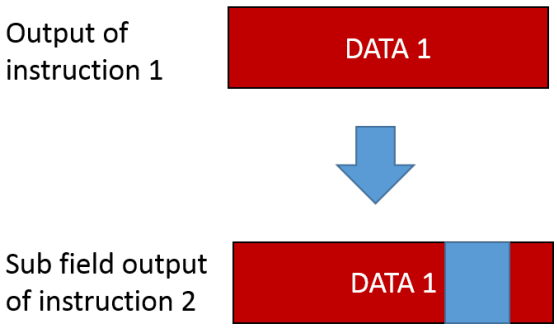

Figure 4.4: Actual field before and after execution of an instruction with sub-field output

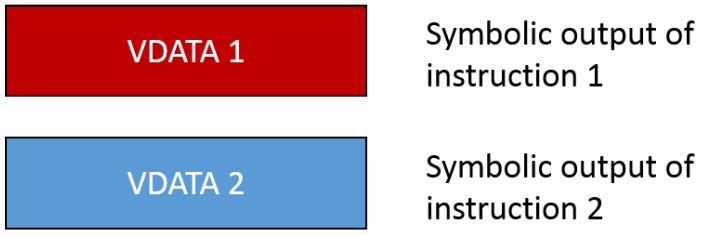

Figure 4.5: Symbolic fields after execution

During the symbolic execution and after the execution of instruction 2 a check will be made if the actual output field of the secure program was a sub-field output. In our example this is the case and two major actions will be made. Firstly, a whole new field will be added to the end of the symbolic field collection, let the new field be called VDATA 3. Then, the content of the symbolic output of instruction 2, stored in VDATA 2, will be copied to the newly generated field VDATA 3. In figure 4.6 the copy operation and the new field can be seen.
Finally, the former symbolic output field of the actual field is determined. In the example it is VDATA 1. This field will be virtually concatenated with the copied field VDATA 3 and the output field of this virtual instruction concatenate is the symbolic field VDATA 2.
Following instructions, which have the symbolic field VDATA 2 as an input field, have now the updated, concatenated, field as an input field. No further actions have to be taken.

4.10.7 Symbolic Instructions

This chapter discusses the used functions of the example in section 5.1. The actual implementation of the security guard consists of more symbolic functions, but discussing all of them would go beyond the scope of this thesis.

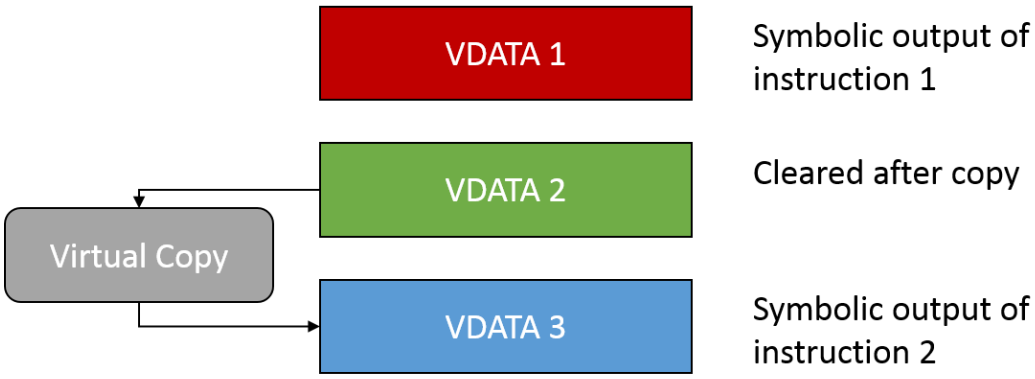

Figure 4.6: Copying symbolic field 2 two the newly generated field 3

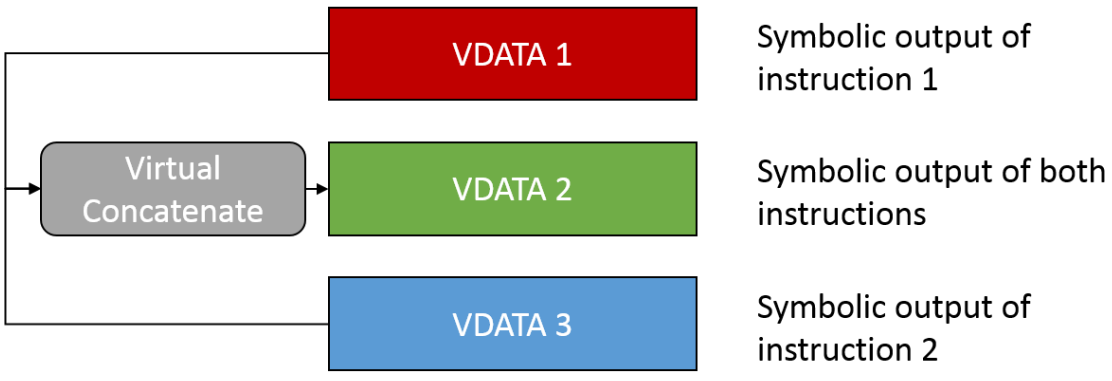

Figure 4.7: Concatenation of symbolic fields 1 and 3 into field 2

Encrypt Symmetric

This instruction encrypts a given message using a symmetric algorithm (Triple DES Encryption Algorithm (TDEA) or Advanced Encryption Standard (AES)) using the specified encryption mode and key.

- **Output:** The reference to the data field to which the result shall be written (can be NULL if the input field shall be used).
- **Key:** The reference to the field containing the key to be used for encryption. The length of the key needs to fit according to the used algorithm (TDEA: 16, 24; AES: 16, 24, 32).
- **Input:** The reference to the field which shall be encrypted. Can be used as an output field if the optional target parameter is omitted and if this field is a data field.
- **Algorithm:** Integer-parameter for the used algorithm: Triple DES Encryption Algorithm (TDEA) or Advanced Encryption Standard (AES).
- **Mode:** Integer-parameter for the used block cipher mode of operation: Electronic Codebook (ECB) or Cipher Block Chaining (CBC).
- **Padding:** Integer-parameter for the used cipher padding.
- **IV:** The reference to the field which is used as initialisation vector (can be NULL if it shall be omitted).
- **Requirements:**
  - Input not in highest secret level or Type(Input) == PublicKey
  - Type(Key) == SymmetricEncryptionKey(Algorithm)
  - IF Mode == ECB:
    - Keys(Input) == {}
  - FOR K in Keys(Input):
    - Key != K
    - Key not in LowerKeys(K)
  - **Comment:** Key input must be of type symmetric encryption key and the algorithm input must match the algorithm of the key. ECB mode is discouraged and, if it is used, the input must not depend on keys.
- **Symbolic Execution:**
  - Type(Output) := Symmetric cipher text with key, mode and input as additional data

  - Keys(Output) := Keys(Input) + Key
  - LowerKeys(Key) += Keys(Input)
  - **Comment:** Key now also encrypts/protects all input keys through this instruction, hence they are added to lower keys.

- **ConfDL(Output):**
  - RETURN 0
  - **Comment:** Without true randomness during encryption, the output of this operation must be considered as known to the adversary. E.g., the adversary could have seen a ciphertext of this input under this key before (different secure program). Without additional tracking of fresh entropy in input and key, 0 is the natural bound for the confidentiality level of the ciphertext.

- **ConfTH(Output):**
  - RETURN 0
  - **Comment:** See comment of ConfDL(Output)

- **Leak(Output, L):**
  - No operation
  - **Comment:** ConfDL(Output) is already zero.

- **AddConfLb(Output, B):**
  - No operation
  - **Comment:** ConfTH(Output) is already zero.

- **Guards:**
  - **On Conf(Key):**
  - IF Conf(Key) < CONF_THRESHOLD:
    - AddConfLB(Input, Conf(Output)) [Conf(Output) == 0!]
  - ELSE
    - AddConfLB(Input, Conf(Key))
  - **Comment:** Encryption is threshold-based regarding confidentiality: As long as the key is strong enough, the input is protected at the confidentiality level of the key. If the key is weak, the confidentiality level of the input is the same as that of the output.

Compute CRC

Computes a Cyclic Redundancy Check (CRC) of the input values.

- **Output:** The reference to the data field to which the result shall be written to.
- **Key:** The reference to the field which contains the key over which the checksum shall be calculated.
- **Size:** Integer-parameter for the size of the checksum in bit.
- **Requirements:** None.
- **Symbolic Execution:**
  - Type(Output) := Data
  - Keys(Output) := Keys(Input)
  - Leakage_Potential := Size
  - **Comment:** The local variable Leakage_Potential keeps track of the leakage that can happen through this CRC. It equals the size of the CRC (meaning the maximum number of bits of information that can leak). Leakage_Potential eventually drops (with a lower bound of zero) as information about the CRC is disclosed. No special output data type needed since the symbolic execution of the security guard does not require at any point during execution that the input should be a MAC data type.
- **ConfDL(Output):**
  - RETURN 0
  - **Comment:** The CRC instruction itself does not include fresh randomness. Consequentially, the output may already be known by the adversary. (Note that the worst case here is that the adversary already knows the CRC, while for leakage the worst-case assumption is that the adversary did not know the CRC value yet.)
- **ConfTH(Output):**
  - RETURN 0
  - **Comment:** See comment of ConfDL(Output).
- **Leak(Output, L):**
  - Leak(Input, min(Leakage_Potential, L))
  - Leakage_Potential := Leakage_Potential - L
  - **Comment:** Once there is some leakage on the Output, it is propagated to the Input.

- **AddConfLb(Output, B):**
  - IF B < CONF_THRESHOLD:
    - Leak(Input, Leakage_Potential)
    - Leakage_Potential := 0
  - **Comment:** If the threshold confidentiality level of Output drops below CONF_THRESHOLD, the Output is considered to be known by the adversary. Consequentially, the remaining Leakage_Potential is leaked on Input.
- **Guards:** None

Concatenate

This instruction concatenates a list of given fields and writes the result into a data field.

- **Output:** Reference to the destination field. The destination must be a data field.
- **Input_n:** Up to N fields that are to be concatenated.
- **Requirements:**
  - Input_n not in highest secret level for n in [1...N].
- **Symbolic Execution:**
  - IF there is an Input_n FOR n in [1...N] such that Input_n has a secret level of a Long Term Secret (LTS) or Type(Input_n) == LTSData:
    - Type(Output) := LTSData
  - ELSE IF there is an Input_n FOR n in [1...N] such that Input_n has a secret level or Type(Input_n) == UGSData:
    - Type(Output) := UGSData
  - ELSE IF all Input_n FOR n in [1...N] Type(Input_n) == Random:
    - Type(Output) := Random
  - ELSE:
    - Type(Output) := Data
  - Keys(Output) := Union of Keys(Input_n) FOR n in [1...N].
- **ConfDL(Output):** RETURN MAX(ConfDL(Input_n) FOR n in [2...N])
  - **Comment:** The confidentiality level (CL) of the output field is at least as high as the CL of the most confidential input. Concatenation of two independent 128 bit long random keys K1, K2 with some published information to search for K1, K2 independently (e.g., Hash(K1) and Hash(K2)) would lead to a 129 bit confidentiality level of the output. Hence, for a better approximation, the independence of values would need to be checked as well as their use.

- **ConfTH(Output):** RETURN MAX(ConfTH(Input_n) FOR n in [2...N])
  - **Comment:** See comment of ConfDL(Output)
- **Leak(Output, L):**
  - FOR n in [1...N]:
    - Leak(Input_n, L)
  - **Comment:** Without knowing which part of the output information was leaked, the lower-bound approximation is considered only as leakage on every input.
- **AddConfLb(Output, B):**
  - FOR n in [1...N]:
    - AddConfLb(Input_n, B)
  - **Comment:** As a reminder: ConfTH defines the expected effort for the adversary to learn the Output, i.e., the concatenation of all inputs here. Therefore, this threshold is also immediately a threshold for all inputs.
- **Guards:** None

SP800-108 Key Derivation

This instruction derives a key from a given master key using the algorithm defined in NIST SP 800 - 108 [21] in counter mode.

- **Output:** The reference to the data field to which the resulting derived key is written.
- **Key:** The reference to the field which contains the master key.
- **Prefix** The reference to the field which contains the prefix of this derivation.
- **Postfix** The reference to the field which contains the postfix of this derivation.
- **Algorithm:** Integer-Parameter indicating which encryption algorithm should be applied: Triple DES Encryption Algorithm (TDEA) or Advanced Encryption Standard (AES)
- **Requirements:**
  - Type(Key) is a symmetric encryption key with the same algorithm
- **Symbolic Execution:**
  - IF there is an earlier call of SP800-108 key derivation with the same key, prefix, postfix, and algorithm in this secure program:

    - Symbolically execute copy(Output', Output) where Output' is the output of the last SP800-108 key derivation call with the same key, prefix, postfix, and algorithm in this secure program.
  - ELSE:
    - Create new key ID, referred to as GenKey, for the generated key.
    - Add GenKey to to the secret level according to User Generated Secrets (UGS).
    - Type(Output) := Symmetric Key
    - Keys(Output) := GenKey, Key
    - LowerKeys(Key) += GenKey
    - Output_Leakage := 0
    - Output_ConfTH := ConfTH(Key)
  - **Comment:** Since the SP800-108 key derivation is a deterministic function, it produces the same output key for the same key, prefix, and postfix combination. This has two consequences:
    (1) If the output key was already computed in this secure program, a second call to a SP800-108 key derivation must not create a new key but instead perform a copy of the already existing one to the output field. Hence the first branch.
    (2) If the key is a Long Term Secret (LTS), meaning it comes from outside the high security device, and prefix and postfix are not a random-data type, there is no guaranteed fresh randomness in the output. Meaning the output key was potentially already computed with leakage in previous run of the secure program. Hence, the confidentiality level of the output is 0. (An alternative would be to treat the Output as a long-term secret, in this case, which would prevent all leakage.)
    Output_ConfTH is a local variable to track the confidentiality level (CL) of the Output which additionally depends on the CL of the used Key. As the type of the output is of SymmetricKey and not SymmetricEncryptionKey, the key cannot be used in a key position within the secure program. However, it is treated as a generated secret and must have the required confidentiality level.

- **ConfDL(Output):**
  - RETURN MAX(0, ConfDL(Key) - Output_Leakage)
  - **Comment:** Output_Leakage is the local variable measuring how many bits of the output have been leaked already.

- **ConfTH(Output):**
  - RETURN MIN(Output_ConfTH, ConfTH(Key))
  - **Comment:** See comment of ConfDL(Output).

- **Leak(Output, L):**
  - Output_Leakage += L
  - **Comment:** See comment of ConfDL(Output).
- **AddConfLb(Output, B):**
  - Output_ConfTH := min(Output_ConfTH, B)
  - **Comment:** See comment of ConfDL(Output).
- **Guards:** None.

Chapter 5

Results and Analysis

This chapter addresses the finished implementation of the security guard. It introduces a simple, but representable, example of a secure program and discusses its analysis by the security guard in section 5.1. Then, a brief performance analysis will be performed in section 5.2. Finally security considerations, which came up during design and implementation of the security guard, will be discussed in section 5.3.

5.1 Example

This section discusses one simple run of the security guard to illustrate the function of the implementation. At first we define the needed fields and the instructions followed by an accurate explanation of the process running through the security guard. Then the example will be a little bit altered to show different results in the concept of the security guard.

5.1.1 Definition of the Secure Program and its Fields

This sections defines, apart from the instructions needed for the example, the used fields to accomplish a correct result.

The exemplary secure program and field configuration is defined as follows:
Two constant fields, coming from outside the high security device defining the constant prefix and the unique identifier (UID) of an exemplary product, together with a Derivation Master Key in a secret field are used in an SP800-108 key derivation function. This is a basic example of a key derivation, with the UID acting as an unique input.

Before encrypting the freshly generated derived key, a 32-bit CRC key checksum is calculated to ensure integrity of the key.
Then the calculated CRC is concatenated with the key into one field. Afterwards, this field is encrypted with the Master Encryption Key, again defined in a secret field coming from outside the high security device. Finally, the result is written to the output, where

it is returned by the high security device.

All in all the following constant and secret fields, together with the needed data fields, are defined to achieve the aforementioned process:

- Constant Fields:
  - SP800 Constant Prefix
  - UID
- Secret Fields:
  - Derivation Master Key
    - **Key type:** AES-256
  - Encryption Master Key
    - **Key type:** AES-256
- Data Fields:
  - Derived Key
  - Checksum
  - Temp
  - Output

The following instructions are defined in the exemplary secure program to generate the derived key, calculate the checksum, concatenate the two results, and encrypt them with the Master Encryption Key. The exact definitions of the instructions itself can be found in section 4.10.7.

1. SP800-108
   - Key: Derivation Master Key
   - Prefix: “SP800 Constant Prefix” constant field
   - Postfix: “UID” constant field
   - Output: Derived Key
   - Algorithm: AES
2. calculate CRC
   - Input: Derived Key
   - Output: Checksum
   - Size: 32

3. Concatenate:
    - Input: Derived Key
    - Input: Checksum
    - Output: Temp
4. Encrypt Symmetric:
    - Input: Derived Key
    - Key: Master Encryption Key
    - Output: Output
    - Algorithm: AES
    - Mode: CBC
    - Padding: ISO 7816-4 [22]
    - IV: Null

Additionally, the security guard needs some additional parameters too: during the symbolic execution generated secrets get the defined secret level for generated secrets, additionally, the threshold, the confidentiality level of a generated secret can drop before the secure program gets rejected, is defined as 240 (CONF_THRESHOLD := 240). To provide simplicity, no sub-fields are referenced in the instructions and every field is written to once, i.e. field uniqueness is given and no fields have been added during preparation.

5.1.2 Preparation

With the definition of the instructions and the needed fields the execution of the security guard can start.

The first operation inside the security guard is processing the imported secret data (described in section 4.4). Here the two symbolic secret fields, Derivation Master Key and Master Encryption Key, are generated together with a respective key ID holding the defined secret level for imported secrets. Additionally, the initial confidentiality level is determined: Both keys are AES 256 keys and as of the design specified in section 3.8.3 the initial confidentiality level of AES 256 keys is 256. The data type is set to Symmetric Key with the algorithm type “AES” as an additional parameter.

After the external Long Time Secrets are initialised as symbolic fields the other fields together with the instructions are parsed from the secure program as described in section 4.5. The symbolic structures are initialised and the symbolic instructions are set with the correct references to the fields as input and output values.

All fields are unique, no additional fields have to be added, and no instructions have

to be updated as described in section 4.6. For an example regarding instruction updating please see section 4.6.1.

5.1.3 Symbolic Execution

After all preperations are finished the symbolic execution starts as described in section 4.7. This means all instructions are called in the order described in the secure program.

SP800 - 108 Key Derivation

At first the requirements, as defined in section 4.10.7, are checked for the SP800-108 key derivation. The only requirement is that the referenced input key algorithm matches the algorithm parameter of the instruction, which in this case is correct.

Then the actual symbolic execution of this particular instruction starts: The referenced fields used as input parameters (prefix, postfix, master key) are checked whether at least one of these fields was never used together with the other fields in an SP800-108 derivation during this execution of the secure program. Hence this is the first instruction this is not the case. The input references are listed as used references in this combination for an SP800-108 derivation, so future executions can recognise them, and a new key will be generated in the output field and also referenced for future executions. The new key gets a new key ID with the secret level of generated secrets and this key ID, together with the key ID of the derivation Master Key, is added to the keys this output field depends on, as described in section 4.10.1. Furthermore, the newly generated key ID is added to the lower keys of the Derivation Master Key. The data type Symmetric Key, without any information of the algorithm and some other internal parameters used by the confidentiality functions (Output_Leakage (0) and Output_ConfTH (256)), is set in the output field.

Lastly, the confidentiality functions are set accordingly. The discussion of these functions will be done in section 5.1.4.

Computing the CRC-Checksum

This instruction calculates a CRC-checksum of the newly derived key to provide integrity of the key. As defined in section 4.10.7 there are no requirements for this instruction so without further ado the symbolic execution can start. The data type of the output field gets set to data, the leakage_potential gets set to 32 as well as the keys, this field depends from, are getting set to the same keys the input field depends from, which are the key IDs of the Derivation Master Key and the derived key itself.

Concatenating the Derived Key and the Checksum

The newly derived key and its computed checksum are concatenated together into a new field to be able to encrypt it in the last step.

There are no Long Term Secrets in the inputs but one Generated Secret and a data field, as a result the output field gets the data type UGSData, indicating that this field contains at least one generated secret but no Long Term Secret. Additionally, the output field depends again on the Derivation Master Key and the derived key because both input fields depend on both keys. These key IDs get added to the keys the output field depends on.

Again the confidentiality functions are set, the discussion of the execution of the confidentiality can be found in section 5.1.4.

Encrypting the Result

Firstly, the requirements are checked: The input, which is the derived key, is a symmetric key with the secret level for generated secrets. The algorithm specified for the instruction, AES, matches with the algorithm of the key and the key hierarchy is also correct.

The output field, which in the end gets written to the output of the secure program and returned by the high security device, gets the data type Symmetric Cipher Text with the following additional data: The key is a reference to the field of the Master Encryption Key, the Mode is AES, and the Input is a reference to the field of the concatenation result. The keys the output field depends on are set to the keys the input and the concatenation result depends on, which are the key IDs of the Derivation Master Key and the derived key itself. The output field depends now from the Encryption Master Key, as a result its key ID gets added to the Keys the output field depends on. Lastly, the derived key and the Derivation Master Key get added to the Lower Keys of the Encryption Master Key.

Confidentiality functions are rather simple in this instruction except the circumstance that this instruction has a guard. As a result the guard function of this instruction gets set in the output field and the output field itself gets saved in the Guarded Field-list.

5.1.4 Confidentiality Check

After the last symbolic instruction has finished all fields in the output get published with the function explained in section 3.8.6. In this example the Publish-function calls the Leak() and AddConfLB() functions of the cipher text. Both of them have no operation so nothing is done and the Publish-function ends.

Then all guards, collected during symbolic execution, are called. For a better understanding the guard execution is illustrated in figure 5.1. The only collected guard in this example is the one from the Output field, which is the final output from the encrypt symmetric instruction and checks whether the confidentiality of the Master Encryption Key is still above the confidentiality threshold. This is the case so the ELSE branch defined

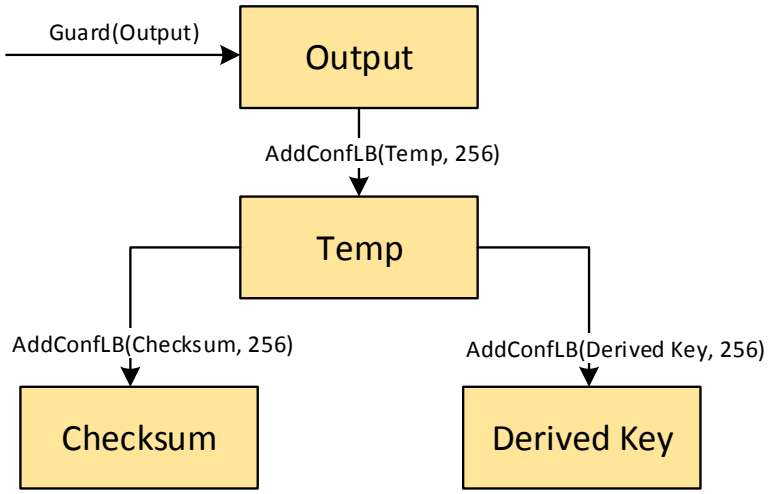

Figure 5.1: Representation of the calling order during guard execution

in the guard in section 4.10.7 is executed and triggers an AddConfLB(temp, 256), which is the input of the encryption function and the concatenation of the derived key together with its CRC-checksum. The AddConfLB(temp, 256) triggers an execution of AddConfLB(Derived Key, 256) and AddConfLB(Checksum, 256) which are the input fields of the concatenation instruction. The order of the execution does not matter, the execution of AddConfLB(temp, 256) waits for the return value of both functions and combines these values with a simple logical Or-function as explained in section 4.10.5.

Firstly, the AddConfLB(Derived Key, 256) execution is discussed: the local variable Output_ConfTH is set to the minimum of itself or 256. Because the value of Output_ConfTH is 256 as well, it stays the same and a return value indicating no error is returned. Then AddConfLB(Checksum, 256) is executed: B, which is 256, is above the CONF_THRESHOLD of 240 and again a return value indicating no error is returned.

Both return values are combined as stated in section 4.10.5 and propagated back to the execution of the guard. Because no security violation or error has happened and no alteration was done the execution of the guards is finished, resulting in the end of the execution of the security guard. The secure program is accepted and can be executed with real values on an actual high security device.

5.1.5 Variation of the Example

The confidentiality check of the first version of the example is rather simple because just an encrypted cipher text is written to the output, this is obviously secure and triggers no

"suspicious" behaviour of the secure program, and therefore no violation is detected by the security guard.

Publishing the Checksum

One somewhat more interesting example would be writing the CRC-checksum from the derived key directly to the output, skipping the concatenation. This results in the encryption of just the key, but not its checksum, and writing both, the checksum and the derived key, to the output and publishing it.

The symbolic execution stays basically the same with the exception of not performing a concatenation of the derived key and its checksum, meaning the discussion in section 5.1.3 is still applicable.

The more critical part of this variation of the example is the confidentiality check. The publishing of the encrypted derived key is still similar to the discussion of the original example in section 5.1.4, but the additionally published checksum makes the difference. Publish(Checksum) entails Leak(Checksum, MAX_CONF_LEVEL) and AddConfLB(Checksum, 0), MAX_CONF_LEVEL is defined as 256. Leak(Checksum, 256) triggers, according to the definition in section 4.10.7, Leak(Derived Key, 32) and the local variable Checksum_Leakage_Potential is set to 0. Leak(Derived Key, 32) triggers $Output\_Leakage := 0 + 32 = 32$. This is already a security violation because the ConfDL function of the Derived Key-field now returns a value of $256 - 32 = 224$ which is below the CONF_THRESHOLD of 240. But this gets detected later, after the execution of the guards, when checking the confidentiality levels of all keys.

Lastly, the second function of publishing the Checksum-field is AddConfLB(Checksum, 0), this function triggers Leak(K, 0). This has no further effect since the local variable Leakage_Potential of the Checksum-field is already zero. The interested reader may notice that executing AddConfLB(Checksum, 0) before the Leak(Checksum, MAX_CONF_LEVEL) instructions would have led to the same result.

5.2 Performance Analysis

The primary focus of the security guard is, as its name suggests, to guard maliciously generated secure programs from an actual execution on a high security device. As defined in section 3.4, the security guard will be run just once when creating a new secure program and therefore performance is, as long as a termination happens in reasonable time, not from high importance. But, because of a potentially limited memory of a high security device, memory usage of the security guard has a far bigger importance. Especially with the requirement of field uniqueness, as defined in section 3.7, and the whole amount of symbolic data used during execution, the security guard is not a memory saving program.

SP	Instructions	Execution time Simulator [ms]	Execution time HSM [ms]	Memory usage [byte]
1	8	1	1	5,888
2	9	1	1	6,760
3	570	15	70	334,736
4	650	16	78	356,032

Table 5.1: Performance values of test runs with different secure programs on a simulator and a real high security device

As described in section 4.2, the security guard was implemented in C as a firmware for a hardware security module. Performance testing was performed on a simulator the hardware security module vendor provides for testing firmware code without needing an actual, expensive, hardware security module, as well as on a real hardware security module. The smallest unit the execution time could be measured was milliseconds.

Table 5.1 shows the performance values of executions of different secure programs. The execution times are mean values of multiple executions, the memory usage is identical for every execution of the same secure program and is also similar on simulator and real high security device. The execution time of a small secure program is not measurable in an exact way. Because the execution was faster than one millisecond (the displayed execution time was zero milliseconds) it has been rounded up to one millisecond.

The first observation is that the execution time does not rise as heavily with higher amount of instructions as the memory usage. With the implementation written in C there is hardly any execution overhead, the implementation can be seen as very time efficient and execution time will not be a problem with even bigger secure programs. Memory usage can be a problem with future, more comprehensive secure programs, but for foreseeable time memory in the high security device will be sufficient.

5.3 Security Considerations

This section describes some of the security considerations that came up while defining the requirements for instructions and the design of the security guard. The definitions are often a balancing act between simple requirements and use cases that need to be supported. The purpose of this section is to help the reader to understand the requirements of the security guard better. Reading and understanding this section is also recommended before making changes to the symbolic execution or the requirement for the instructions. Many of the considerations do not directly point to a specific attack that would be possible if they are ignored. The design principle here - as in general for the use of cryptography - is “safe use”, meaning sticking to constructions that are known to be secure and avoiding

those which are not, even if no concrete attack is publicly known (yet).

5.3.1 Iterative Attacks

In an iterative attack the adversary uses parts of the output again as inputs for the same or for a different secure program. This undermines the type system and the key hierarchy and immediately allows for several types of security violations, for instance encrypting a key in one run of a secure program and decrypting it in a next run results in revealing the formerly considered secure key. Hence, as an organizational measure, it is important to keep secure program inputs under control.

5.3.2 Step-wise Leakage

So called step-wise leakage is an important attack to consider for long-term secrets. For a step-wise leakage attack, the adversary tries to learn seemingly little information, e.g., only one bit per run of the same secure program. For instance, the secure program could output a CRC of a long-term secret for each run. Alternatively, the adversary could output a hash depending on a (random) small part of a long-term secret. The idea is to have hash-values where the input has low entropy and can thus be guessed. Consequently, computing a hash function on data that depends on long-term secrets is forbidden. Step-wise leakage is the reason why no leakage at all is allowed for long-term secrets.

5.3.3 Leakage on Asymmetric Keys

Bitwise leakage on asymmetric keys is a much more complex matter than on symmetric keys. E.g., if 16 bits of a secret exponent in a 1024 bit modulus leak, this will, very likely, not degrade the security by 16 bit. On the other hand, attacks like Coppersmith's [23] allow to reconstruct RSA keys from a surprising small amount (roughly 27%) of the key bits. The conservative assumption for the symbolic execution is based on the worst case: If the expected work for the adversary was 2^{80} before, it is 2^{64} in the worst case assuming that the 16 leaked bits help him to reduce the search space as much as they possibly can.

5.3.4 Related-key Attacks

Related-key attacks (for secret key encryption) are attacks where the adversary does not only get encryptions Enc(K, D) for arbitrary adversarial chosen data D and a key K unknown to the adversary, but additionally encryption Enc(f(K), D)) for a function f picked by the adversary. AES (in particular AES-256) is known to be susceptible to related-key attacks where the function f is a XOR of the key K with a value A. For details about the attack see Alex Biryukov and Dmitry Khovratovich's article [24]. While the XOR of keys with a bit-vector is important for some use cases which need to be supported by the trust provisioning process, XOR for keys is restricted to random or a small set of bit-vectors which renders all known related-key attacks on AES useless.

5.3.5 User-generated Secrets

User-generated secrets are secrets that are not input to the secure program, but generated during its execution in the high security device. These secrets are either generated from fresh randomness or derived from other secrets through key derivation. The use of secrets generated from fresh randomness can be tracked with the current ruleset described above. For derived user-generated secrets this is much harder. To track the use of a derived key, we also need to track that the same key was not derived using different means such that it exists twice. E.g., a secure program that is just a CMAC can be built by just using a MAC. Then the key also exists as data and could be leaked without the system noticing. One countermeasure for this would be to require specific "derivation keys" that may only be used for key derivation. However this also reduces the flexibility for the customer. Hence we have decided to use a general key type for derived secrets (SecretKey). This prevents further usage of these secrets in the secure program. They are tracked as secrets which prevents accidental leakage but they are not necessarily protected against malicious leakage.

5.3.6 Sub-field References

The symbolic execution (mostly) abstracts from the details of addresses and the length of fields. In the secure program it is possible to address a sub-field. This leads to several security hazards:

- Reading - and then effectively using - only a part of a key (e.g. a key is a 3kTDEA key in the symbolic execution but only a 1kTDEA key would be used in the real execution).
- Lower entropy than expected (e.g., for hash functions). The symbolic execution calculates the entropy for the full field (and assumes that this is the input to a hash function), but in practice only a sub-field and hence only a part of the entropy is used.
- Reading with wrong data type (e.g., the start address of a field is in a field of type Random but most data is read from beyond the field's boundary and is not random at all).
- Similar to the first scenario, a part of a key may be overwritten with static data to reduce its entropy.

These threats are mainly countered with two security mechanisms:

- Separation of types for data and secret-dependent data. Data that depends on secrets (user-generated secrets / long-term secrets) must only be read/written completely (i.e., using the full field) or not at all.
- Field boundaries are checked and must not be crossed.

Chapter 6

Conclusion and Outlook

The following chapter covers the lessons learned during the conception and implementation of this thesis. Furthermore, an outlook will be given on possible future work that can be done to improve the discussed topics in this thesis.

6.1 Conclusion

Keeping confidential data secure while having a customised program handling and processing this data is a challenging process. The challenge is not only tp prevent the publication of confidential data, but also to verify the generation of this data. Especially when used for industrial purposes, defective output of a secure program leads to corrupted data with, not only, financial impact, but also a reputation loss for the company.
The security guard is a powerful security mechanism hindering malicious or flawed secure programs from being published and furthermore executed. Performance shows that the security guard can run on almost any type of high security device with high memory efficiency and execution time being swift.

6.2 Outlook

The current implementation was not done with performance being the highest priority, but simplicity and comprehensibility. Short-range improvements can be improving the implementation itself. For instance, by using more efficient data types for storing data and therefore reducing memory usage or implementing faster algorithms and to reduce execution time.

6.2.1 Improvements in the Implementation

The performance of the security guard, as discussed in section 5.2, is adequate. The secure programs in the foreseeable future will not be too extensive, meaning exceeding the limited memory of the high security device or pushing the execution time to an unacceptable

amount. But especially the memory usage can be optimised. For instance, some data members in the symbolic field structure, discussed in section 4.10.1, can be dynamically allocated. Another improvement would be a different dynamic data structure managing the symbolic fields and instructions as the currently used basic linked list implementation.

6.2.2 Implementation of the Security Guard outside a High Security Device

Currently only the implementation for the high security device exists. To check if a secure program is approved by the security guard it must be executed on a high security device. This is inconvenient because there have to be additional steps done to get a confirmation that the secure program is secure. Therefore, to have a kind of pre-test before having the actual check by the security guard, an additional implementation of the security guard outside of a high security device would be a convenient addition when generating a secure program. Additionally, it is possible to "debug" the execution of the secure program to see the history of its fields during the execution. Hence this new implementation can provide thorough analysis on why a security violation or other error has been triggered, which eases causal investigation.

6.2.3 Code Generation from a Formal Model

A completely new approach of generating a secure program would be by restricting the parametrisation and sequence.
The sequence of instructions in the secure program, which then are analysed by the security guard, is generated by a Java tool. This tool can be extended by a so called restriction framework, limiting the possible parametrisation of instructions or even limiting the use of instructions itself with regard to preceding instructions. The advantages of this approach are that the person creating the secure program gets instant feedback whether a sequence of instructions is possible or even gets a context sensitive choice of instructions dependent from preceding instructions during creation.

Bibliography

[1] A. Sabelfeld and A. C. Myers, "Language-based Information-flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[2] A. Sabelfeld and D. Sands, "Declassification: Dimensions and Principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[3] M. Bond and R. Anderson, "API Level Attacks on Embedded Systems," *IEEE Computer Magazine*, vol. 34, no. 10, pp. 67–75, 2001.

[4] N. Flemming, H. R. Flemming, and C. Hankin, *Principals of Program Analysis*.

[5] D. Jackson and M. Rinard, "Software Analysis: A Roadmap," *Proceedings of the conference on The future of Software engineering - ICSE '00*, pp. 133–145, 2000.

[6] B. Chess and J. West, *Secure Programming with Static Analysis*. Addison-Wesley Professional, first ed., 2007.

[7] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007.

[8] A. Gosain and G. Sharma, *A Survey of Dynamic Program Analysis Techniques and Tools*, pp. 113–122. Cham: Springer International Publishing, 2015.

[9] M. D. Ernst, "Static and Dynamic Analysis: Synergy and Duality," *WODA 2003 ICSE Workshop on Dynamic Analysis*, pp. 24–27, 2003.

[10] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Latice Model," *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

[11] M. Sintzoff, "Calculating properties of programs by valuation on specific models," in *Proceedings of the ACM Conference on Proving Assertion about Programs*, no. 7(1), pp. 203–207, 1972.

[12] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[13] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven Compositional Symbolic Execution," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4963 LNCS, pp. 367–381, 2008.

[15] F. Böhl, *Symbolic Analysis of Cryptographic Protocols*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[16] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, *Relations among Notions of Security for Public-key Encryption Schemes*, pp. 26–45. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[17] M. Abadi and P. Rogaway, "Reconciling two Views of Cryptography (The Computational Soundness of Formal Encryption)," *JOURNAL OF CRYPTOLOGY*, vol. 15, pp. 103–127, 2002.

[18] V. Cortier and B. Warinschi, "A Composable Computational Soundness Notion," *CCS '11*, pp. 63–74, 2011.

[19] F. Böhl, V. Cortier, and B. Warinschi, "Deduction Soundness: Prove One, Get Five for Free." Cryptology ePrint Archive, Report 2013/457, 2013. http://eprint.iacr.org/.

[20] E. B. Barker, W. C. Barker, W. E. Burr, W. T. Polk, and M. E. Smid, "SP 800-57. Recommendation for Key Management, Part 1: General (Revised)," tech. rep., Gaithersburg, MD, United States, 2007.

[21] L. Chen, "Recommendation for Key Derivation using Pseudorandom Functions," *NIST special publication*, vol. 800, p. 108, 2008.

[22] "Identification Cards – Integrated Circuit Cards – Part 4: Organization, Security and Commands for Interchange," standard, International Organization for Standardization, 2013.

[23] I. K. Salah, A. Darwish, and S. Oqeili, "Mathematical Attacks on RSA Cryptosystem," *Journal of Computer science*, vol. 2, no. 8, pp. 665–671, 2006.

[24] A. Biryukov and D. Khovratovich, "Related-key Cryptanalysis of the Full AES-192 and AES-256," *Advances in Cryptology–ASIACRYPT 2009*, pp. 1–18, 2009.