FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Study of verifiability of an RSA implementation
Student:	Jakub Tetera
Supervisor:	doc. RNDr. Dušan Knop, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

The verification of algorithms is a very important topic in security and computer science in general. The task is to implement and verify an implementation of the RSA algorithm in the Frama-C framework.

The main tasks are the following:
1) Learn the RSA algorithm and provide a high-level description of its basic functionality.
2) Learn the Frama-C framework.
3) Implement a verifiable (simplified) version of the RSA algorithm (in 64-bit arithmetic and with 32-bit prime numbers).
4) Demonstrate the correctness of the partial verification by an SO/DLL attack.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 30 January 2023 in Prague.

Bachelor's thesis

STUDY OF VERIFIABILITY OF AN RSA IMPLEMENTATION

Jakub Tetera

Faculty of Information Technology
Katedra informační bezpečnosti
Supervisor: doc. RNDr. Dušan Knop, Ph.D.
May 10, 2023

Citation of this thesis: Tetera Jakub. *Study of verifiability of an RSA implementation.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

List of Figures

List of Code Listings

*I would like to express my deepest appreciation to all the extraordinary individuals without whom this thesis would never have been possible. Chiefly, I'd like to thank doc. RNDr. Dušan Knop, Ph.D. for his insight, support and openness while supervising this thesis.
I would also like to extend special gratitude to my student colleagues Bc. Lucie Procházková and Bc. Ladislav Louka for their corrections. In addition to this, the countless hours we spent together working on our respective theses have provided much needed motivation and mutual support.
Finally, I'd like to thank my beautiful girlfriend and my parents for their continued support of my studies and their patience in times of need.*

Declaration

In Prague on May 10, 2023 ....

Abstract

This thesis explores the possibilities of formal deductive verification of the RSA encryption algorithm. The theoretical part serves as an introduction to the realm of formal deductive verification and its basic concepts, along with a brief survey of current applications of its concepts in industry. Furthermore, the RSA encryption algorithm and the Frama-C verification environment for C-language programs, which uses annotations and automatic systems for formal verification of theorems, are briefly described. The practical part provides a demonstration of application of concepts explored in the theoretical part using an implementation of the RSA algorithm in the C programming language and its partial verification using Frama-C. The final result of this thesis is therefore a partially verified implementation of the RSA algorithm. Some of the possible utility of such an implementation is demonstrated using an example SO/DLL injection attack against one of its components.

Keywords Formal verification, deductive verification, RSA, Hoare logic, Frama-C, E-ACSL, SO/DLL injection

Abstrakt

Tato práce se zabývá zjišťováním možností formální deduktivní verifikace šifrovacího algoritmu RSA. Teoretická část čtenáře uvede do problematiky formální deduktivní verifikace, představí její základní koncepty a nastíní její současné průmyslové využití. Dále je představen samotný algoritmus RSA a verifikační prostředí Frama-C určené pro programy v jazyce C, které využívá anotací a automatických systémů k dokazování teorémů. Praktická část se zabývá praktickou ukázkou využití konceptů nastíněných v teoretické části pomocí demonstrační implementace algoritmu RSA v jazyce C a jejího částečného ověření pomocí Frama-C a anotačního jazyka E-ACSL. Výsledkem práce je částečně verifikovaná implementace šifrovacího algoritmu RSA. Jeden z praktických přínosů verifikace je pak demonstrován pomocí zastavení ukázkového SO/DLL injection útoku na jednu z komponent této implementace.

Klíčová slova Formální verifikace, deduktivní verifikace, RSA, Hoareova logika, Frama-C, E-ACSL, SO/DLL injection

List of abbreviations

ACSL	ANSI C Specification Language
ANSI	American National Standards Institute
AST	Abstract syntax tree
DLL	Dynamic link library
ETCS	European Train Control System
GCD	Greatest common divisor
LSB	Least significant bit
MSB	Most significant bit
NIST	National Institute of Standards and Technology
OS	Operating system
PKCS	Public Key Cryptography Standard
RNG	Random number generator
SO	Shared object
SSL	Secure Socket Layer
TP	Theorem prover
VCG	Verification condition generator
WSL	Windows Subsystem for Linux

Introduction

Cryptography has always served a critical role in our society. For centuries, human endeavors, organizations and even entire empires hinged upon their ability to communicate in a confidential manner. The importance of this ability cannot be overstated, and is perhaps best illustrated on the instances where encrypted messages had been intercepted and their encryption broken by adversaries. Some of the notable historical cases include Mary Stuart's letters, the breaking of *Kriegsmarine*'s Enigma code in 1941, or the CIA's cold war era VENONA project. At no point in history, however, had the importance of secure encryption been as great as in today's interconnected digital world with its ever increasing volumes of exchanged confidential information – and ever increasing numbers of adversaries.

Numerous cryptographic algorithms have been developed over the years to ensure information confidentiality. One of the most widely used among them is the RSA asymmetric key cryptosystem, first publicly described by Rivest, Shamir and Adleman in 1977 [1]. Its security relies on the computational difficulty of factoring large integers, and it is generally considered secure as long as sufficiently large integers are used as keys. However, the implementation of this algorithm can be tricky. Even subtle errors in any of the algorithm's components may result in an insecure cipher. Mistakes in implementation may also open the program itself to an array of attacks unrelated to cryptography, such as SO/DLL injection or buffer overflow. This may threaten both its operation and the confidentiality of information contained within its memory.

The dangers of such errors are not new, nor are they unique to RSA or other cryptography applications. Indeed, for this very reason, most software is extensively tested before entering service. However, testing can only show the *presence*, not *absence*, of errors. In safety and security-critical applications this may not be sufficient to ensure the required levels of confidence in the system. Therefore, significant effort has been dedicated to ensure the formal and logical correctness of programs and algorithms. This process is called *formal verification*.

Nowadays, two main techniques of formal verification are used in practice – *model checking* [2], which relies on a complete and exhaustive exploration of the mathematical model of the application, and *deductive verification*, which seeks to prove that all possible states and behaviors of the program satisfy formally defined requirements. While formal verification has its limits, especially the latter technique is widely and successfully used in several key industries, such as aviation, nuclear energy, and industrial control in order to reduce both the likelihood of failure and the attack surface for any prospective threats. Therefore, we endeavor to explore the possibilities of this approach in other critical areas, such as encryption.

Objectives

The main objective of this thesis is to explore the viability of developing a partially verified implementation of the RSA encryption using C programming language and Frama-C formal verification framework. Furthermore, the high-level concepts of formal verification will be explored, as well as its use within certain industries. Finally, a possible use case of a verified encryption algorithm will be demonstrated using a mock attack scenario.

Thesis structure

The thesis is divided into five chapters. Chapter 1 introduces the reader into formal verification and its core concepts. The field's theoretical foundations and practical implications are explored, along with the state of the art in industry. Chapter 2 endeavors to present the case for applying formal verification to a widely used cryptographic algorithm, namely the RSA cryptosystem. Chapter 3 provides a brief introduction to Frama-C, an industrialized formal verification framework, which was used for implementing the practical segment of this thesis. Chapter 4 presents a deep dive into our proof-of-concept verified implementation of RSA, outlining the challenges faced, describing solutions used, and providing pen-and-paper proof where required. Chapter 5 provides a demonstration of the results of our work.

Chapter 1
Deductive formal verification

Deductive software verification is a subset of formal verification that works by expressing the correctness, i.e. the expected behavior of the program, as a mathematical statement and then proving it [3]. This verification process is based on a series of logical inferences (or "deduction") from known and firmly defined properties of the program. Seminal works, upon which deductive software verification is based, include Hoare logic, Dijikstra's weakest preconditions, and Burstall's intermittent assertions [4].

Hoare's work introduced the *Hoare triple* of preconditions, commands, and postconditions, which collectively describe how execution of a command in question affects the internal state of the program [5]. Dijkstra later noted that it is possible to compute the minimal precondition that guarantees the required postconditions [6]. Finally, Burstall combined symbolic execution with mathematical induction to prove that any program execution path implies its postcondition. These works are explored in more detail in Section 1.1.

The three works formed the theoretical basis for the first successful deductive verification tools, such as Tatzelwurm for Pascal. Tatzelwurm was published in 1989 and included facilities for proving that the specified preconditions were stronger than the weakest preconditions required to produce the stronger postconditions. Even so, for a long time, the process of formal verification remained a largely pen-and-paper exercise, and the development of tools for a more automated approach frequently happened in isolation on a per-institution basis [4].

Major progress in wider adoption of deductive verification started around the year 2000. A wider community had formed in the field, facilitating easier exchange of ideas and resulting in the development of several frameworks that enabled a complex approach to deductive verification. These include KeY, SPARK, JML, and Frama-C; many of which have found their way into industrial use and are still in active development today [4]. As the field matured, most of them have developed a common array of features, which are described in Section 1.3 and shown on the example of Frama-C which forms the basis of the practical segment of this thesis.

1.1 Theoretical foundations

Formal verification is an idea that is nearly as old as computer science itself. As early as 1949, Alan Turing has demonstrated an approach to proving the correctness of a large routine by breaking it down into individual mathematical operations, which are in turn individually proven using standard mathematical and logical methods [7]. Turing has even included a full flow diagram of his routine, as well as the map of possible program and memory states, which are used to support his proof.

However, as program complexity increases, this and similar approaches quickly become complex, arduous, and time-consuming. In fact, with many common programs, the effort required to provide such proofs verges on impossible due to the sheer number of individual elements that need to be proven, raising the need for automating this process where possible.

1.1.1 Hoare logic

In 1969, C. A. Hoare proposed a logical basis for evaluating the execution states of computer programs in his article titled *An axiomatic basis for computer programming* [5]. He observed that the results of executions of a program can be expressed using the relationship between a set of *preconditions*, i.e., the known state of the environment before an execution of a program, the program execution itself, and a set of *postconditions*, which describe the results of the execution. Using mathematical notation, this can be expressed as follows:

$$P\{Q\}R$$

Here, P is the set of preconditions, Q is the program execution, and R is the set of postconditions. Hoare also provided the following natural language explanation of this statement:

▶ **Theorem 1.1.** *If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion.*

This theorem provides the basis for deductive verification. If we are able to express the initial requirements of a program and its end result using assertions, we can prove its correctness by showing that an execution with given preconditions will always result in required postconditions being established. According to Hoare, two elements are required for such proof of correctness – *Axioms* which have already been proven, and *inferences* that enable deductive proof of additional statements. The latter take the form of *"If $\vdash X$ and $\vdash Y$ then $\vdash Z$"* statements, i.e. *"If X is proven, and Y is proven, then Z is also proven."*

Hoare has provided several axioms and inferences in his original paper, which had been later expanded in subsequent works by both Hoare and other researchers.

1.1.2 Weakest preconditions

The concept of weakest preconditions was proposed by E. W. Dijkstra, initially as a supplementary notion in order to facilitate a divide-and-conquer solution to a class of sequence generation problems, rather than as a program verification tool [6]. The main observation is that it is possible to recursively calculate the weakest precondition required for a complex program to establish a required postcondition as an union of the weakest preconditions of all its constituent elements.

This allows the simplification and partial automation the proofs of even large algorithms – at least to an extent. It is sufficient to prove that the stated preconditions for any given complex program are stronger than any and all of the preconditions of each of its components.

1.1.3 Intermittent assertions

The intermittent assertion method further expands on the idea of weakest preconditions by extending it with techniques from symbolic execution and mathematical induction [8]. The final goal of this method is to, in essence, prove that the program in question implies its postconditions via a series of implications. Intermittent assertions usually take the following form:

At a point A in the program, the assertion T holds.

The correctness of a program segment (or an entire program) can be expressed using the following intermittent assertions. In this instance, A denotes the entry point of a segment, B

denotes the exit point[1], x is an input variable, y is an output variable, $P(x)$ is a precondition and $T(x, y)$ is a desired postcondition:

▶ **Theorem 1.2.** *If A is reached and $P(x)$ holds, then B will be reached and $T(x, y)$ will hold.*

Notably, this form of assertion requires that both the entry and exit points have to be reached, otherwise, the implication is rendered moot. Therefore, an additional lemma needs to be proven independently for each program loop:

▶ **Lemma 1.3.** *If at some time C is reached, $a = x$ and $Q_1(a)$ holds, then at some time C will be reached and $Q_2(a, x)$ will hold.*

$Q_1(a)$ and $Q_2(a, x)$ denote the properties of x and a required by the loop's operation. C denotes a breakpoint within a loop, which will be reached during each iteration.

The intermittent assertion method is particularly useful for automated proofs of the properties of loops. It allows us to leverage structural induction on partially ordered finite sets in order to prove these properties on an incremental basis for each loop iteration [9]. In particular, loop termination can be efficiently proven with proper choice of provable properties.

1.2 Practical usage

While the three seminal works have provided effective approaches to manually proving the correctness of programs (and indeed, that was their initial intention), their main benefit lies in the fact that, unlike earlier pen-and-paper methods, they lend themselves very well to automation. When used in conjunction they allow us to specify the high-level expected behaviors of a program and provide a deductive proof of correctness with a series of well-defined automated steps.

In modern practice, the verification process starts by collecting a comprehensive set of formally described requirements for algorithm correctness. This includes the required high-level behaviors of the algorithm (e.g., the desired mathematical properties of the output), as well as low-level ones (e.g., an absence of memory leaks or invalid read operations). Once established, the requirements are translated into the so-called *function contracts*, which describe the collective preconditions, postconditions and intermediate assertions required for the proof.

Function contracts usually take the form of assertions written in an annotation language, placed at various stages of the program. Their purpose is to enable the automated deductive verification capabilities built into the frameworks. The exact form and syntax used by each framework vary but they follow the same general structure. This can be demonstrated using a publicly available code snippet from the documentation of a modern verification framework, Frama-C, and its associated annotation language, ACSL [10].

In the example algorithm in Listing 1.1 – a function that sets all elements of an array to zero – ACSL annotations are written in blocks delimited by `/*@` and `*/`. The first block that precedes the function signatures specifies the function contract. It describes the required preconditions (in the `requires` clause), the assigned variables (in the `assigns` clause) and the postconditions that this function will establish (`ensures`).

The second block contains the contract for the main loop of this algorithm. It contains assertions that are required to prove its termination and correctness. First two assertions contain *loop invariants* - conditions that must hold before entering the loop and hold upon exiting the loop. Furthermore, any of its locations which was allocated outside of the loop body must preserve its value. In this case, i must be at least 0 and at most n, and at each iteration, zero must be assigned to the given array position. This is followed by an *assignment declaration*, which tracks the variables whose values are modified by the loop. Finally, a *loop variant* is included. A loop variant denotes an expression whose value must strictly decrease at every iteration of the loop,

[1]Assuming a single exit point for the segment.

Code listing 1.1 Annotated program example

```
#include <stddef.h>
/*@
  requires \valid(a+(0..n-1));
  assigns  a[0..n-1];
  ensures
  \forall integer i;
    0 <= i < n ==> a[i] == 0;
*/
void set_to_0(int* a, size_t n){
  size_t i;

  /*@
    loop invariant 0 <= i <= n;
    loop invariant
    \forall integer j;
      0 <= j < i ==> a[j] == 0;
    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for(i = 0; i < n; ++i)
    a[i] = 0;
}
```

while staying non-negative throughout the execution. It is required to prove the termination of a loop.

A program annotated in this way is then processed by the chosen verification framework. An intermediate normalized representation of the program is generated, which is subsequently analyzed by the framework's proving tools. The exact approach varies between frameworks, both in architecture and in functionality[2]. Finally, a report is generated, which contains the proof status of each individual building block of the program. Generally, a contract will be marked as *valid* (i.e., all of its requirements are conclusively satisfied at every possible state in the program's flow), *invalid* (i.e., some of its requirements are conclusively unsatisfied at some possible state), or *inconclusive*. The resulting report can then be used by the program's authors to discover previously unseen erroneous or otherwise undesired behaviors, examine the program's flow in depth, or to serve as a conclusive proof of correctness of critical software.

1.3 Industrial use

Initially, automated verification was mostly the domain of academic endeavors. The *Tatzelwurm* [11] tool for Pascal, developed at University of Karlsruhe in 1995, is one of the first notable projects in this regard. In many ways it can be viewed as a direct precursor to today's software verification platforms. It is built using the same general approach and with the same general elements, albeit with some notable limitations, such as the heavily restricted set of Pascal commands available. It features a specification language which is used to specify preconditions and postconditons in the form of *contracts*. Contracts are, in turn, used by an automated verification condition generator (or VCG) to generate the Hoare logic formulae necessary to verify the program. Finally, an automated theorem prover (or TP) is used to show the correctness of the input program based on both manual and auto-generated preconditions and postconditions.

[2]Frama-C's approach is provided in Section 3.1

The adoption of deductive verification gained pace around the year 2000 with the birth of several large and long-lasting projects, including some that are still in active development to this day [4]. The first notable example is SPARK [12], a formally defined and verifiable programming language based on Ada. Initially developed at University of Southampton based on requirements by the UK Department of Defense, SPARK is intended to be used in high integrity applications, such as real-time control systems, avionics, healthcare, and defense and military technology. SPARK follows a similar structure to Tatzelwurm in that it includes a restricted set of Ada commands extended with specification language, a VCG, and a TP, which are used to develop complete, unambiguous and provably partially correct programs.

SPARK was quickly adopted by, in particular, the aeronautics and defense industries, due to their need for highly dependable software with minimal margins for error. Many high profile NATO defense projects rely on SPARK for verifying their critical software components written in ADA. Notable examples include the Eurofighter Typhoon attack aircraft, or Lockheed Martin's Mk. 41 Vertical Launch System for various shipborne missiles. In the civilian sector, SPARK had been used in avionics for modern aircraft (e.g., Boeing 777 and 787), safety systems such as the ETCS, and multiple space vehicles developed by the European Space Agency [13]. Similar verification tools are also available for other languages, such as JML for Java, which largely follow a similar approach to program verification as the aforementioned examples.

Chapter 2

RSA and deductive verification

RSA is one of the most prevalent cryptosystems used for secure data transmission. Designed in 1977 by Rivest, Shamir and Adleman [1], its main objective was to provide the capability for secure encryption while also solving the key distribution problem inherent to all symmetric key ciphers. In most earlier cryptosystems, before secure communication can be established, the keys themselves needed to be exchanged first via a secure channel. RSA addresses this via *public key cryptography* – a concept first attributed to Whitfield Diffie and Martin Hellman in 1976 [14].

Suppose that Alice wants to receive an encrypted message from Bob, with a third person, Eve, intercepting their communications. In this scenario, public key cryptography allows Alice to generate a *key pair* comprised of a public (encryption) key and a private (decryption) key. This is done in such a way that neither of the keys can be used to derive the other. The private key is kept secret by Alice, while the public key is transmitted to Bob through any channel. Bob uses the public key to encrypt the message for Alice, which is then sent to her through a potentially insecure channel. Eve is able to intercept the public key and the message, but since key is not a valid decryption key (as it would be in a symmetric cryptography scenario) and cannot itself be used to derive the decryption key, it cannot be used to decrypt the message. Alice and Bob's secrets therefore remain safe.

To achieve the properties above, RSA relies on modular exponentiation and the computational difficulty of factoring large numbers, as described below. The resulting algorithm is relatively easy to use and very secure in practice, so long as it is correctly implemented. However, it has a number of prerequisites that need to be satisfied by its inputs, as well as many potential pitfalls that make it exceedingly difficult to implement securely — to the point that custom implementations are often discouraged within the software development community.

Deductive verification can help eliminate a large class of such issues. It can provide a conclusive proof that the implementation contains neither low-level mistakes nor high-level architectural faults that may be used for attacks.

2.1 The RSA algorithm

The operation of RSA, as described by Rivest, Shamir and Adleman is as follows [1]:

To encrypt a message M using RSA, a public key comprised of two positive integers (e, n) is used. First, the message is converted into a sequence of integers. Then, each member of the sequence is raised to the power of e modulo n, producing the ciphertext C:

$$C \equiv M^e \pmod{n} \tag{2.1}$$

To decrypt a ciphertext C, it can be raised to the power of d modulo n, resulting in a decryption key of (d, n).

$$M \equiv C^d \pmod{n} \tag{2.2}$$

The key generation is somewhat more complex. First, a pair of two very large primes p and q is selected (and kept secret). Then, their product n, which serves as the modulus for both encryption and decryption, is calculated and released.

$$n = p \cdot q \tag{2.3}$$

The next step is to calculate Euler totient function value $\phi(n)$, which will serve as an input for subsequent generation of the private exponent d.

$$\phi(n) = (p - 1) \cdot (q - 1) \tag{2.4}$$

Then, either the encryption exponent e or the decryption exponent d is chosen arbitrarily, with the other being derived later. In the original paper, d is the exponent of choice. However, in most practical applications, e is the arbitrarily chosen exponent. The reasons for this are security and execution speed. Since common binary exponentiation algorithms work faster when a small exponent with low Hamming weight is chosen, it is convenient to choose such an exponent. If this exponent was to be the decryption key, an attacker would have a significantly easier time guessing it, resulting in insecure encryption.

The chosen exponent (e in this case) must satisfy the following requirements:

$$2 < e < \phi(n) \tag{2.5}$$

$$\gcd(e, \phi(n)) = 1 \tag{2.6}$$

The lowest (and fastest) such value is 3. In practice, the value of 65537 ($2^{16} + 1$) is frequently used, as it partially mitigates the danger of sub-optimal padding, while retaining a low Hamming weight and thus enabling faster operation.

Finally, the remaining exponent (in our case d) is derived as a modular multiplicative inverse of the arbitrarily chosen exponent.

$$d \equiv e^{-1} \pmod{\phi(n)} \tag{2.7}$$

Rivest, Shamir and Adleman have also included a mathematical explanation and proof of their algorithm, leveraging Euler's generalization of Fermat's little theorem [1, Chapter 6]

2.2 Common implementation flaws

A successful implementation of RSA requires the developer (and the user) to make many design decisions which influence the algorithm's properties in subtle, but meaningful ways. Unfortunately, many common optimizations and parameter choices could result in degraded security. Worse still, this may not be immediately apparent in many cases. There have been instances of such insecure implementations resulting in widespread vulnerabilities, such as the vulnerable random number generator (RNG) included in a vast number of commonly used smart cards and which is described in depth in Section 2.2.2.

2.2.1 Mistakes from implementation complexity

Practical implementations of RSA can be highly complex. For instance, if the cipher is to have sufficient strength, it has to operate with very large integers. For common applications the United States of America, the National Institute for Standards and Technology (NIST) recommends the use of at least 2048-bit RSA keys [15]. As many languages do not possess native implementations of sufficiently wide integers, developers must frequently resort to use of third-party libraries, or implement their own large number arithmetic and handling procedures.

As the complexity of a program increases, so does the likelihood of vulnerabilities from design choices or developer mistakes. Potential issues range from application instability to enabling information disclosure or even code execution (e.g., via buffer overflow). Mistakes of this type are known to occur occasionally even in commonly used RSA libraries. An example of this would be a buffer overflow vulnerability affecting certain versions of the popular OpenSSL library, which may result in compromise of RSA-1024 under certain (albeit unlikely) circumstances [16].

Even in cases where third party libraries are used to provide some of the required functionality, vulnerabilities within them may transitively expose the RSA implementation to attacks. Erroneous library usage also opens the application to previously unavailable attacks, such as SO/DLL injection.

2.2.2 Poor parameter choice

RSA requires the developers to choose several parameters – namely p, q and e (or d). The way in which they are selected is integral to the cipher's security. Over time, many flawed choices of numbers have appeared in public projects. For example, in 2017, a developer for *Salt*, an infrastructure management tool written in Python, has selected 1 as the value for public exponent e. The resulting implementation effectively performed no encryption and transmitted plaintext information [17].

For p and q, many vulnerabilities arise from insufficient RNG entropy during their generation. If two keys share one prime but not the other (i.e. $p = p'$ but $q \neq q'$), an attacker can easily factor both corresponding public moduli by computing their GCD (p) and dividing to find q and q', respectively. From there, calculating the private exponent can be done using the normal RSA key generation approach. Such flaws are particularly common in embedded systems, whose usage patterns may result in reduced entropy available to the RNG and whose low computing power frequently leads developers to seek questionable optimizations. In 2012, Heninger et al. [18] have performed a study aimed at cracking publicly available SSH and TLS certificates. They have discovered that a small proportion of devices (0.40% of the sample) produced keys factorable in this manner. Almost invariably, these were embedded devices presumably using insecure RNG implementations.

Perhaps the most widely publicized example of poor entropy in random number selection was a vulnerability affecting a range of cryptographic hardware produced by Infineon Technologies produced between (at least) 2004 and 2017. The vulnerability, dubbed "Return of Coppersmith's attack" (ROCA), by the researchers that discovered it, lay in the specific method that was used to obtain primes for use in RSA key generation. All of them were generated with the following form in order to optimize the generation process:

$$p = k \cdot M + (65537^a \bmod M) \tag{2.8}$$

While the integers k and a were not known, the value of M was calculated as a product of first n primes, where n was fixed for each key size. Since the result of this operation was rather large compared to key size (e.g., 219 bits for 256-bit primes used in RSA-512), the amount of entropy was greatly reduced – in the case of RSA-512, the reduction was from 256 to 99 bits. The researchers were then able to use their knowledge of public keys, along with a modification of Coppersmith's algorithm for finding small roots of modular equations, to factorize the entire prime, and thus compromise the private key. In practice, keys up to 2048 bits were found to be vulnerable. The approximate financial cost of such attack to an attacker using common cloud services was found to be between $0.06 (for RSA-512) and $40,000 (for RSA-2048) [19].

The publication of this vulnerability had severe practical consequences – individuals and organizations using the faulty hardware were faced with the possibility of compromise by a determined attacker. This included approximately 750,000 citizens of Estonia, who were issued vulnerable smart personal ID cards [20].

2.2.3 Faulty hardening

RSA possesses certain mathematical properties that make its plain (unharedened) implementation vulnerable to a variety of attacks. For instance, short messages and low encryption exponents (i.e. $m < n^{1/e}$) result in ciphertext smaller than the modulus and thus trivially breakable as the eth root of the ciphertext. Another class of attacks leverages the fact that plain RSA is not semantically secure, making it vulnerable to chosen plaintext attacks (by encrypting likely plaintexts using the public key and checking whether they are equal to the attacked ciphertext). Other attacks include side-channel attacks via differential power analysis, timing attacks [21], or branch prediction analysis [22].

For these (and other) reasons, RSA implementations are frequently hardened by including padding, implementing extra operations in order to break the relationship between key values and side-channel readings, as well as other similar hardening constructions.

In addition to the increased complexity of a hardened implementation, (which in turn leads to many of the issues described in Section 2.2.1), the hardening mechanisms may themselves be flawed. This results in a false sense of security at best and additional avenues for attacks at worst. Notably, an earlier padding scheme in use with RSA – PKCS #1, up to version 1.5 – enabled adaptive chosen ciphertext attacks against Secure Socket Layer (SSL) session keys. Using repeated decryption requests and the fact that only very few ciphertexts resulted in PKCS #1-compliant plaintexts (beginning with bytes 0x00 and 0x02), it has been demonstrated that it is possible to reconstruct the key within practical timeframes [23].

2.3 Benefits of a verified implementation

Deductive verification can potentially help developers mitigate the impact of some of the common pitfalls listed above. Common programming mistakes (e.g., buffer overruns, variable overflows, flow control errors...) can frequently be conclusively ruled out in many cases. Using modern verification frameworks, a number of such issues can be even avoided using auto-generated assertions and contracts at next to no additional development effort cost. SPARK and Frama-C both include provisions for automatically detecting a range of common faulty behavior including integer overflows and invalid memory reads. This feature is also present in most other commonly used frameworks.

Expanding upon auto-generated assertions with custom ones can improve the implementation's integrity even further. The validity of input parameters can be checked, ruling out parameter choice outside of specification. Moreover, output properties of such an algorithm can be guaranteed to an extent, increasing the trustworthiness of such an implementation.

Verification also has benefits which arguably cannot be empirically measured. It helps foster a culture of consistency, attention to detail within the development team, as well as a reliability-focused mindset in development. This (as well as all previously listed benefits) makes verification especially desirable in applications where safety and security is mission critical, or where retroactive remediation of implementation flaws is not viable.

Unfortunately, verification is not the proverbial silver bullet for algorithm correctness. Some of the more subtle (and more dangerous) flaws can be difficult or impossible to prevent using common verification. For RSA specifically, it is difficult to verify properties such as the security of padding or RNG entropy, or similar subtle flaws. As such, it remains important to not rely on a "silver bullet" method of ensuring program correctness, but to adhere to established good development, testing, and lifecycle management practices.

Chapter 3

Frama-C

The Frama-C framework, developed at the behest of a consortium of French public companies, is emblematic of modern automated software verification frameworks. It is widely used in development of safety and security-critical C language programs. Notable users include Airbus, as well as the French *Commissariat à l'Énergie Atomique et aux Énergies Alternatives*. It has been chosen as the verification framework selected for the practical segment of this thesis due to its high degree of maturity and a wide array of features.

3.1 Architecture

On the top level, Frama-C is comprised of a kernel, which parses annotated C code into a normalized representation, and various plugins that leverage this normalized representation in order to perform various individual verification tasks [24]. This section contains the detailed breakdown of its structure.

3.1.1 Annotations

In addition to application code itself, deductive verification requires the application's authors to be able to provide specifications for their expected behavior. In most modern software verification frameworks, this takes the form of *function contracts*. Frama-C is no exception in this regard. It uses an annotation language called ACSL, or *ANSI/ISO C Specification Language*. According to its authors, ACSL is heavily inspired by the JML specification language for Java applications in both form and function. Its main goal is to provide a convenient way for *"writing the kind of properties that make up a function contract"* [10].

ACSL annotations are presented in the form of special purpose comment blocks, enabling code compatibility with regular C-language compilers. They contain precise formal specifications of the underlying code's behavior in a manner both understandable to human programmers and convenient for automated tool operation. A comprehensive set of verification features is provided by ACSL. These include low-level verification primitives for low-level C language features (e.g., provisions for pointer arithmetics and/or verifying pointer validity), arithmetic functions (including arbitrary precision arithmetic), high-level abstract reasoning constructs such as logical predicates, and provisions for defining custom advanced verification logic.

An example of ACSL annotated code is published on the Frama-C website [10]. This same example is included in Listing 1.1.

3.1.2 Kernel

The Frama-C kernel's purpose is to process ACSL contracts and specifications and integrate them with the underlying code. It parses both native C code and its ACSL annotations and produces a normalized representation in the form of an abstract syntax tree (AST) [24]. This is required to provide an unified basis for subsequent operations, both by the kernel itself and by various verification plugins.

In addition to this main purpose, the kernel serves as an orchestration hub within the framework. It is responsible for managing the analysis sequence by scheduling and executing capabilities from various plugins. It also provides features such as serialization and deserialization of analysis results, enforcing separation of various analyses, maintaining analysis integrity or providing means for communication between various plugins. The intent of this functionality is to provide interoperability between analysis plugins while maintaining operational integrity and user comfort [24].

Another benefit of this solution is the ability to easily swap the framework's components or integrate additional plugins as required. Notably, multiple automatic prover plugins are provided with Frama-C out of the box, each best suited for different use cases.

3.1.3 Plugins

The vast majority of program verification tasks in Frama-C is handled by various plugins. Many of them are bundled as standard with ever Frama-C installation, others are developed and distributed separately. They cover a wide variety of functionality, ranging from core automated theorem proving to providing additional toolsets for specialized applications such as smart cards. Only the plugins leveraged for the purposes of this thesis will be described in more detail. The full list of available plugins can be found at Frama-C's website [25].

3.1.3.1 EVA

EVA, which stands for *Evolved Value Analysis*, is a plugin designed to compute variable variation domains[1] [26]. It serves both as a basis for the operation of other plugins and as a standalone tool. In the latter use case, its main purpose is to infer the impossibility of variable-related runtime errors such as integer overflows. EVA will emit a warning when a possibility of such an error is detected, with the absence of warnings serving as a guarantee of absence of errors.

During the course of development of our partially verified RSA implementation (described later in Chapter 4), EVA was used during initial debugging to validate the results of arithmetic operations. Several possible integer overflows have been detected and corrected in this manner.

3.1.3.2 RTE

RTE automatically generates ACSL assertions protecting each program expression against undefined behavior [27]. This includes (but is not limited to) arithmetic overflows, invalid pointer dereference, out-of-bounds array access. In many ways, it provides the same functionality as the EVA plugin described above. Indeed, the plugin's manual explicitly states that parallel use of EVA and RTE is unnecessary. The main difference is that RTE does not automatically attempt to prove its assertions, but always relies on external plugins such as RTE or E-ACSL.

3.1.3.3 WP

WP is the plugin responsible for proving that ACSL contracts are valid for all possible execution paths of the code [28]. It does this by leveraging weakest precondition calculus, and can be

[1]In layman's terms, the range of possible values for program variables

extended with a variety of external prover tools and proof assistants. It contains its own built-in prover – `qed` – but this can be supplemented or supplanted by any prover compatible with the `why3` prover platform, such as the interactive prover `coq`. According to its creators, it should serve as a strong alternative to traditional unit testing. Our proof-of-concept uses the plugin in this capacity as well.

3.1.3.4 E-ACSL

E-ACSL is an example of a plugin that does not perform deductive verification per se. Instead, its purpose to is to generate C code that performs runtime verification of ACSL assertions – even in cases where deductive verification was inconclusive. This is very useful in cases when full verification is impossible (e.g., due to unverified components or prover limitations). A program generated by E-ACSL will terminate if an assertion is violated. Otherwise, it will behave in a manner identical to the original program. This is somewhat analogous to the defensive programming technique of putting C-language assertions in critical segments of code. The main benefit is that ACSL allows us to specify significantly more complex behaviors with comparative ease, potentially enabling the use of runtime assertions even in places where regular C assertions would be impractical.

This thesis shows some of these possibilities by demonstrating a program to fail when an invalid parameter is provided to an annotated function. A detailed demonstration is provided in Chapter 5

3.2 Installation and usage

Frama-C was natively developed for Linux, although it can be run on Windows using `WSL2` and on Mac using `Homebrew`. It is available through a third party package manager called `OPAM`. Full installation guidelines are available at the project's website [29].

For the setup of our local environment (Ubuntu 20.04 under WSL2), the following steps were executed, as specified in the guidelines:

Code listing 3.1 Frama-C installation

```
sudo apt install make m4 gcc opam

opam init --disable-sandboxing --shell-setup
eval $(opam env)
opam install depext

opam depext --install lablgtk3 lablgtk3-sourceview3
opam depext --install -y frama-c
```

Following installation, Frama-C can be started in GUI mode using the following command:

```
GDK_BACKEND=x11 frama-c-gui <options>
```

The exact syntax depends on the required behaviors and plugins. For the purposes of this thesis, the following options have been used. This allows us to perform static deductive verification with GUI.

```
GDK_BACKEND=x11 frama-c-gui -wp  -rte <source_files> -cpp-extra-args
    ="-I<include_paths>"
```

Additionally, we have used the bundled utility for compilation with E-ACSL

```
 e-acsl-gcc.sh -c --rte=all -E"<compilerOptions>" -l"<linkerOptions
    >"
```

The specific commands used during the development of our proof-of-concept are provided as needed in Chapter 5.

Chapter 4
Implementation

In order to demonstrate the utility of a verified implementation of an encryption algorithm, we've developed a simplified proof-of-concept RSA library in C. This implementation was then annotated using ACSL, and a static partial deductive verification using Frama-C's `wp` and `rte` plugins has been performed. This has been combined with the `E-ACSL` plugin, which provides runtime condition verification capability, and which allowed us to prevent an attack using a spoofed component.

In this chapter, the implemented algorithm will be broken down to its individual building blocks, which include their C code, as well as their ACSL verification annotations. For each block, a brief explanation is provided.

4.1 Design overview

Our implementation has been developed as a C library, using Rivest, Shamir and Adleman's original RSA specifications [1]. It is self contained and consists of encryption/decryption functions, a key generation function, and various mathematical functions required for RSA's operation. This chiefly includes the extended Euclidean algorithm, the Rabin-Miller probabilistic primality test, and seveal functions for performing modular arithmetic without integer overflows. The only external dependency is the random number generator. A wrapper for Windows and Linux built-in RNG implementations is provided. The Windows implementation uses the cryptographically secure `rand_s()` function from Microsoft Visual C++, and the Linux implementation retrieves its random data from `/dev/urandom`.

All internal functions are annotated using ACSL and have been partially verified using Frama-C to ensure their correctness. The explanations (as well as the results) for the annotations of notable functions are included in subsequent sections.

As a proof-of-concept, the implementation comes with several limitations. Encryption using this library supports keys and data of up to 64 bits of length. This is due to the fact that longer inputs would require an implementation of arbitrary precision arithmetic. This would result in considerably increased complexity of all operations without meaningfully contributing to the main goals of this thesis. For similar reasons, neither hardening techniques such as padding nor countermeasures for side channel attacks are implemented. This implementation is therefore not to be used in any real-world cryptographic application unless further effort is made to strengthen it.

4.2 Development

The proof-of-concept had been developed using a method outlined in an earlier thesis by Tomáš Homola [30]. His method, dubbed the *Minimal contract method*, operates in several steps:

1. Develop a plain C-language implementation.
2. Perform automated verification and correction of runtime behaviors.
3. Verify preconditions and postconditions to function contracts.
4. Verify loop behaviors.
5. Verify memory manipulation verification.
6. Verify final specifications of high-level behavior (e.g., mathematical relationships between outputs).

We have leveraged a similar approach, with minor modifications to accommodate our specific needs where required. These modifications are described further in this section.

4.2.1 Basic implementation and automated verification

The proof-of-concept application has been developed incrementally on a function-by-function basis. Upon completion of each function, we have performed automated variable value analysis using Frama-C's `eva` and `rte/wp`[1] plugins to highlight possible undesirable behaviors. This behavior had then been corrected before performing another round of verification. In some ways, variable analysis was used as a more powerful alternative unit testing. It allowed us to guarantee the absence of undefined behavior under all possible execution conditions, as opposed to under a limited set of test inputs.

The benefits of this approach can be demonstrated on the case of an early version of the `modMultiply()` function. An addition operation contained an edge case which might have resulted in an integer overflow for too large operands. The `rte` plugin has generated an integer overflow assertion. The `wp` plugin has subsequently attempted to prove this assertion, but was predictably unable to do so. This is indicated by the orange blip in the highlighted section in a screenshot from the Frama-C GUI included in Figure 4.1. Once this mistake had been corrected through including an additional check, the verification output in Figure 4.2 shows that an overflow is no longer possible.

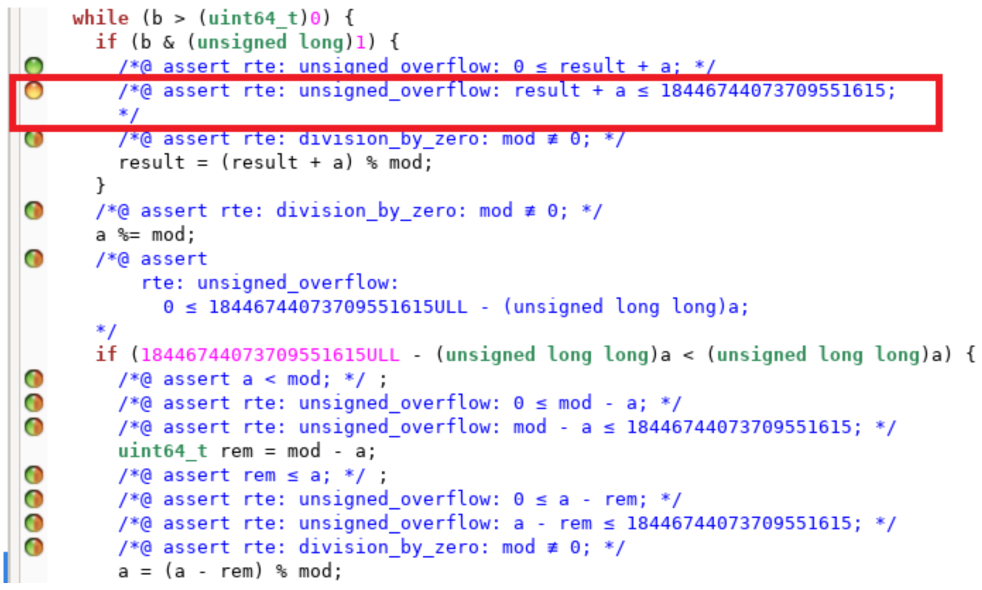
4.2.2 Manual verification using contracts

Most basic verification steps were performed in a manner analogous to Homola's approach to verifying Jarník's spanning tree algorithm. Preconditions and postconditions have been specified using ACSL's `requires` and `ensures` clauses, respectively. For loop verification, `loop variants` (strictly decreasing loop variables) have been used to verify termination. Higher-order specifications, such as the inductive definition of greatest common denominator, have been specified using the `inductive` and `predicate` clauses.

In some cases, our functions' desired behaviors depend on inputs. Frama-C allows us to specify multiple separate behaviors using the `behavior` notation. Each behavior specified in this way contains an `assumes` clause, which describes the conditions specific to each respective behavior.

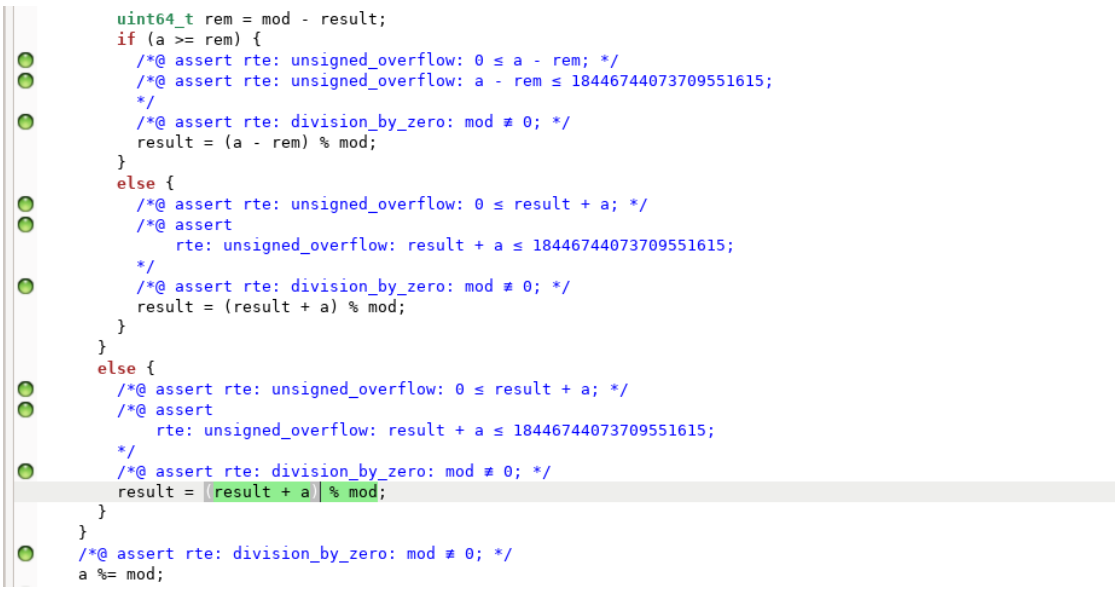
[1]The former was used for cursory checks during development, as it executes faster than the latter

Figure 4.1 Example of generated assertions with possible overflow



```
while (b > (uint64_t)0) {
  if (b & (unsigned long)1) {
    /*@ assert rte: unsigned_overflow: 0 ≤ result + a; */
    /*@ assert rte: unsigned_overflow: result + a ≤ 18446744073709551615;
    */
    /*@ assert rte: division_by_zero: mod ≢ 0; */
    result = (result + a) % mod;
  }
  /*@ assert rte: division_by_zero: mod ≢ 0; */
  a %= mod;
  /*@ assert
      rte: unsigned_overflow:
        0 ≤ 18446744073709551615ULL - (unsigned long long)a;
  */
  if (18446744073709551615ULL - (unsigned long long)a < (unsigned long long)a) {
    /*@ assert a < mod; */ ;
    /*@ assert rte: unsigned_overflow: 0 ≤ mod - a; */
    /*@ assert rte: unsigned_overflow: mod - a ≤ 18446744073709551615; */
    uint64_t rem = mod - a;
    /*@ assert rem ≤ a; */ ;
    /*@ assert rte: unsigned_overflow: 0 ≤ a - rem; */
    /*@ assert rte: unsigned_overflow: a - rem ≤ 18446744073709551615; */
    /*@ assert rte: division_by_zero: mod ≢ 0; */
    a = (a - rem) % mod;
```

Figure 4.2 Example of generated assertions without overflow



```
    uint64_t rem = mod - result;
    if (a >= rem) {
      /*@ assert rte: unsigned_overflow: 0 ≤ a - rem; */
      /*@ assert rte: unsigned_overflow: a - rem ≤ 18446744073709551615;
      */
      /*@ assert rte: division_by_zero: mod ≢ 0; */
      result = (a - rem) % mod;
    }
    else {
      /*@ assert rte: unsigned_overflow: 0 ≤ result + a; */
      /*@ assert
          rte: unsigned_overflow: result + a ≤ 18446744073709551615;
      */
      /*@ assert rte: division_by_zero: mod ≢ 0; */
      result = (result + a) % mod;
    }
  }
  else {
    /*@ assert rte: unsigned_overflow: 0 ≤ result + a; */
    /*@ assert
        rte: unsigned_overflow: result + a ≤ 18446744073709551615;
    */
    /*@ assert rte: division_by_zero: mod ≢ 0; */
    result = (result + a) % mod;
  }
}
/*@ assert rte: division_by_zero: mod ≢ 0; */
a %= mod;
```

4.2.3 Practical considerations

We have found that some elements of full formal verification using ACSL would result in significant increase of effort required with few additional benefits compared to a pen-and-paper proof of the algorithm. In these instances, we have opted to perform the latter and provide references to the pen and paper proof. This was done in the interest of practicality and applicability of our approach, especially with regards to possible adoption beyond the academic sphere.

Furthermore, Frama-C was occasionally unable to reason about validity of certain assertions. Notably, pointer validity in loop or recursive call scenarios appears to pose difficulties to the `wp` plugin. In those cases, the validity was manually verified and the reasoning behind it was included as a code comment. However, the faulty verification may propagate throughout the chain through preconditions and nested calls, causing subsequent verification to fail as well. The validity of failed verification was also reasoned manually using pen-and-paper methods. The results of such reasoning are similarly either included as code comments, or as a commentary in Section 4.4 of this chapter.

4.3 File structure

Individual functions are separated into header and corresponding implementation files based on their purpose within the library. There are four main structural elements – `RSAMath`, which handles the arithmetic required for RSA's operation, `RSAUtils`, which contains the wrappers for calling the operating system's RNG APIs, `RSAKeygen`, which contains functions integral to key generation, and `libRSAVerified`, which handles the main cryptographic functionality.

4.4 Algorithm structure and verification

4.4.1 Encryption and decryption

RSA encryption and decryption is performed using simple modular exponentiation. The encryption and decryption functions are therefore included mainly for ease or use. As such, they act as a wrapper for the modular exponentiation function `modEx()`, described in Section 4.4. They take pointers to buffers holding the plaintext/ciphertext, the key values, and a destination buffer for each operation's result.

Due to the aforementioned fact that these functions are merely wrappers for other verified functions, their ACSL contracts only include provisions for pointer validity. No additional proof of mathematical properties of the modular exponentiation function within them is necessary.

The code for both functions can be seen in Listing 4.1

4.4.2 Extended Euclidean algorithm

The extended Euclidean algorithm serves the purpose of finding Bézout's coefficients required for calculating the modular multiplicative inverse and determining whether they exist. This widely known algorithm recursively calculates the greatest common divisor of two integers, retroactively computing the coefficients after each iteration. The implementation used is one that is widely available, and as such, has been both described and proven multiple times in literature. One such proof and description had been done by Andreas Klappenecker of Texas Agricultural and Mechanical University [31]. Based on this proof, we have included an inductive construct in the Frama-C annotation that proves that our C-language implementation – `eea()` – does indeed find the greatest common divisor of the inputs.

Code listing 4.1 Encryption and decryption code

```
/*@
@ requires \valid(pubKeyBuf) && \valid(ptBuf) && \valid(ctBuf);
*/
int8_t rsa_encrypt(uint64_t e, uint64_t* pubKeyBuf, uint64_t* ptBuf,
    uint64_t* ctBuf)
{
        return modEx(*ptBuf, e, *pubKeyBuf, ctBuf);
}

/*@
@ requires \valid(pubKeyBuf) && \valid(privKeyBuf) && \valid(ctBuf) && \
    valid(ptBuf);
*/
int8_t rsa_decrypt(uint64_t *pubKeyBuf, uint64_t* privKeyBuf, uint64_t*
    ctBuf, uint64_t* ptBuf)
{
        return modEx(*ctBuf, *privKeyBuf, *pubKeyBuf, ptBuf);
}
```

`rte` has highlighted two possibilities for an overflow during the calculation of the greatest common denominator, requiring additional checks to remedy. However, this overflow is mathematically impossible. A general idea of a proof of this fact can be found in literature [32]. We have used this proof as an inspiration to prove this for our own implementation.

▶ **Lemma 4.1.** *If $gcd(a, b) = t = ax + by$, then $|x| \leq |b/2t|$ and $|y| \leq |a/2t|$.*

Proof. By induction on n, where n is the number of recursive iterations of the extended Euclidean algorithm. For the moment, let's assume that $t = \gcd(a, b) = 1$ and $b > a > 0$. The algorithm then completes in n rounds.

For $n = 0$, $a = 1$ and $b > a$, $x = 1$ and $y = 0$. In this case, it trivially holds that $0 \leq \frac{a}{2t}$ and $1 \leq |\frac{b}{2t}|$ for all $(b > a) \in \mathbb{Z}$

If $b > a > 1$ and the algorithm completes in a single round ($n = 1$), since $t = 1$, we know that there exists k such that $b = ka + 1$, $k \in \mathbb{Z}$

Our algorithm calculates the coefficients from the $n = 0$ case as follows:

$$x = 0 - \frac{ka + 1}{a} \cdot 1 = -\frac{ka + 1}{a}$$

$$y = 1$$

Since $a \geq 2$, we know that $|y| \leq |\frac{a}{2}|$. $|\frac{ka+1}{a}| \leq |\frac{b}{2}|$ Holds as well, since $b = ka + 1$ and $a \geq 2$, giving us:

$$|\frac{ka + 1}{a}| \leq |\frac{ka + 1}{2}.$$

In the inductive case, we're assuming coprime a and b such that $b > a$ and $n + 1$ recursive rounds of the algorithm are required. This means that there exists r such that $b = ka + r$, $k, r \in \mathbb{Z}.$, $b > r > 1.$[2] Furthermore, we know that the nth application of the algorithm calculates $\gcd(r, a)$, that $\gcd(r, a) = \gcd(a, b) = 1$, and that in the recursive chain, the round $n = 1$ will eventually be reached. Let x_n, y_n therefore be the solutions to Bézout's identity $xr + ya = 1$. The base case assumption for $n = 1$ is the following:

[2] $r = 1$ is covered by the base case

$$|x_n| \le |\frac{a}{2}|$$
$$|y_n| \le |\frac{r}{2}|$$

For $n+1$, our algorithm calculates the coefficient as follows:

$$x_{n+1} = y_n - \frac{b}{a} \cdot x_n$$
$$y_{n+1} = x_n$$

We need to prove that $|x_{n+1}| \le |\frac{b}{2}|$ and $|y_{n+1}| \le |\frac{a}{2}|$. The latter assertion holds thanks to the fact that $y_{n+1} = x_n$ and $|x_n| \le |\frac{a}{2}|$. For the former:

$$|\frac{b}{2}| \le |y_n - \frac{b}{a} \cdot x_n|$$
$$|\frac{ka+r}{2}| \le |y_n - \frac{ka+r}{a} \cdot x_n|$$

If we apply base case assumptions, the "worst case" scenario for our proof is if y_n and x_n have opposite signs. That would make the right hand side expression's absolute value $|y_n + (\frac{(ka+r)}{a} \cdot x_n)$.

$$|\frac{ka+r}{2}| \le |\frac{r}{2} + \frac{ka+r}{a} \cdot \frac{a}{2}|$$
$$|\frac{ka}{2} + \frac{r}{2}| \le |\frac{r}{2} + \frac{(ka+r)a}{2a}$$
$$|\frac{ka}{2}| \le |\frac{ka+r}{2}|$$

Because $r > 1$, our assertion holds and the proof is complete. Proof for cases when $\gcd(a,b) = t \neq 1$ is unnecessary, because that simply means that $\gcd(a,b) = t \cdot \gcd(a/t, b/t)$ and the coefficients don't change in any way. ◀

The function contract itself is merely a formal specific of preconditions and postconditions of the algorithm's operation. Under all circumstances, it is required that the buffers for Bézout's coefficients are valid (to prevent C runtime errors) and disjointed (to ensure that they are being written correctly by the function). Furthermore, in order to prove that the recursive call of this function will terminate, a `decreases` clause is used. This clause behaves in a manner similar to a `loop variant` - it ensures that the variable's integer value strictly decreases between calls [33].

The individual behaviors describe the preconditions and postconditions of the Euclidean algorithm. `behavior valid_non_zero` describes the conditions for a standard step, whereas `behavior zero` describes the base case scenario. Both behaviors are disjointed, and combined, they cover all possible inputs (as described by their `assumes` clauses).

The code for this algorithm can be found in Listing 4.2.

Code listing 4.2 Extended Euclidean algorithm code

```
/*@
@ inductive is_gcd(integer p, integer q, integer d) {
@       case gcd_zero:
@               \forall integer n; is_gcd(n, 0, n);
@       case gcd_else:
@               \forall integer p,q,d; is_gcd(q, p%q, d) ==> is_gcd(p, q
    , d);
@ }
*/
/*@
@ requires \valid(x) && \valid(y);
@ requires x != y;
@ decreases (a);
@ behavior non_zero:
@       assumes a != 0;
@       assigns *x, *y;
@       admit ensures \result == (*x * a) + (*y * b);
@       ensures \valid(x) && \valid(y);
@       ensures x != y;
@       ensures is_gcd(a, b, \result);
@ behavior zero:
@       assumes a == 0;
@       assigns *x, *y;
@       ensures \result == \old(b);
@       ensures X_eq_1: *x == 0;
@       ensures Y_eq_0: *y == 1;
@       ensures \valid(x) && \valid(y);
@       ensures x != y;
@       ensures is_gcd(a, b, \result);
@ complete behaviors;
@ disjoint behaviors;
@ */
uint64_t eea(uint64_t a, uint64_t b, int64_t* restrict x, int64_t*
    restrict y) {
        if (a == 0) {
                *x = 0;
                *y = 1;

                //@ assert *x==0;
                //@ assert *y==1;
                return b;
        }

        int64_t xTemp = 0;
        int64_t yTemp = 0;

        uint64_t gcd = eea(b % a, a, &xTemp, &yTemp);

        // Cannot possibly overflow or underflow. Proven in thesis text.
        *x = yTemp - ((b / a) * xTemp);
        *y = xTemp;

        return gcd;
}
```

4.4.3 Modular arithmetic

4.4.3.1 Modular multiplication

The first modular arithmetic function requiring formal verification in our implementation is `modMultiply()`, the 64-bit modular multiplication function. Despite the fact that the end result of this operation is at most 64-bit wide (i.e., the same or lower width than the modulus), the naïve approach[3] may result in an integer overflow. To solve this issue, we leverage a variant of the *long multiplication* algorithm.

Numbers represented in binary can be converted to decimal[4] using a sum of powers of two. This is illustrated by Expression 4.1:

$$42_{10} \equiv 101010_2 \equiv (1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} \tag{4.1}$$

The associative properties of multiplication allow us to express the multiplication of a decimal number with a binary number as shown in Expression 4.2:

$$(5 \cdot 42)_{10} \equiv (5 \cdot 1 \cdot 2^5 + 5 \cdot 0 \cdot 2^4 + 5 \cdot 1 \cdot 2^3 + 5 \cdot 0 \cdot 2^2 + 5 \cdot 1 \cdot 2^1 + 0 \cdot 2^0)_{10} \tag{4.2}$$

A more general expression of sum is shown in Expression 4.3 below:

$$\sum_{n=0}^{length(b)} (a \cdot 2^n) \cdot b_n \tag{4.3}$$

Our algorithm operates as a sum similar to Expression 4.3, with the modulus operator applied at various points of the sum in order to prevent overflows. The algorithm's code can be seen in Listing **??**. It operates in a loop, processing the binary digits of the second multiplicant (b) in a loop. For each digit, if this digit is one, it adds the current value of the first multiplicant (a) to the intermediate result. At the end of each iteration, the current value of a is multiplied by two to obtain the next value ($a \cdot 2^n$). The modulus operator is then applied to the result to reduce its width, and the next digit of b is shifted in for the next operation.

This algorithm is not free from edge cases. Both the addition `(r + a)` and multiplication `(a * 2)` can produce an overflow. The included C code has provisions for detecting and remediating such conditions.

Frama-C is mainly used to prove that the function is without side effects and will terminate. This is done using a loop variant and `assigns` clauses. Additional assertions are present throughout the algorithm to enforce more stringent requirements and prove the absence of undefined behavior.

An admittance was made about the validity of the function's final output in its contract, which is automatically treated as valid in subsequent operation. The reason for this is the difficulty of finding a loop invariant due to the properties of modular arithmetic. An automated proof using Frama-C would add significant difficulty to the verification process compared to a pen-and-paper proof, with few benefits for the purposes of this thesis.

The code can be found in listing 4.3

[3]Multiplying the operands with each other and feeding the result into the modulus operator
[4]Or, indeed, any other base

Code listing 4.3 Modular multiplication code

```
/*@
@ predicate isModMultiply(integer r, integer a, integer b, integer m) =
@       r == (a * b) % m;
@*/

/*@
@ requires \valid(resBuf);
@ behavior good:
@       assumes mod != 0;
@       assigns \result, *resBuf;
@       admit ensures isModMultiply(*resBuf, a, b, mod);
@       ensures \result == 0;
@ behavior bad:
@       assumes mod == 0;
@       assigns \result;
@       ensures \result == -1;
@ complete behaviors;
@ disjoint behaviors;
*/
int8_t modMultiply(uint64_t a, uint64_t b, uint64_t mod, uint64_t *
    resBuf) {

        // A modulus of zero would cause undefined behavior
        if (mod == 0) {
                return -1;
        }

        // Convert a to its modular equivalent
        a = a % mod;

        uint64_t result = 0;


        // Start long multiplication based on B's binary digits
        // -------------------------------------------------------
        /*@ loop assigns result, a, b;
                @ loop variant b;
        @*/
        while (b > 0) {

                if (b & 1) {

                        // Edge case - possible overflow during addition
                        // If that happens:
                        //      - Take the difference from result to
                            modulus
                        //      - Remove it from a and take the remnant
                            as the new result (this is essentially an
                            implicit modulo operation)
                        if((UINT64_MAX - a) < result){

                                // Paranoid (and possibly superfluous)
                                    check to ensure that result is
                                    smaller than mod
                                result = result % mod;
                                uint64_t rem = mod - result;
                                // Additional verbose check to enable
                                    Frama-C's proof of impossibility of
                                    overflows
                                if (a >= rem){
                                        result = (a - rem) % mod;
                                }
                                else {
                                        result = (result + a) % mod;
                                }

                        }
                        else {
                                // If there's no overflow, this
                                    operation is enough
                                result = (result + a) % mod;
                        }
                }

                a = a % mod;
                // Edge case: possible overflow during multiplication
                // The solution is the same as above
                if ((UINT64_MAX - a) < a) {
                        //@ assert a < mod;
                        uint64_t rem = mod - a;
                        // Modulus cannot be larger than 2a - wouldn't
                            fit into 64bit int
                        //@ assert rem <= a;
                        a = (a - rem) % mod;
                }
                else {
                        // If there's no overflow, this operation is
                            enough
                        a = (a * 2) % mod;
                }

                // Next binary digit
                b = b >> 1;
        }

        *resBuf = result % mod;

        return 0;
}x
```

4.4.3.2 Modular exponentiation

Modular exponentiation is handled by the `modEx()` function. It is performed using the well-known square-and-multiply algorithm, adapted for modular arithmetic. This algorithm is based on a recursive application of the following observations:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \\ x \cdot ((x^2)^{\frac{n-1}{2}}) & \text{if } n \text{ is odd} \end{cases}$$

This allows us to calculate higher order exponents using their binary representation. Numbers with least significant bit set represent odd exponents, and those with least significant bit (LSB) set represent even exponents. Furthermore, bitwise shifting a value n by one position to the right produces the value $\frac{n}{2}$ for odd integers and $\frac{n-1}{2}$ for even ones. This allows us to perform the exponentiation operation iteratively by either squaring the base a if the current exponent is even, or squaring and multiplying if the current exponent is odd. The next bit of the exponent is then shifted in, and the operation is performed again for the new exponent. The algorithm's termination is guaranteed by the fact that zeroes are shifted to the most significant bit of the exponent during each iteration. At some point in the operation, the exponent's value will reach zero and the algorithm will terminate.

ACSL assertions serve a purpose similar to those included in all other functions in Section 4.4.3. They ensure loop termination, preempt side effects and provide protections against runtime errors and undefined behavior.

Throughout the operation, the value is reduced to fit into a 64-bit variable by applying modular multiplication instead of regular multiplication. As with modular multiplication, this behavior significantly increases the effort required to produce a loop invariant compared to a pen-and-paper proof. An admittance has therefore been included in the contract based on the pen-and-paper proof included in this section to simplify the verification process.

The code can be found in Listing 4.4.

4.4.3.3 Modular multiplicative inverse

The final modular arithmetic function, `modInverse()`, is the function for computing the modular multiplicative inverse of a 64-bit integer. A modular multiplicative inverse z of an integer a modulo m is defined as follows:

$$az \equiv 1 \pmod{m}$$

To calculate it, our code Bézout's coefficients previously calculated using the extended Euclidean algorithm[5]. Bézouts coefficients belonging to a pair of integers a, b are numbers x, y such that:

$$ax + by = \gcd(a, b)$$

If a and m are coprime, i.e. $\gcd(a, mod) = 1$, and if we apply the modulo operation to both sides, we may try to find their Bézout's coefficients in the following manner:

$$ax + my = 1 \pmod{m}$$

Since $my \equiv 0 \pmod{m}$, we know that $ax \equiv 1 \pmod{m}$. Bézout's coefficient x is therefore the modular multiplicative inverse of a modulo m. If a and mod aren't coprime, a modular multiplicative inverse does not exist. `modInverse()` therefore merely verifies that the inputs are coprime and returns the value of the corresponding Bèzout's coefficient. In order to satisfy the requirements of subsequent operation, the possibly negative value of this coefficient is transformed into its corresponding value in modular representation.

[5]as described in Section 4.4.2

Code listing 4.4 Modular exponentiation code

```
/*@
@ requires \valid(resBuf);
@ behavior good:
@       assumes mod != 0;
@       assigns *resBuf;
@       admit ensures *resBuf == (a^b) % mod;
@       ensures \result == 0;
@ behavior bad:
@       assumes mod == 0;
@       assigns \nothing;
@       ensures \result == -1;
@ complete behaviors;
@ disjoint behaviors;
*/
int8_t modEx(uint64_t a, uint64_t b, uint64_t mod, uint64_t * resBuf) {
        if (mod == 0) return -1;

        // Will be returned if b == 0;
        uint64_t result = 1;
        a = a % mod;

        /*@
        @ loop assigns result, a, b;
        @ loop variant b;
        */
        while (b > 0) {
                if (b & 1) { // if current LSB of B is 1
                        modMultiply(result, a, mod, &result);
                }

                b = b / 2;
                modMultiply(a, a, mod, &a);
        }

        // printf("Result within the function: %lu\n", result);
        *resBuf = result;
        return 0;
}
```

The Frama-C contract mostly serves the purpose of verifying that the function has no side effects in terms of assignments. The algorithm's correctness itself has once again been proven using pen-and-paper methods and supplemented with additional assertions to strengthen its requirements. Furthermore, Frama-C was unable to reason about the validity of preconditions and certain assignments, causing some subsequent verification steps to remain inconclusive.

The code for modular multiplicative inverse can be found in Listing 4.5

4.4.4 Prime number generation

Prime number generation for use in our RSA implementation uses the high-level algorithmic approach described by the pseudocode in Listing 4.6.

Random numbers are sourced from the operating system's RNG API. The call is wrapped inside a custom function, `csRandom()`. The exact specification is determined at compile time based on the target operating system. For Windows, `rand_s()` function from Microsoft Visual C++ is called directly, and the Linux implementation reads its random data from `/dev/urandom`. A Frama-C contract is not included, as there are no properties that can be verified by the framework.

To determine whether a number is prime, the basic Rabin-Miller primality test is used [34]. This test is performed by the `rabinMiller()` function. It is based on verifying that a property that is known to be true for all prime numbers also holds for the number being tested.

▶ **Lemma 4.2.** *Let x, n be integers. n is considered a strong probable prime to base x if one of the following congruences is satisfied:*

$$x^d \equiv 1 (mod\ n)$$

$$x^{2^r \cdot d} \equiv -1 (mod\ n) \text{ for some } 0 \leq r < s$$

With $n = 2^s \cdot d$, s being a positive integer and d being an odd positive integer.

The algorithm which checks whether these congruences are satisfied is based on Rabin's description [34] and optimized via leveraging findings by Webster and Sorenson [35]. Webster and Sorenson have been able to demonstrate that Rabin-Miller testing against a limited set of bases – 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 and 37 – is sufficient to deterministically prove whether a 32-bit unsigned integer is prime. The proofs for both the algorithm itself, and its deterministic modification, are included in their respective papers. Its Frama-C certification therefore only includes automatically generated assertions by the `rte` plugin. Additionally, some RTE assertions could not be proven automatically due to Frama-C's internal limitations. The reasoning behind the correctness of operations marked by these assertions is explained by comments within the code itself.

The prime number generation function – `getPrime()` is the final implementation of the algorithm and performs exactly as specified by the pseudocode in Listing 4.6. Unfortunately, a deterministic formal proof of this function is impossible. Its termination cannot be guaranteed. As such, it also does not include any ACSL specification beyond guaranteeing a non-zero output.

4.4.5 Key generation

The key generation is handled by function `generateKeyPair()`. Its algorithm operates in the manner specified by Rivest, Shamir and Adleman in their original description of the RSA algorithm [1]. It operates with an externally supplied public exponent in the following steps:

1. Obtain a set of 32-bit primes, p and q
2. Calculate the public key modulus as their product

Code listing 4.5 Modular multiplicative inverse code

```
/*@
@ requires mod != 0;
@ requires \valid(resBuf);
@ behavior coprime:
@       assumes is_gcd(a, mod, 1);
@       assigns *resBuf;
@       admit ensures (a * (*resBuf)) % mod == 1;
@       ensures \result == 0;
@ behavior not_coprime:
@       assumes !is_gcd(a, mod, 1);
@       assigns \nothing;
@       ensures \result == -1;
@ complete behaviors;
@ disjoint behaviors;
*/
int8_t modInverse(uint64_t a, uint64_t mod, uint64_t * resBuf)
{
        int64_t x = 0;
        int64_t y = 0;

        uint64_t gcd = eea(a, mod, &x, &y);

        uint64_t result = 0;

        // If a and mod aren't coprime, the modular inverse doesn't
            exist.
        if (gcd != 1) {
                return -1;
        }
        else {
                if (x >= 0) {
                        // If x is positve, it's the modular inverse
                        result = ((uint64_t)x) % mod;
                }
                else {
                        // If the coefficient is negative, we need to
                            find the corresponding modular equivalent
                        uint64_t x0 = (uint64_t)(-(x + 1));
                        result = (mod - 1 - x0) % mod;
                }
        }

        // Write out the result
        *resBuf = result;

        return 0;
}
```

Code listing 4.6 Prime number generation pseudocode

```
algorithm getPrime is:
    main:
        x <- random number
        if x is prime:
            return x
        else
            goto main
```

3. Calculate the Euler totient for p and q
4. Calculate the private exponent as a modular multiplicative inverse of the public exponent

Certification of this algorithm using Frama-C is not possible due to the aforementioned uncertain termination of prime number generation. Furthermore, in edge cases, the Euler totient function value may potentially not satisfy the requirements for existence of modular multiplicative inverse of the public exponent. The only verification included for this function is the pen-and-paper proof provided by Rivest et al. [1], supplemented by automated assertions generated by the `rte` plugin, which were either proven automatically or supplemented with code comments of proof.

4.4.6 Dynamic library *libExponentGenerator*

The `libExponentGenerator` serves the purpose of providing an attack vector for demonstrating the possibilities of the E-ACSL plugin. It contains a single function – `getPublicExponent()` – whose purpose is to generate the exponent e used in RSA encryption. Two implementations of this function are provided – a "non-malicious" version which returns one of the 10 most commonly used exponents, and a "malicious" version which always returns 1, effectively rendering the encryption process moot. Since the library is loaded by the driver application dynamically, the malicious and non-malicious versions can be interchanged by simply replacing the respective shared object or dynamic link library file in the binaries folder. No E-ACSL annotations are present in this library, as it is supposed to mimic a potential unverified third party library in our demo scenario.

4.4.7 Driver

The driver is built to illustrate the capabilities of the developed RSA library, as well as to explore additional possibilities of the E-ACSL plugin. It performs the following operations:

1. Obtains e. It does so by attempting to dynamically load `libExponentGenerator` and calling `getPublicExponent()`. In case of a failure, the value defaults to 65537
2. Attempts up to 100 times to generate a valid key pair using `rsa_generateKeyPair()`
3. Encrypts the plaintext 1234567890 using the generated key pair
4. Decrypts the ciphertext from the previous step using the generated key pair

By design, `libExponentGenerator` is loaded by name without any additional specifications. This results in the system standard load order being applied and potentially making the application vulnerable to various DLL-related attacks. in Chapter 5, we are using the `E-ACSL` plugin to mitigate a subset of such attacks.

Chapter 5

Practical demonstration

A practical demonstration of everything above can be performed using the proof-of-concept code accompanying this thesis. This includes standard operation of the RSA library without Frama-C, static verification using Frama-C plugins `rte` and `wp`, and a dynamic demonstration of the capabilities of E-ACSL in stopping an injection of malicious parameters via .so injection on Linux.

5.1 Standard operation

The proof-of-concept is provided as a CMake project. By default, Linux and Windows are supported. There are no additional dependencies beyond the C standard library, common OS functions, and CMake for build.

5.1.1 Windows

For Windows, the provided project can be built imported into Visual Studio as a CMake project. A CMakePresets.json file is provided, specifying build targets for Windows (x86 and x64 debug/release) and Ubuntu under WSL. It can be built and executed using the IDE's builtin functions, or it can be built directly using CMake from the project root within developer PowerShell. The demo (using the driver program) can be executed by running `DemoDriver.exe` from the build output directory.

5.1.2 Linux

On Linux, the build can be performed via CMake from the root directory. A convenience script – `build_all.sh` is provided with the distribution to simplify the process. If this script is used, it will output all binary files into `./linux_build/bin/`. The demo driver program can then be executed by running `./linux_build/bin/DemoDriver`.

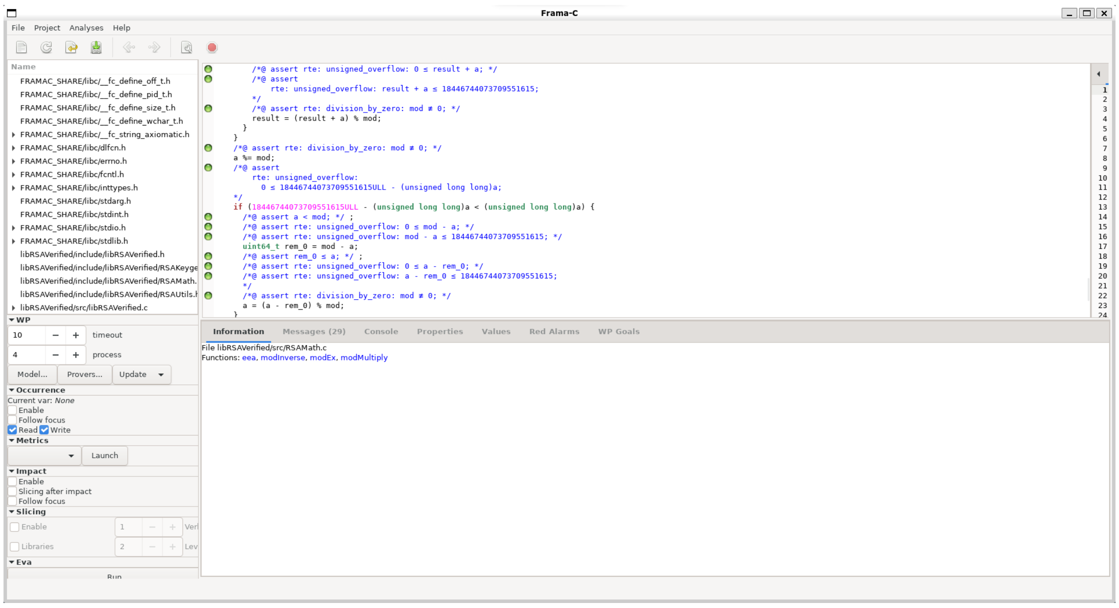
5.2 Static verification

Static verification is done via Frama-C GUI. From the project root directory, the command listed in Listing 5.1 can be executed. This will bring up a window which allows us to explore the code, its ACSL annotations and the results of `wp`'s reasoning about them. The verification status appears as a symbol next to each annotation. Green symbols indicate valid and verified

Code listing 5.1 frama-c-gui command

```
frama-c-gui -wp -rte -warn-unsigned-overflow ./driver/driver.c ./
    libRSAVerified/src/*.c -cpp-extra-args="-I ./libRSAVerified/
    include -I ./libRSAVerified/include/libRSAVerified"
```

Figure 5.1 Example Frama-C GUI screenshot



```
FRAMAC_SHARE/libc/__fc_define_off_t.h
FRAMAC_SHARE/libc/__fc_define_pid_t.h
FRAMAC_SHARE/libc/__fc_define_size_t.h
FRAMAC_SHARE/libc/__fc_define_wchar_t.h
FRAMAC_SHARE/libc/__fc_string_axiomatic.h
FRAMAC_SHARE/libc/dlfcn.h
FRAMAC_SHARE/libc/errno.h
FRAMAC_SHARE/libc/fcntl.h
FRAMAC_SHARE/libc/inttypes.h
FRAMAC_SHARE/libc/stdarg.h
FRAMAC_SHARE/libc/stdint.h
FRAMAC_SHARE/libc/stdio.h
FRAMAC_SHARE/libc/stdlib.h
libRSAVerified/include/libRSAVerified.h
libRSAVerified/include/libRSAVerified/RSAKeyge
libRSAVerified/include/libRSAVerified/RSAMath.
libRSAVerified/include/libRSAVerified/RSAUtils.h
libRSAVerified/src/libRSAVerified.c
```

```
        /*@ assert rte: unsigned_overflow: 0 ≤ result + a; */
        /*@ assert
            rte: unsigned_overflow: result + a ≤ 18446744073709551615;
        */
        /*@ assert rte: division_by_zero: mod ≢ 0; */
        result = (result + a) % mod;
      }
    }
    /*@ assert rte: division_by_zero: mod ≢ 0; */
    a %= mod;
    /*@ assert
        rte: unsigned_overflow:
          0 ≤ 18446744073709551615ULL - (unsigned long long)a;
    */
    if (18446744073709551615ULL - (unsigned long long)a < (unsigned long long)a) {
      /*@ assert a < mod; */ ;
      /*@ assert rte: unsigned_overflow: 0 ≤ mod - a; */
      /*@ assert rte: unsigned_overflow: mod - a ≤ 18446744073709551615; */
      uint64_t rem_0 = mod - a;
      /*@ assert rem_0 ≤ a; */ ;
      /*@ assert rte: unsigned_overflow: 0 ≤ a - rem_0; */
      /*@ assert rte: unsigned_overflow: a - rem_0 ≤ 18446744073709551615;
      */
      /*@ assert rte: division_by_zero: mod ≢ 0; */
      a = (a - rem_0) % mod;
    }
```

File libRSAVerified/src/RSAMath.c
Functions: eea, modInverse, modEx, modMultiply

assertions and yellow symbols indicate inconclusive verification. In contrast, assertions marked in red are conclusively proven to be false. An example Frama-C GUI window showing a segment of our implementation can be found in Figure 5.1.

5.3 Dynamic demonstration

To illustrate the possibilities of runtime assertion checking, we have prepared a dynamic demo that is built along with the main application. As mentioned in Section 4.4.7, the driver application dynamically loads a library that generates the encryption exponent e. This is done to mimic a common scenario in endpoint software, where a third party module is included with the application. During build, two versions of this library are produced – `libExponentGenerator.so` or `ExponentGenerator.dll`, which operates within safe bounds, and `libMaliciousGenerator.so` or `MaliciousGenerator.dll`, whose exponent generation function always returns a value of 1. If `ExponentGenerator` is replaced with a renamed `MaliciousGenerator`, the program will load the malicious library. Under normal circumstances, this results in no visible error – the encryption will execute with an exponent on 1. However, if the executable is compiled with E-ACSL, the assertion violation is detected, and the program is terminated. This behavior can be seen in Figure 5.2 and Listing 5.2.

E-ACSL is currently only available for Linux. The Linux build script `build_all.sh` will build the dynamic demo, and output the results of E-ACSL compilation into `linux_build/dynamic`. Two binaries are produced - `a.out`, which contains the regular (non-validated) program, and `a.out.e-acsl`, which does perform runtime verification.

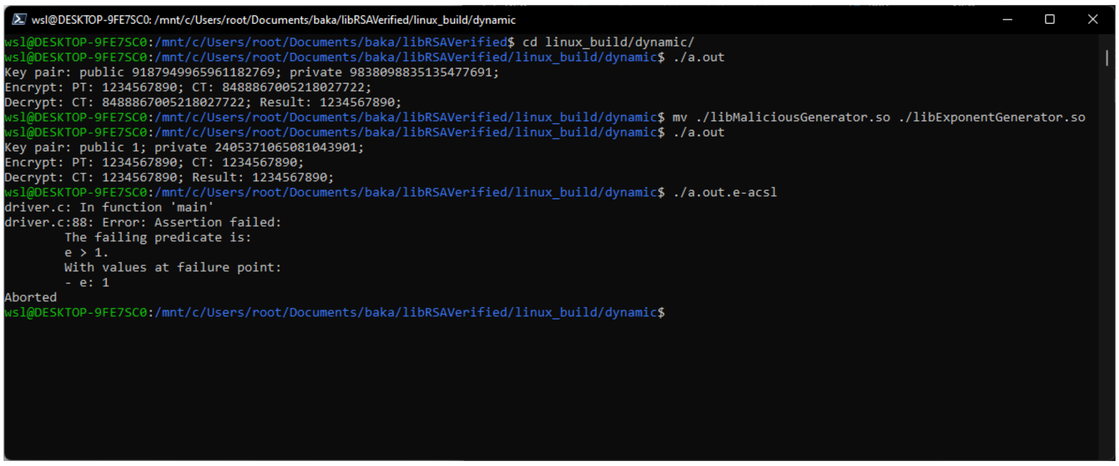


Figure 5.2 E-ACSL demonstration

```
wsl@DESKTOP-9FE7SC0: /mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified$ cd linux_build/dynamic/
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic$ ./a.out
Key pair: public 9187949965961182769; private 9838098835135477691;
Encrypt: PT: 1234567890; CT: 8488867005218027722;
Decrypt: CT: 8488867005218027722; Result: 1234567890;
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic$ mv ./libMaliciousGenerator.so ./libExponentGenerator.so
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic$ ./a.out
Key pair: public 1; private 2405371065081043901;
Encrypt: PT: 1234567890; CT: 1234567890;
Decrypt: CT: 1234567890; Result: 1234567890;
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic$ ./a.out.e-acsl
driver.c: In function 'main'
driver.c:88: Error: Assertion failed:
        The failing predicate is:
        e > 1.
        With values at failure point:
        - e: 1
Aborted
wsl@DESKTOP-9FE7SC0:/mnt/c/Users/root/Documents/baka/libRSAVerified/linux_build/dynamic$
```

Code listing 5.2 E-ACSL demonstration terminal output

```
$ cd linux_build/dynamic/
./linux_build/dynamic$ ./a.out
Key pair: public 9187949965961182769; private 9838098835135477691;
Encrypt: PT: 1234567890; CT: 8488867005218027722;
Decrypt: CT: 8488867005218027722; Result: 1234567890;
./linux_build/dynamic$ mv ./libMaliciousGenerator.so ./
    libExponentGenerator.so
//linux_build/dynamic$ ./a.out
Key pair: public 1; private 2405371065081043901;
Encrypt: PT: 1234567890; CT: 1234567890;
Decrypt: CT: 1234567890; Result: 1234567890;
./linux_build/dynamic$ ./a.out.e-acsl
driver.c: In function 'main'
driver.c:88: Error: Assertion failed:
        The failing predicate is:
        e > 1.
        With values at failure point:
        - e: 1
Aborted
./linux_build/dynamic$
```

Chapter 6
Conclusion

We would like to conclude with a quick review of our stated objectives' completion. In Chapter 1, the fundamentals of deductive verification have been explored. We have briefly explored the three seminal works that form the foundation of the field of formal deductive verification, how those concepts have been brought from theory into practice, and how they have been ultimately adopted within various industries.

Chapter 2 makes the case for the implementation of a verified version of the RSA algorithm. It shows how RSA's intrinsic properties make it difficult to implement properly, which is evidenced by the numerous flawed implementations that have appeared throughout its history. The chapter also outlines the areas where deductive verification techniques can be used to reduce the likelihood of erroneous implementation.

Chapter 3 presents the Frama-C verification framework, which was subsequently used in our implementation of verified RSA. It briefly outlines the architecture and the capabilities of this framework, and presents a few notable functionalities that have been leveraged during the development of this thesis' practical segment.

This is expanded upon in Chapter 4. This chapter describes the thinking behind our verified implementation of RSA. It provides a structural overview of the architecture, as well as in-depth descriptions of the algorithms used, problems that we have faced, and the solutions that we have applied. This is supported by the implementation code, including the annotations for automated verification, and supplemented by pen-and-paper manual proofs of correctness where needed.

Finally, Chapter 5 contains guidance on validating the work outlined in Chapter 4 using the included proof-of-concept application. It shows how to operate the basic verified implementation using the included driver application, how to view the verification results by using the Frama-C GUI, and how to demonstrate the possibilities of runtime verification using E-ACSL

We believe that this marks our stated objectives as achieved. We have shown that although complex, partial formal verification of RSA is possible, viable and useful. We have been able to develop a verified implementation, use Frama-C to preclude undefined or otherwise faulty runtime behavior, and to prove the correctness of its underlying components using a combination of automated and pen-and paper methods.

Nevertheless, there is still room for additional work in this space. Our implementation is strictly proof-of-concept – it is not hardened, and operates merely on 64-bit inputs. This precludes any practical use in its current form. Expanding it to contain hardening techniques (e.g., padding) and to operate using arbitrary precision arithmetic is therefore crucial to further development.

Additionally, the majority of verification performed as a part of this thesis was still done using pen-and-paper methods. This was sufficient to achieve our goals. However, there are opportunities for more stringent verification using Frama-C and ACSL, particularly in the arithmetic functions used. Such work may use our pen-and-paper proofs as a basis to expand upon. Moreover, we have not utilized Frama-C to its full potential in terms of plugins. This could be another area of study that could yield results.

To broaden the possible audience, a verified implementation of RSA could also be developed for other languages and frameworks. Namely, verification using JML could provide significant benefits to the security of embedded devices running Java. Implementation in Ada SPARK could also be beneficial, as SPARK is widely known and used in industrial applications.

Bibliography

1. RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*. 1978, vol. 21, no. 2, pp. 120–126. ISSN 0001-0782. Available from DOI: `10.1145/359340.359342`.
2. CLARKE, Edmund M.; SCHLINGLOFF, Bernd-Holger. Chapter 24 - Model Checking. In: ROBINSON, Alan; VORONKOV, Andrei (eds.). *Handbook of Automated Reasoning*. Amsterdam: North-Holland, 2001, pp. 1635–1790. Handbook of Automated Reasoning. ISBN 978-0-444-50813-3. Available from DOI: `https://doi.org/10.1016/B978-044450813-3/50026-6`.
3. FILLIÂTRE, Jean-Christophe. Deductive software verification. *International Journal on Software Tools for Technology Transfer*. 2011, vol. 13, pp. 397–403.
4. HÄHNLE, Reiner; HUISMAN, Marieke. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by STEFFEN, Bernhard; WOEGINGER, Gerhard. Cham: Springer International Publishing, 2019, pp. 345–373. ISBN 978-3-319-91908-9. Available from DOI: `10.1007/978-3-319-91908-9_18`.
5. HOARE, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM*. 1969, vol. 12, no. 10, pp. 576–580. ISSN 0001-0782. Available from DOI: `10.1145/363235.363259`.
6. DIJKSTRA, Edsger Wybe; DIJKSTRA, Edsger Wybe; DIJKSTRA, Edsger Wybe; DIJKSTRA, Edsger Wybe. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976.
7. Checking a large routine [online]. 1949 [visited on 2023-03-25]. Available from: `https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8`.
8. COUSOT, P.; COUSOT, R. “A la Burstall” intermittent assertions induction principles for proving inevitability properties of programs. *Theoretical Computer Science*. 1993, vol. 120, no. 1, pp. 123–155. ISSN 0304-3975. Available from DOI: `https://doi.org/10.1016/0304-3975(93)90248-R`.
9. VOSATKA, Karel. Intermittent-assertion method as a structural induction. *Kybernetika*. 1979, vol. 15, pp. 122–135. ISSN 0023-5954.
10. *ANSI/ISO C specification language*. CEA LIST, 2023. Available also from: `https://frama-c.com/html/acsl.html`.

11. DEUSSEN, Peter; HANSMANN, Alex; KÄUFL, Thomas; KLINGENBECK, Stefan. The verification system Tatzelwurm. In: *KORSO: Methods, Languages, and Tools for the Construction of Correct Software: Final Report.* Ed. by BROY, Manfred; JÄHNICHEN, Stefan. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 285–298. ISBN 978-3-540-47802-7. Available from DOI: `10.1007/BFb0015468`.
12. *SPARK - The SPADE Ada Kernel.* AdaCore, 2011. Available also from: `https://docs.adacore.com/sparkdocs-docs/SPARK_LRM.htm#_Toc311793130`.
13. FELDMAN, Michael B. *Who's Using Ada? Real-World Projects Powered by the Ada Programming Language.* George Washington University, 2014. Available also from: `https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html`.
14. DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE Transactions on Information Theory.* 1976, vol. 22, no. 6, pp. 644–654. Available from DOI: `10.1109/TIT.1976.1055638`.
15. BARKER, Elaine; DANG, Quynh. *Recommendation for key management, part 3: Application-specific key management guidance.* National Institute of Standards and Technology, 2015. Available also from: `https://csrc.nist.gov/publications/detail/sp/800-57-part-3/rev-1/final`.
16. *CVE-2019-1551.* [N.d.]. Available also from: `https://www.cve.org/CVERecord?id=CVE-2019-1551`.
17. *saltstack/salt commit 5dd304276ba5745ec21fc1e6686a0b28da29e6fc.* [N.d.]. Available also from: `https://github.com/saltstack/salt/commit/5dd304276ba5745ec21fc1e6686a0b28da29e6fc`.
18. HENINGER, Nadia; DURUMERIC, Zakir; WUSTROW, Eric; HALDERMAN, J Alex. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12).* 2012, pp. 205–220.
19. NEMEC, Matus; SYS, Marek; SVENDA, Petr; KLINEC, Dusan; MATYAS, Vashek. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In: *24th ACM Conference on Computer and Communications Security (CCS'2017).* ACM, 2017, pp. 1631–1648. ISBN 978-1-4503-4946-8/17/10.
20. DAN GOODIN - OCT 16, 2017 11:00 am UTC; WISE, Aged Ars Veteran jump to post vintagedave. *Millions of high-security crypto keys crippled by newly discovered flaw.* 2017. Available also from: `https://arstechnica.com/information-technology/2017/10/crypto-failure-cripples-millions-of-high-security-keys-750k-estonian-ids/`.
21. BRUMLEY, David; BONEH, Dan. Remote timing attacks are practical. *Computer Networks.* 2005, vol. 48, no. 5, pp. 701–716. ISSN 1389-1286. Available from DOI: `https://doi.org/10.1016/j.comnet.2005.01.010`. Web Security.
22. ACIIÇMEZ, Onur; KOÇ, Çetin Kaya; SEIFERT, Jean-Pierre. On the Power of Simple Branch Prediction Analysis. In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security.* Singapore: Association for Computing Machinery, 2007, pp. 312–320. ASIACCS '07. ISBN 1595935746. Available from DOI: `10.1145/1229285.1266999`.
23. BLEICHENBACHER, Daniel. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In: *Annual International Cryptology Conference.* 1998.
24. *Frama-C - Architecture.* CEA LIST, [n.d.]. Available also from: `https://frama-c.com/html/kernel.html`.
25. *Frama-C - Plugins.* CEA LIST, [n.d.]. Available also from: `https://frama-c.com/html/kernel-plugin.html`.

26. *Frama-C - EVA*. CEA LIST, [n.d.]. Available also from: `https://frama-c.com/fc-plugins/eva.html`.

27. *Frama-C - RTE*. CEA LIST, [n.d.]. Available also from: `https://frama-c.com/fc-plugins/rte.html`.

28. BAUDIN, Patrick; BOBOT, François; CORRENSON, Loïc; DARGAYE, Zaynah; BLANCHARD, Allan. *WP Plug-in Manual*. 2023. Available also from: `https://frama-c.com/download/frama-c-wp-manual.pdf`.

29. *Frama-C - Installation*. [N.d.]. Available also from: `https://git.frama-c.com/pub/frama-c/blob/master/INSTALL.md`.

30. HOMOLA, Tomáš. *Certifikované algoritmy pro hledání minimální kostry*. 2021. Available also from: `https://dspace.cvut.cz/handle/10467/96906`. Bachelor's Thesis. Czech Technical University in Prague.

31. KLAPPENECKER, Andreas. *The Extended Euclidean Algorithm* [online]. 2006. [visited on 2023-05-01]. Available from: `https://people.engr.tamu.edu/andreas-klappenecker/cpsc311h-f06/exgcd.pdf`.

32. RANKIN, S. A. The Euclidean Algorithm and the Linear Diophantine Equation ax + by = gcd(a, b). *The American Mathematical Monthly* [online]. 2013, vol. 120, no. 6, pp. 562–564 [visited on 2023-05-07]. ISSN 00029890, ISSN 19300972. Available from: `https://www.jstor.org/stable/10.4169/amer.math.monthly.120.06.562`.

33. PATRICK BAUDIN1, Pascal Cuoq; FILLIÂTRE, Jean-Christophe; MARCHÉ, Claude; MONATE, Benjamin; MOY, Yannick; PREVOSTO, Virgile. *ACSL: ANSI/ISO C Specification Language*. 2023. Version 1.18. Available also from: `https://frama-c.com/download/acsl.pdf`.

34. RABIN, Michael O. Probabilistic algorithm for testing primality. *Journal of Number Theory*. 1980, vol. 12, no. 1, pp. 128–138. ISSN 0022-314X. Available from DOI: `https://doi.org/10.1016/0022-314X(80)90084-0`.

35. SORENSON, Jonathan; WEBSTER, Jonathan. Strong pseudoprimes to twelve prime bases. *Mathematics of Computation*. 2016, vol. 86, no. 304, pp. 985–1003. Available from DOI: `10.1090/mcom/3134`.

Enclosed repository contents

- `README.md` Brief description of the repository's contents
- `build_all.sh` Linux build script
- `CMakeLists.txt` Root CMake build instructions
- `CMakePresets.json` CMake presets for Visual Studio
- `driver` Sources for the demo driver application
  - `CMakeLists.txt` CMake instructions for building the demo driver application
  - `driver.c` Demo driver source code
- `libExponentGenerator` Sources for the exponent generator module
  - `include` Header files
    - `libExponentGenerator.h`
  - `src` Source files
    - `libExponentGenerator.c` Implementation of the valid exponent generator
    - `libMaliciousGenerator.c` Implementation of the malicious exponent generator
  - `CMakeLists.txt` CMake instructions for building the exponent generator
- `libRSAVerified` Sources for the main RSA library
  - `CMakeLists.txt` CMake instructions for building the RSA library
  - `include` Header files
    - `libRSAVerified.h` Main RSA header
    - `libRSAVerified` Additional headers, used internally by libRSAVerified
      - `RSAKeygen.h`
      - `RSAUtils.h`
      - `RSAMath.h`
  - `src` Source files
    - `libRSAVerified.c` Source for main RSA functionality
    - `RSAKeygen.c` Source code for key generation functions
    - `RSAUtils.c` Source code for utility functions
    - `RSAMath.c` Source code for arithmetic functions
- `text` Text of the thesis
  - `thesis` LaTeX source
  - `thesis.pdf` Text of the thesis in PDF format