Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations

José Bacelar Almeida[1] Manuel Barbosa[1] Gilles Barthe[2] François Dupressoir[2]
[1]HASLab – INESC TEC and Universidade do Minho
[2]IMDEA Software Institute

Abstract

We present a computer-aided framework for proving concrete security bounds for cryptographic machine code implementations. The front-end of the framework is an interactive verification tool that extends the EasyCrypt framework to reason about relational properties of C-like programs extended with idealised probabilistic operations in the style of code-based security proofs. The framework also incorporates an extension of the CompCert certified compiler to support trusted libraries providing complex arithmetic calculations or instantiating idealised components such as sampling operations. This certified compiler allows us to carry to executable code the security guarantees established at the high-level, and is also instrumented to detect when compilation may interfere with side-channel countermeasures deployed in source code.

We demonstrate the applicability of the framework with the RSA-OAEP encryption scheme, as standardized in PKCS#1 v2.1. The outcome is a rigorous analysis of the advantage of an adversary to break the security of assembly implementations of the algorithms specified by the standard. The example also provides two contributions of independent interest: it is the first application of computer-aided cryptographic tools to real-world security, and the first application of CompCert to cryptographic software.

1. Introduction

The security of computer and communication infrastructures critically relies on software implementations of cryptographic standards. Practitioners implementing such standards face significant challenges. First, they must resolve all cases of underspecification and address operational considerations that are often ignored by standards. Then, they must ensure that the generated code is, not only correct and efficient, but also respects a programming discipline that minimizes the possibility of side-channel attacks. Such disciplines, or countermeasures, may be enforced directly over assembly code, or they may be specified and validated over source code in a language such as C [11]; in this latter case the developer will either trust the compiler to preserve the countermeasures in generated code, or further validation is performed. Unfortunately, there is limited tool support to help practitioners address these issues (see § 6), which makes the development process error-prone. As a consequence of subtle errors, software implementations may provide little or no security [15].

Another problematic technological gap for practitioners lies between the security claimed by cryptographic standards and the concrete security bounds derived using provable security [22]. Indeed, provable security typically relies on an idealized model of computation and elides security-relevant aspects of runtime environments (e.g. memory management) and implementation details (e.g. error management). Although this abstraction gap was identified very early in the development of provable security, principled approaches narrowing this gap are only starting to emerge. Prominent examples include real-world provable cryptography [17, 31], which analyzes realistic descriptions of cryptographic algorithms. However, realism comes at a significant cost: security analyses in these extended models are vastly more complex, and therefore more error-prone and more difficult to check. Moreover, the additional realism achieved by real-world cryptography does not primarily address the aforementioned issues faced by implementers.

In short, there are two significant gaps, with cumulative effects on the real-world security of cryptographic software implementations. This paper addresses the challenge of narrowing these gaps.

Technical overview To achieve this goal, we build on two recent and independent developments: computer-aided cryptography, which provides tool support for provable security, and verified compilation, which delivers machine-checkable evidence of semantic preservation between source and target programs.

To address the gap between cryptographic proofs and standards, we extend the EasyCrypt framework [10] to reason about C-like programs extended with idealised probabilistic operations in an enhanced security model in which the adversary is given access to execution traces meant to capture side-channel leakage. The approach is general, but we focus on the well-known Program Counter Model [29]. The advantage of this approach is twofold. Firstly, one can make explicit in the security proof various aspects of cryptographic scheme specifications that address side-channel attacks. For example, we are able to check that a specification of a decryption algorithm does not reveal information about the secret key by returning a failure value at an early point in its execution. Secondly, this computational model serves as a reference for the deployment of countermeasures against side-channel attacks throughout the compilation process. For example, even though modular exponentiation is treated as a native operation in our high-level computational model, we can make it explicit in its formalization that, in order to ensure security in the Program Counter Model, the trace produced by its execution must be independent of its input values.

To address the second issue, of relating cryptographic standards and their low-level implementations, we leverage the annotation mechanism and semantic correctness proof of the CompCert compiler [26] to prove that security of the C implementation implies security of the assembly implementation and, particularly, that countermeasures against side-channel attacks are correctly deployed *in the generated assembly code*. In addition, we extend CompCert with the notion of a trusted library providing multi-precision arithmetic functionality and instantiations of idealised operations. This allows practitioners to develop and compile their implementations according to common practices, by providing a clean interface that specifies the guarantees required from an external library in order to obtain assembly code that is correct and secure.

The EasyCrypt libraries and formal proofs, the corresponding C code and our extended version of CompCert will be made available online soon.

Contributions We introduce a software development framework that allows practitioners to obtain rigorous mathematical guarantees for low-level (PowerPC, ARM and x86) implementations of cryptographic software. More precisely, our framework enables practitioners to:

1. formally verify that a C implementation of a cryptographic algorithm is secure in a security model that captures both theoretical security and side-channel leakage.
2. automatically generate an optimised assembly implementation that is proven to retain the security properties of the C implementation, namely by correctly deploying side-channel countermeasures suitable for a rigorously defined leakage model.

We illustrate the effectiveness of our framework by proving security and generating a secure assembly implementation of the RSA-OAEP encryption scheme, as standardized in PKCS#1 v2.1. The main challenges in the security proof were the following:

- Formalizing and verifying a security proof taking into account the exact padding scheme adopted in the standard.
- Carrying out this proof in a computational model that incorporates Program Counter Model traces, taking into account the implementations of *all* the algorithms specified in the standard, including encoding and decoding routines, as well as the recommended and mandated side-channel countermeasures.

The process of generating an assembly implementation of RSA-OAEP that inherits the security guarantees established at the C level with EasyCrypt is a fully automatic process using our extended version of CompCert. We thus believe that our extensions are of independent interest to cryptographic software developers deploying side-channel countermeasures at the C level. Intuitively, they permit validating that assembly code leaks *no more information* than the source code via its control flow. This is a non-functional property that is not known to be offered by CompCert in general (nor, to the best of our knowledge, by any other C compiler), and our extension provides this guarantee in the style of translation validation [26]. For example, we have used this feature to check the assembly code generated by compiling various components in the NaCl cryptographic library, which includes relevant side-channel countermeasures at C-level [11].

2. Syntax and security of PKE implementations

Although the techniques we discuss in this paper are generally applicable to implementations of arbitrary cryptographic primitives, we will deal only with public-key encryption (PKE) schemes.

We will be concentrating on real-world implementations of this primitive. This means that, throughout the paper, a PKE *scheme* will provide descriptions of the encryption and decryption algorithms as implementations in either C or assembly code. Our notion of a PKE implementation does not include the key generation algorithm, which will be seen as an abstract algorithm that produces key pairs, and that is modelled as a family of (efficiently samplable) distributions $\mathcal{G}_\lambda$, indexed by the security parameter. We adopt this simplification for ease of presentation and note that, from a theoretical point of view, this is without loss of generality. From a practical point of view, this is also not a major limitation, as in many applications key generation is performed in a trusted environment, using implementations developed independently of the encryption and decryption algorithms. Nevertheless, we also emphasise that all our techniques and results can be easily extended to cover the implementation of the key generation algorithm.

Definition 1 (PKE syntax)**.** *A public-key encryption scheme implementation* Π *is a tuple* $(\mathsf{lang}, \mathsf{Enc}, \mathsf{Dec}, \ell_{(\cdot)})$ *where:*

- $\mathsf{lang} \in \{\mathsf{C}, \mathsf{asm}\}$ *indicates the implementation language.*
- *For each value of* λ, $\mathsf{Enc}_\lambda^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk})$ *implements a deterministic polynomial-time encryption algorithm. On input a message* m *and a public key* pk*, this algorithm outputs a return code* rc *and a ciphertext* c*, possibly after making a series of calls to external functions (oracles)* Rand *and* RO.
- *For each value of* λ, $\mathsf{Dec}_\lambda^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk})$ *implements a deterministic polynomial-time decryption algorithm. On input a ciphertext* c *and a secret key* sk*, this algorithm outputs a return code* rc *and a message* m*, possibly after making series of calls to external function (oracle)* RO.
- $\ell_{\mathsf{pk}}(\cdot)$ *and* $\ell_{\mathsf{sk}}(\cdot)$ *are polynomials denoting the length of the octet strings representing public keys and secret keys, respectively.*
- $\ell_{\mathsf{m}}(\cdot)$ *and* $\ell_{\mathsf{c}}(\cdot)$ *are polynomials denoting the (maximum) length of the octet strings representing messages and ciphertexts* output *by the decryption and encryption algorithms, respectively.*[1]

Here, $\mathsf{Rand}(\ell)$ *returns a random octet string of length* ℓ*, and* $\mathsf{RO}(i, \ell_i, \ell_o)$ *gives access to a family of random functions indexed by* (ℓ_i, ℓ_o)*, where each function takes an octet string* i *of length* ℓ_i *and returns an octet string of length* ℓ_o*. The return codes* rc *are chosen from a finite set, and they indicate the success* (T) *of the operation or its justified failure* $(\mathsf{F}_{(\cdot)})$.

We treat implementations as families of algorithms indexed by the security parameter, which should be interpreted for practical purposes as admitting that the security parameter must be known at compile-time. We deviate slightly from the standard approach of adopting bit-strings as the representation of inputs and outputs to cryptographic algorithms, and adopt octet strings instead. This is without loss of generality. We also restrict our attention to PKEs for which the (maximum) lengths of the inhabitants of the message, randomness and ciphertext spaces are fixed in the description of the scheme (as polynomials in λ) for each value of the security parameter. This is *not* without loss of generality, but is a necessary constraint when dealing with implementations operating in computational platforms with limited memory, and a natural one to make when considering cryptographic standards such as RSA-OAEP.

Correctness The correctness of a PKE scheme implementation requires that the decryption operation inverts the encryption procedure. We present this definition using a code-based approach, and again emphasise that a scheme Π may be given as a C implementation or as an assembly implementation. The notion of correctness is the same in either case.

Definition 2 (PKE correctness)**.** *Let game* $\mathsf{Corr}_{\Pi,\mathcal{A}}$ *be as shown in Figure 1. The correctness of a PKE scheme* Π *relative to* $\mathcal{G}$ *requires that, for all adversaries* $\mathcal{A}$ *and for all* λ*, we have that*

$$\Pr[\, \mathsf{Corr}_{\Pi,\mathcal{G},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T} \,] = 1 \,.$$

Game $\mathsf{Corr}_{\Pi,\mathcal{G},\mathcal{A}}(1^\lambda)$:
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_\$ \mathcal{G}_\lambda$
$\mathsf{m} \leftarrow_\$ \mathcal{A}(\mathsf{pk}, \mathsf{sk})$
If $|\mathsf{m}| > \ell_{\mathsf{m}}(\lambda)$ Return T
$(\mathsf{rc}, \mathsf{c}) \leftarrow_\$ \mathsf{Enc}_\lambda^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk})$
$(\mathsf{rc}', \mathsf{m}') \leftarrow \mathsf{Dec}_\lambda^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk})$
Return $\mathsf{rc} = \mathsf{T} \wedge \mathsf{rc}' = \mathsf{T} \wedge \mathsf{m}' = \mathsf{m}$

Figure 1. Game defining the correctness of a PKE scheme.

[1] We do not restrict the size of the message and ciphertext *inputs* to these algorithms, and require implementations to check the validity of their lengths.

We note that security games are presented here at a level of abstraction that omits certain details of the implementations, e.g., how parameter passing is handled. Such details are made explicit in the formalization of the model, as described in the next section.

Security For security, we consider the standard notion of indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA), adapted to fit in our syntactic conventions and with an extension to capture side channel leakage.

Definition 3 (IND-CCA security of a PKE implementation)**.** *Let game* $\mathsf{IND\text{-}CCA}_{\Pi,\mathcal{G},\mathcal{A}}$ *be as defined in Figure 2. The* IND-CCA *security of a PKE scheme* Π *relative to* $\mathcal{G}$ *requires that, for all adversaries* $\mathcal{A}$*, the following advantage definition is negligible:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\Pi,\mathcal{G},\mathcal{A}}(\lambda) := 2 \cdot \Pr\left[\mathsf{IND\text{-}CCA}_{\Pi,\mathcal{G},\mathcal{A}}(\lambda) \Rightarrow \mathsf{T}\right] - 1\,.$$

Game $\mathsf{IND\text{-}CCA}_{\Pi,\mathcal{G},\mathcal{A}}(1^\lambda)$:

$b \leftarrow_\$ \{0,1\}$
$(\mathsf{pk},\mathsf{sk}) \leftarrow_\$ \mathcal{G}_\lambda$
$(\mathsf{m}_0,\mathsf{m}_1,\mathsf{st}) \leftarrow_\$ \mathcal{A}_1^{\mathsf{Decrypt},\mathsf{RO}}(\mathsf{pk})$
$(\mathsf{rc},\mathsf{c}^*) \leftarrow_\$ \mathsf{Enc}_\lambda^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}_b,\mathsf{pk}) \rightsquigarrow t$
$b' \leftarrow_\$ \mathcal{A}_2^{\mathsf{Decrypt},\mathsf{RO}}(\mathsf{c}^*,t,\mathsf{st})$
Return $(|\mathsf{m}_0| = |\mathsf{m}_1| \wedge b' = b)$

oracle $\mathsf{Decrypt}(\mathsf{c})$:

$(\mathsf{rc},\mathsf{m}) \leftarrow \mathsf{Dec}_\lambda^{\mathsf{RO}}(\mathsf{c},\mathsf{sk}) \rightsquigarrow t$
Return $(\mathsf{rc},\mathsf{m},t)$

Figure 2. Game defining the IND-CCA security of a PKE scheme. An adversary $\mathcal{A}$ is legitimate if $\mathcal{A}_2$ does not call Decrypt on c^*.

The $\rightsquigarrow t$ notation in Figure 2 denotes information that is leaked to the adversary by the execution environment, in addition to the outputs explicitly produced by the PKE algorithms.[2] We recover the standard definition of security for PKEs when this leakage is the empty string ϵ for all algorithms.

Leakage models We will model leakage as a trace of constant identifiers that reveal some information about the control flow of the executed algorithm. This approach allows us to capture some leakage models that are relevant for practical applications, and can be extended to deal with arbitrary types of leakage by defining different types of observable events in the semantics of the implementation languages, and controlling if and how this information is revealed to the adversary.

For concreteness, we will focus on two leakage models for C and assembly implementations. The leakage model that we adopt at the C level is chosen for its simplicity, and can be seen as an adaptation of the Program Counter Model [29] (PC) to C programs. The leakage model that we adopt at the assembly level is the standard PC model. Informally, these models are defined as follows:

- C **leakage.** We associate to each branching point in the C program two observable events with identifiers Event_True and Event_False. One such event is added to the (initially empty) trace whenever a branch is taken, consistently with the Boolean value of the evaluated branching condition.
- **Assembly leakage.** Each instruction in the assembly program is associated with a unique constant identifier, which is added to the (initially empty) trace whenever this instruction is executed.

[2] Consistently with our discussion above, we consider that only Enc and Dec are executed in environments that are observable by the adversary.

Referring to Figure 2, we will consider adversaries attacking C implementations and assembly implementations, and receiving the leakage defined above as extra information. Later in the paper we will see formalizations of both types of leakage.

RSA-OAEP as described in PKCS#1 We now illustrate the level of abstraction at which cryptographic algorithms are described in standards. This will both justify our approach of taking C as a high-level language for implementing cryptographic algorithms, and also to facilitate the understanding of its EasyCrypt formalization in the next section. Figure 3 shows the PKCS#1 v2.1 standard's description of the encoding function that takes a variable-length message m, pads it to a fixed-length data blob DB, and produces the final encoded message EM. $\mathsf{OS2IP}_{\mathsf{PKCS}}$, $\mathsf{I2OSP}_{\mathsf{PKCS}}$ and $\mathsf{RSAEP}_{\mathsf{PKCS}}$ are specified in the standard; random and MGF instantiate the Rand, and RO oracles, respectively.

The scheme, as standardized, is parameterized by two lengths (which can be seen as the security parameter): k is the length of the canonical octet string representation of the RSA modulus; and $hLen$ is the length of the output of the hash function used to concretely implement the MGF, and to compute label hashes.

Given these parameters, some derived static lengths can be introduced: $dbLen = k - hLen - 1$ is the length of the padded message, or *data blob*; and $maxMLen = dbLen - hLen - 1$ is the maximum length of message that can be encrypted.

fun $\mathsf{OAEP_Encode}_{\mathsf{PKCS}}(m : \mathsf{octet}[])$:

$PS = \mathtt{0x00}^{(k-|m|-2*hLen-2)}$;
$DB = dLHash \| PS \| \mathtt{0x01} \| m$;
$seed = \mathsf{random}(hLen)$;
$dbMask = \mathsf{MGF}(seed, dbLen)$;
$maskedDB = DB \oplus dbMask$;
$seedMask = \mathsf{MGF}(maskedDB, hLen)$;
$maskedSeed = seed \oplus seedMask$;
$EM = \mathtt{0x00} \| maskedSeed \| maskedDB$;
return EM;

fun $\mathsf{OAEP_Encrypt}_{\mathsf{PKCS}}(m : \mathsf{octet}[], pk : \mathsf{pkey})$:

if $(maxMLen < mL)$
  $rc = RC_MessageTooLong$;
else {
  $rc = \mathsf{OAEP_Encode}_{\mathsf{PKCS}}(m)$;
  $p = \mathsf{OS2IP}_{\mathsf{PKCS}}(EM)$;
  $(rc, c) = \mathsf{RSAEP}_{\mathsf{PKCS}}(pk, p)$;
  $(rc, res) = \mathsf{I2OSP}_{\mathsf{PKCS}}(c, k)$;
  $rc = RC_Success$; }
return (rc, res);

Figure 3. Standard specification for PKCS#1 encryption

A* C*-like language In this paper we consider only C programs that follow a strong programming discipline and respect a strict notion of safety. The programming discipline forbids expressions with side-effects, pointer arithmetic, and only allows the programmer to refer to arrays using the address of their first element (and carrying an explicit offset value where necessary). Our notion of safety excludes the standard out-of-bound memory accesses and arithmetic error conditions, but also behaviours that would fall in the underspecified parts of the C standard. For simplicity, our notion of safety also imposes that C implementations terminate for all possible input values.

We allow static allocation of new arrays, and follow standard practice to pass output parameters by reference to let C functions return multiple values, or arrays. In addition, we consider that the language is equipped with a non-primitive type for arbitrary precision unsigned integers that are, similarly to arrays, passed by reference. We equip this type with the usual operations, including modular arithmetic.

We use the const type modifier to prevent functions from overwriting parameters passed by reference (arrays, big integers and pointers to primitive types) that are used purely for input. Purely for simplicity, we forbid the use of short-circuiting boolean operators, which introduce potentially unwanted conditional jumps, and prefer the corresponding bitwise operators.

In addition, as discussed above, we impose that the programmer correctly annotates the program with Leak(Event_True) and Leak(Event_False) trace extension statements. Correct annotation means that the next statement in each execution path after a conditional branch is an annotation exposing the corresponding value of the branching condition (cf. the code in the next section). Semantically, these statements append the corresponding event at the end of a global trace that is initially empty and can only be manipulated using the annotation mechanism.

3. Security of C-like code in EasyCrypt

We formalize security proofs using EasyCrypt [9], an SMT-based interactive prover geared towards proving security properties of cryptographic schemes. Cryptographic algorithms, oracles and games are described in a probabilistic imperative language pWhile. Reduction proofs can then be made by proving probabilistic relational properties of pairs of functions and computing concrete probability bounds.

Importantly, pWhile's grammar of expressions can be extended with user-defined types and functional (pure and total) operators. The language is equipped with some built-in types, including booleans, integers, tuples and *fixed-length* bitstrings. However, the latter type is insufficient to write, for example, algorithms whose input are bitstrings whose length is chosen by the adversary. As we define grammar extensions to model variable-length octet strings (in fact, polymorphic arrays) below, we write them in a style that narrows the gap between pWhile and the subset of trace-annotated C with big integers described above.

***A* C *mode for* EasyCrypt** We extend EasyCrypt with several libraries to lower the level of abstraction of specifications to one similar to our C-like language. The first of these libraries implements variable-sized arrays. It provides a polymorphic type $\alpha[]$ and select ($\cdot\,[\cdot]$) and update ($\cdot\,[\cdot] = \cdot$) operators, defined only when used within the array's bounds. We let the user declare statically-sized array variables (var $\cdot : \cdot[\cdot];$). For specification purposes, we also define a length operator ($|\cdot|$) and write valid(a, o, l) whenever o and l are non-negative integers and a is an array such that $o + l \leq |a|$. (This guarantees that any access to a with an index i such that $o \leq i < o + l$ is within the array bounds, and allows us to express memory-safety conditions.)

We also introduce a type bigint to distinguish variables meant to model machine integers from those meant to be implemented as arbitrary precision integers. The bigint type is equipped with operators for all operations needed, including comparisons, shifts and modular operations.

Our third library extension deals with parameter-passing: in order for our adversaries to have at least as much power over inputs and outputs as standard IND-CCA adversaries in the context of a C program, we model, in EasyCrypt, parameter passing by-reference, where a reference to an array or variable is passed as argument, and used to return multiple values by side-effect. When a function takes a parameter of an array or bigint type, we always assume that it may in fact overwrite that parameter's contents, unless otherwise specified using the const type modifier. (We syntactically ensure that all such parameters that are not used for output are marked as const.) Conversely, when a parameter of primitive type is meant to be used as an out-parameter, it should be marked as such using the out type modifier. Type α out is equipped with a dereference operator ($^*\cdot$), and an update statement ($^*\cdot = \cdot;$).

Finally, we define in EasyCrypt some constants and operators that capture the abstract C leakage described in Section 2. Abstract traces can be the empty trace Zero, the true and false branch events Event_True and Event_False, or any combinations of these using an associative extension operator $\cdot$ ++ $\cdot$ for which Zero is a left and right neutral element. In functions, oracles and games, we write Leak(e) to denote the fact that the observable trace is extended to the right (using ++) with e.

In addition to these basic events, we also consider abstract leakage from some of the external functions and big integer primitives. This allows us to make formal and precise, using axioms, the usual assumptions on their leakage. We use the $\rightsquigarrow t$ notation introduced in Section 2 to denote that a particular function call extends the execution trace to the right with a trace t that may depend on all of the call's arguments. (For algorithms, this trace extension is concrete, whereas it is kept abstract for chosen big integer primitives and for the external functions.)

Application to PKCS#1 v2.1 We discuss our implementation of the PKCS standard based on the encryption algorithm. Details of the other algorithms and subroutines can be found in Appendix C.

Figure 4 shows our implementation of the encryption algorithm. Auxiliary algorithms, as well as external functions, are written so that they return their results by passing a reference to some output parameters, but otherwise provide the same functionality. The encoding algorithm is optimized to save space by reusing its internal buffers $seed$ and DB once their contents become obsolete. Apart from the verbosity of using while loops to implement array operations, this implementation does not differ much from the encoding function described in the PKCS#1 standard (see Figure 3).

Between lines (3) and (8), the data blob is built, by copying the default label hash, writing the zero padding and the separator, and finally copying the message m into it. On line (9), the seed is sampled uniformly at random. On line (10), the MGF oracle is called and its result ($seedMask$) written directly into the output buffer, and used, in the while loop at (12) to mask the data blob in place. On line (13), the MGF oracle is queried with the masked data blob and its result written and used as before to mask the seed. Finally, on line (16), the first byte of the output buffer is set to 0.

Although it is a rather simple refinement of the specification shown in Figure 3, this implementation allows us to concretely reason about the leakage traces produced when executing the encryption algorithm. From a formal point of view, the first steps in our security and correctness proofs very much reduce the security of the low-level model to the security of the high-level scheme (augmented with concrete side-channel leakage).

However, even the high-level description displayed in Figure 3 is far removed from the description for which Fujisaki *et al.* obtained their well-known security proof [21]. We display it for comparison in Figure 5. There are three major differences between the two specifications:

- the standard uses only one random oracle, whereas Fujisaki *et al.* use two; for the lengths used in the standard, these are stricly equivalent, since F and G have disjoint domains.
- to remedy the fact that RSA is not a random permutation over the entire encoded message space, the standard ensures that it is always called on plaintexts whose first byte is zero; this has an incidence on the security and the proof.
- finally, the standard permits the encryption of variable (but bounded) length messages, using some additional padding to fixed length; this does not affect theoretical security, but has led to well-known padding oracle attacks [28, 34] when implemented carelessly.

```
fun OAEP_Encode(res : octet[], m : octet const[], mLen : int):

 1: var seed : octet[hLen];
 2: var DB : octet[dbLen];
 3: i = 0;
 4: while (i < hLen) { Leak(Event_True);
      DB[i] = dLHash[i];
      i = i + 1; } Leak(Event_False);
 5: while (i < dbLen − mLen − 1) { Leak(Event_True);
      DB[i] = 0x00;
      i = i + 1; } Leak(Event_False);
 6: DB[i] = 0x01;
 7: i = i + 1;
 8: while (i < dbLen) { Leak(Event_True);
      DB[i] = m[i − dbLen + mLen];
      i = i + 1; } Leak(Event_False);
 9: sample_octets(seed, 0, hLen);
10: MGF1(res, 1 + hLen, dbLen, seed, 0, hLen);
11: i = 0;
12: while (i < dbLen) { Leak(Event_True);
      res[1 + hLen + i] = res[1 + hLen + i] ⊕ DB[i];
      i = i + 1; } Leak(Event_False);
13: MGF1(res, 1, hLen, res, 1 + hLen, dbLen);
14: i = 0;
15: while (i < hLen) { Leak(Event_True);
      res[1 + i] = res[1 + i] ⊕ seed[i];
      i = i + 1; } Leak(Event_False);
16: res[0] = 0x00;
17: return RC_Success;

fun OAEP_Encrypt(res : octet[],
                 m : octet const[], mLen : int, pk : pkey):

 1: var p, c : bigint;
 2: var EM : octet[k];
 3: if (maxMLen < mLen) { Leak(Event_True);
      rc = RC_MessageTooLong; }
 4: else { Leak(Event_False);
      rc = OAEP_Encode(EM, m, mLen);
      rc = OS2IP(p, EM, k);
      rc = RSAEP(c, pk, p);
      rc = I2OSP(res, c, k);
      rc = RC_Success; }
    return rc;
```

Figure 4. Implementation of PKCS#1 v2.1 encryption

Despite these differences, the security proof is relatively easy to adapt and follows the same general schema. We perform it in a refinement of the adversary model presented in Section 2, which gives the adversary more control over output parameters. Oracle wrappers ensure that adversaries respect memory-safety side conditions when calling the algorithms and environment functions.

```
fun OAEP_Encrypt_proof(m : octet[], pk : pkey):

  r = random(hLen);
  s = (m‖0x00^hLen) ⊕ G(r);
  t = r ⊕ H(s);
  return RSA(pk, s‖t);
```

Figure 5. Fujisaki *et al.*'s specification for OAEP encryption

Security theorem Following Fujisaki *et al.*, we reduce the leakage-aware IND-CCA security of our PKCS#1 implementation to a variant of the set partial-domain one-way assumption (s-PDOW) on RSA. This variant takes into account the fact that the first byte of the random input is set to 0, and also lets the adversary observe the leakage produced when evaluating the one-way permutation and its inverse. More precisely, Figure 6 shows the assumption in game form, parameterized by a one-way permutation f, a leakage function $\mathsf{leak_f}$, and an adversary $\mathcal{B}$. The leakage function $\mathsf{leak_f}$ takes a secret key sk, and outputs a secret key sk_ℓ such that, for any challenge c, the computations of $\mathsf{f}^{-1}_{\mathsf{sk}}(c)$ and $\mathsf{f}^{-1}_{\mathsf{sk}_\ell}(c)$ produce the same execution trace. This key is used in the proof to simulate the trace produced by computing the decryption algorithm.

Game $s\text{-PDOW}_{\mathsf{f},\mathsf{leak_f},\mathcal{G},\mathcal{B}}(1^\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow_\$ \mathcal{G}$
$s \leftarrow_\$ \{0,1\}^{hLen}$
$t \leftarrow_\$ \{0,1\}^{dbLen}$
$c \leftarrow \mathsf{f_{pk}}(\mathtt{0x00}\|s\|t) \rightsquigarrow \tau$
$\mathsf{sk}_\ell \leftarrow \mathsf{leak_f}(\mathsf{sk})$
$T \leftarrow_\$ \mathcal{B}^{\mathsf{f}^{-1}_{\mathsf{sk}}(\cdot)}(c,\tau,\mathsf{sk}_\ell)$
return $t \in T$

Figure 6. Leakage-aware s-PDOW assumption

Given a one-way permutation f and a leakage function $\mathsf{leak_f}$ for it, we define, the s-PDOW advantage of an adversary $\mathcal{B}$ against f relative to $\mathcal{G}$ as

$$\mathsf{Adv}^{\mathsf{s\text{-}pdow}}_{\mathsf{f},\mathsf{leak_f},\mathcal{G},\mathcal{B}}(\lambda) := 2 \cdot \Pr\left[s\text{-PDOW}_{\mathsf{f},\mathsf{leak_f},\mathcal{G},\mathcal{B}}(\lambda) \Rightarrow \mathsf{T}\right] - 1\,.$$

The validity of assuming that this advantage is negligible depends greatly on the leakage function $\mathsf{leak_f}$ and the trace produced when evaluating the permutation f. In particular, when the leakage function is constant (that is, when the trace produced when evaluating the inverse permutation does not depend on the secret key), and evaluating the permutation does not leak information about the plaintext, our s-PDOW game only differs from the one used by Fujisaki *et al.* [21] by the fixed first byte of the challenge plaintext. This only introduces an additional factor 256 in the bound.

We can now state our security theorem for PKCS#1 v2.1 as we implement it.

Theorem 1 (Security of PKCS#1 v2.1)**.** *Let* Π *be our implementation of the standard. For all key generation algorithm* $\mathcal{G}$*, and* IND-CCA *adversary* $\mathcal{A}$ *making at most* q_{D} *queries to the decryption oracle and* q_{G} *queries to the random oracle with* $\ell_i = hLen$ *and* $\ell_o = dbLen$*, we build a* s-PDOW *adversary* $\mathcal{B}$ *against RSA such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\Pi,\mathcal{G},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{s\text{-}pdow}}_{\mathsf{RSA},\mathsf{leak_{RSA}},\mathcal{G},\mathcal{B}}(\lambda) + \frac{2q_{\mathsf{D}}q_{\mathsf{G}} + q_{\mathsf{G}} - q_{\mathsf{D}}}{2^{8hLen}}$$

The proof is formalized in EasyCrypt, and relies on several assumptions on the leakage produced by the library and environment functions. The assumption on modular exponentiation is formalized using the $\mathsf{leak_{RSA}}$ function. All other arithmetic functions are implicitly assumed to produce constant leakage traces. The random oracle and random sampling operations are assumed to produce leakage traces that depend only on their length parameters.

4. Security-aware compilation

Our goal is to take a C implementation such as that described in the previous section and compile it to an assembly implementation that retains the security properties that were established (or are simply assumed to hold) for the source code. A natural question that arises is then: what properties must the C compiler guarantee to ensure that the assembly implementation is secure based on the assumption that the C implementation is secure?

The classical notion of correctness for any compiler is known as *semantic preservation*. Intuitively, this property guarantees that, for any given source program S, the compiler will produce a compiled program T that operates *consistently* with the semantics of S.

Consistency is defined based on the *observable behaviour* of a program, which can be a simple relation on input states and output states, or it can be a more complex notion including observable events occurring during program evaluation. A bit more formally, let us denote the evaluation of a program P over inputs $\vec{p}$, resulting in outputs $\vec{o}$ and observable behaviour B as $P(\vec{p}) \Downarrow (\vec{o}, B)$. Then, semantic preservation could be written as

$$\forall B, \vec{p}, \vec{o},\ T(\vec{p}) \Downarrow (\vec{o}, B) \quad \Longrightarrow \quad S(\vec{p}) \Downarrow (\vec{o}, B)$$

This means essentially that any observable behaviour of the target program is observable in the source program. Furthermore, if the source language is deterministic (i.e., it has no intrinsic non-determinism and it interacts with a deterministic environment) then this also gives an implication in the reverse direction [26].

Here we introduce a notion of *security-aware semantic preservation* that refines the previous high-level notion. We also prove that, when enforced by a C compiler, this new notion is sufficient to guarantee that, not only the functionality of the source program is preserved, but also its security. We call C compilers that are proven to enforce this type of semantic preservation *security-aware*. In the next section we will show how we have extended CompCert to enable the security-aware compilation of C implementations.

Observable behaviour of an implementation The observable behaviours that we will consider for security-aware compilation will include, not only the leakage that is provided to the adversary, but also all the interactions of the implementation with the environment via Rand and RO. Formally, we see behaviours B as sequences of observable events ν, $B ::= \epsilon \mid \nu.B$, where we consider events of the following types:

$$\nu ::= \quad \mathsf{const}(id) \quad | \quad \mathsf{Rand}(v, \ell) \quad | \quad \mathsf{RO}(v, i, \ell_i, \ell_o)\,.$$

Intuitively, $\mathsf{const}(id)$ will correspond to a leakage event. In the case of C implementations, the identifier id will be either T or F, signalling the evaluation of a branching condition similarly to what was described in the previous section. In the case of assembly implementations, the identifier id will contain the unique identifier (PC address) of the instruction being executed, according to the program counter model. Rand events signal a call to an external random sampling function, including the length and output of the random octet string that was obtained from the environment. Similarly, RO events reveal the full details of an interaction with an external function representing the idealised MGF function.

We will refer to the projection of a behaviour that retains only the const events as $\mathsf{const}(B)$. Similarly, we will refer to the projection that excludes the const events as $\mathsf{coins}(B)$. We can now present our notion of security-aware semantic preservation.

Definition 4 (Security-aware semantic preservation)**.** *Take PKE implementation* $\Pi = (\mathsf{C}, \mathsf{Enc}, \mathsf{Dec}, \ell_{(\cdot)})$ *in* C. *We say that assembly implementation* $\pi = (\mathsf{asm}, \mathsf{Enc}', \mathsf{Dec}', \ell_{(\cdot)})$ *securely preserves the semantics of* Π *if there exists an efficient deterministic simulator* $\mathcal{S}$ *such that the following two conditions hold*

$$\begin{array}{l}\forall \lambda, B_\Pi, B_\pi, \mathsf{m}, \mathsf{pk}, \mathsf{c}, \mathsf{rc}.\\ \quad \mathsf{coins}(B_\Pi) = \mathsf{coins}(B_\pi) \wedge {\mathsf{Enc}'}_\lambda^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk}) \Downarrow (\mathsf{rc}, \mathsf{c}, B_\pi) \Rightarrow\\ \quad\quad \mathsf{Enc}_\lambda^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk}) \Downarrow (\mathsf{rc}, \mathsf{c}, B_\Pi) \wedge\\ \quad\quad \mathsf{const}(B_\pi) = \mathcal{S}(\mathsf{Enc}, \mathsf{Enc}', \mathsf{const}(B_\Pi))\,.\end{array}$$

$$\begin{array}{l}\forall \lambda, B_\Pi, B_\pi, \mathsf{c}, \mathsf{sk}, \mathsf{m}, \mathsf{rc}.\\ \quad \mathsf{coins}(B_\Pi) = \mathsf{coins}(B_\pi) \wedge {\mathsf{Dec}'}_\lambda^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk}) \Downarrow (\mathsf{rc}, \mathsf{m}, B_\pi) \Rightarrow\\ \quad\quad \mathsf{Dec}_\lambda^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk}) \Downarrow (\mathsf{rc}, \mathsf{m}, B_\Pi) \wedge\\ \quad\quad \mathsf{const}(B_\pi) = \mathcal{S}(\mathsf{Dec}, \mathsf{Dec}', \mathsf{const}(B_\Pi))\,.\end{array}$$

The intuition is the following. Consider behaviours B_Π and B_π where the randomness taken by the C implementation from the environment matches that taken by the assembly implementation. Then, for all possible parameter inputs, the assembly implementation must produce an output that is consistent with that of the C implementation. Furthermore, it must be possible to simulate the PC trace of the assembly implementation, given only the leakage of the C implementation.

The next theorem establishes that security-aware semantic preservation implies that PKE correctness is preserved in compilation. The proof is a direct reduction and is presented in Appendix A.

Theorem 2. *Take PKE implementations* $\Pi = (\mathsf{C}, \mathsf{Enc}, \mathsf{Dec}, \ell_{(\cdot)})$ *and* $\pi = (\mathsf{asm}, \mathsf{Enc}', \mathsf{Dec}', \ell_{(\cdot)})$. *Suppose that* Π *is a correct PKE implementation. Then, if* π *securely preserves the semantics of* Π, *it is also correct.*

The next theorem shows that security-aware semantic preservation guarantees that security is preserved by compilation, i.e., that the assembly implementation will be secure against adversaries that get program counter leakage, assuming that the C implementation is secure in the leakage model described in the previous section.

Theorem 3. *Take PKE implementations* $\Pi = (\mathsf{C}, \mathsf{Enc}, \mathsf{Dec}, \ell_{(\cdot)})$ *and* $\pi = (\mathsf{asm}, \mathsf{Enc}', \mathsf{Dec}', \ell_{(\cdot)})$. *Suppose that* Π *is an* IND-CCA *secure PKE implementation. Then, if* π *securely preserves the semantics of* Π, *it also is* IND-CCA *secure* .

The proof is presented in Appendix B, and it hinges on the following observation on security-aware semantic preservation. Conceptually, what we are doing in Definition 4 when we quantify over behaviours B_Π and B_π is to quantify over all random coins taken by the implementations, and to ensure that the same coins are provided to both Π and π. This can also be seen as quantifying over a set of deterministic environments, each of them providing a possible value of the random coins. Interestingly, in this case, and given that our source implementation language is a deterministic subset of C, we get that security-aware semantic preservation also gives an implication in the reverse direction [26]. In other words, in addition to the implication shown in Definition 4, we also get the following: for each deterministic environment and for each input $\vec{p}$, the source program has a single observable behaviour which maps to the single observable behaviour in the compiled assembly code (which is also deterministic). In a nutshell, this means that one can directly reduce the security of the assembly implementation to the security of the C implementation, provided that leakage can be simulated.

5. Rendering CompCert security-aware

In this section we show how we extended the CompCert certified compiler [26] and used it to perform this type of security aware compilation.

***Background on* CompCert.** CompCert is a formally verified optimizing C compiler [26]. It produces target code with strong correctness guarantees and reasonable efficiency when compared to general purpose compilers. CompCert supports the C language (with almost complete coverage of the ISO C 90 / ANSI C standard) and produces assembly code for the PowerPC, ARM, and IA32 (x86 32-bits) architectures. CompCert is mostly implemented in Coq, and its development is subdivided into 19 compiler phases, each of which builds a semantic preservation proof between semantically defined intermediate languages.

Formally, CompCert's correctness theorem establishes the strong notion of semantic preservation that was introduced in the beginning of Section 4, referred in CompCert terminology as a *backward simulation*. This guarantees that, if a source program P^{C} is successfully compiled into P^{asm}, then the observable behaviour of this last program is an admissible behaviour of the original program. The proof of this result is based on a formalization of the semantics of both the compiler's source and target languages (C

and assembly), as well as of all the compiler passes. Behaviours are captured by a possibly infinite sequence of events that model interactions of the program with the outside world, such as accesses to volatile variables, calls to system libraries, or user defined events (so called annotations).

The need for* CompCert *extensions. There are various aspects in which we needed to enhance both the functionality of CompCert and the formal correctness guarantees that it provides in order to guarantee security-aware semantic preservation. In the remainder of this section we will begin by identifying precisely what CompCert does and does not provide in this direction, and then explain in detail how we have implemented the necessary extensions.

CompCert's semantic preservation result establishes guarantees very close to Definition 4: conditioning on similar interactions with the environment, the observable behaviour for the compiled assembly program matches the observable behaviour of the source C program. However, the following caveats need to be addressed before applying the results of Section 4:

- **Expressiveness of CompCert behaviours.** The notion of observable behaviour of a C program and an assembly program in CompCert is conceptually more general than the one we adopted in the previous section. It considers, for example, the possibility that programs go wrong or do not terminate. However, it is more restrictive in the sense that, on one hand it considers only programs with a well-defined entry point (the main function) and does not include support for the Rand and RO events that we require (as these imply exposing or updating the values of memory regions).
- **Absence of a leakage model.** CompCert behaviours do not have an associated intrinsic notion of side-channel leakage (neither at the C nor at the assembly levels). Although it is possible to emulate such leakage using annotations directly placed over the source code, there is no way to guarantee that the target code would have an observable behaviour that follows the instrumented semantics conventions of the PC model that we have described in the previous sections. This means that there is no straightforward way to capture the simulation of PC traces that we require for security-aware compilation.
- **Complex data types.** The common practice in the implementation of cryptographic software is to split the development in two parts: i. a trusted number theory library that extends the high-level language of choice with the complex date types required for public-key cryptography; and ii. code that implements a specific scheme by relying on the functionality of the trusted library. As described in Section 3, we also adopt this approach at the EasyCrypt level, by writing C code that relies on operators that carry out multi-precision integer calculations. A set of axioms describes important properties for these operators, similarly to what happens to other C native data types such as octet strings. Conceptually, we see this as an extension to C that adds support for an additional data type, which means that the semantics of C as formalized in CompCert need to be extended for our purposes.

In the remainder of this section we will explain how we have extended CompCert to eliminate the previous caveats.

Adding support for additional external operations A difficulty that arises when we try to base our results on CompCert's semantic preservation guarantee is how to handle the environment that was setup in the EasyCrypt formalization. This includes primitives for sampling values, the choice of appropriate hash functions to instantiate the oracles, and other support functionality. This is an important issue, since we do not want to impose an a-priori commitment on these choices — this would, not only weaken (specialize) the guarantees offered by the implementation, but also it would not match what is the common practice in the implementation of cryptographic software. The semantics formalized in CompCert support calls to system libraries, but these calls are assumed to leave the memory state unaltered. This makes them inappropriate to model the environment we require. Consequently, our first extension to CompCert is a mechanism for declaring external functions that may impact the memory.

Our solution is an extension of the treatment adopted by CompCert for system-calls. A call to a system library triggers an observable event that registers the name of the called function, its arguments, and an additional value that represents the result (provided by the execution environment). In order to address functions that change the memory state, we have introduced a new event whose arguments are byte-arrays containing data read-from/written-to specified memory regions. For that, we have implemented a simple mechanism to specify the memory footprint of an external function. We allow the arguments of each function to be either a primitive C type or a pointer to a memory region that is either read or written. The size of regions is declared by an expression that is allowed to refer to other (int) arguments. As an example, consider the declaration of the sample_octets and random_oracle functions that instantiate the Rand and RO oracles in our PKCS implementation:

```
#pragma libspec sample_octets : OutPtr #1, Int -> Void
extern void sample_octets(unsigned char *,size_t);
#pragma libspec random_oracle :
                 OutPtr #1, Int, InPtr #3, Int -> Void
extern void random_oracle(unsigned char *,
                 size_t, unsigned char *, size_t);
```

In the first line, the pragma directive informs CompCert that the first argument is a pointer to a memory region written by the function. The size of the region is given by the integer passed as second argument. This mechanism is implemented in a general way that allows specifying arbitrary functions that feed output memory regions to the environment and fill input memory regions with data collected from the environment. More precisely, the semantics of a function declared using this mechanism is defined to:

1. read the memory regions and other input arguments;
2. ask the environment for a byte-array with size equal to the sum of the output sizes;
3. split the obtained array in the required pieces, and write them in the corresponding memory regions;
4. produce an event with arguments as described above.

By extending the correctness result of CompCert to cover these external functions, it follows that the observable behaviour of a source program will be preserved in the assembly code, for any instantiation of these functions. However, in the context of security-aware compilation, one must recall that these functions are actually idealised constructions that are used in the security proofs. It is critical to the security of the final implementation that the instantiation of these functions complies as much as possible with the assumptions that are made explicit in the EasyCrypt proof. In other words, these functions must be *trustworthy* in order to obtain assembly-level security, and this is why we named this mechanism *TrustedLib*.

In the particular case of the RSA-OAEP implementation the above discussion implies that, in line with standard cryptographic practice, it is up to the end-user to instantiate sample_octets with a function that provides good quality randomness, and to follow the recommentations in the PKCS standard for the instantiation of the mask generation function that we model as a random oracle. Furthermore, the assumptions regarding the leakage of these external functions that are axiomatized in EasyCrypt should also be satisfied by the instantiation.

Correctness of the cryptographic API. In CompCert, the meaning of a C program is the behaviour of executing a specific entry-

point: the main function. We are instead interested in the fact that a given API is correctly implemented.[3] We have therefore anchored our correctness result on a program entry-point with the following shape:

Function main( )
$id \leftarrow$ read( )
$\vec{p} \leftarrow$ read( )
$o \leftarrow id(\vec{p})$
write(o)

This generic entry point reads the identifier of the function to evaluate (in the case of a PKE implementation this means either Enc or Dec). It then uses external functions read and write to obtain input values for the function from the environment, and to externalise the corresponding outputs.[4] This pattern registers every possible input/output behaviour of each function as an admissible behaviour of the program. Hence, the correctness result of CompCert ensures preservation of this behaviour along the compilation process.

The main function is added to the cryptographic implementation to ensure that the translation to assembly is explicitly captured by the correctness theorem of CompCert. This function can be discarded from the assembly code, since the implementation is itself intended to be used as a library by some higher level application.

Adding support for a big-number library The implementation of public-key cryptography code often requires the use of big-number libraries to carry out complex algebraic computations such as modular exponentiation. For example, a typical implementation of RSA-OAEP will delegate the big-number computations to a (often pre-compiled) library such as GMP.[5] From the developers' point of view, this can be seen as extending C with a new data type and native operations that provide support for big integers.

One possible approach to handle such external libraries would be to use the *TrustedLib* mechanism to equip our framework with external functions, which would leave it to the environment to provide the big-integer operators.[6] However, this would not match the setting we have captured in the EasyCrypt formalization, in which we consider a well-defined intended semantics for these operations Furthermore, the correctness and security of the C implementation depend crucially on the correct implementation of these functions (otherwise, the axioms assumed in the EasyCrypt theory might not be validated). For this reason, we have opted to include the necessary big-integer operations as new built-in operations in CompCert, with a fully defined semantics, which we see as a contract on the library that we will use for linking with our program. Once again we must trust the library that instantiates these operations to correctly implement those contracts, which would ideally be addressed through formal verification. Unfortunately, the complexity of state-of-art libraries such as GMP make such an enterprise an enormous effort (although results obtained on smaller libraries [32] and in specific algorithms [12] provide confidence on feasibility in a near future).

Technically, the formalization of the semantics of the big-number builtin operations is a refinement of that presented for the *TrustedLib* functions, in which the transformations on the state are fully specified, and where the events signalling communication with the environment are ommitted. The transformations on the state are described in three steps: i. reading the big-integer word representations of all the inputs and converting them into integers (using the SSReflect library [23]); ii. performing the operations over the integers; and iii. converting back into the memory representation and storing the results. A consequence of our approach is that we needed to commit to a concrete representation of big-integers. We use a 2^{30} radix representation, stored as an array of 32-bit machine integers. This means that we use 30-bits in each machine word for storage, where the remaining two bits should always be kept at zero at the function boundaries (functions may internally and temporarily cause these bits to be non-zero). Our formalization is flexible enough to enable straightforward adaptation to other representations. The formalization of the big-number library as built-in operations in CompCert has the additional advantage of automatically extending the CompCert interpreter to support these operations, which may be useful for debugging.

Our framework includes an instantiation of the big-number library that was developed for illustrative purposes, adapted from the *Long Integer Package (LIP)* library by Arjen Lenstra.[7] One of our concerns was to ensure that all the big-integer operations comply with the leakage requirements imposed by the EasyCrypt proof: the leakage should be constant and independent of the concrete inputs passed to the functions. To this end, we have incorporated a standard countermeasure against side-channel attacks, and simplified the library to consider only unsigned integers and use static memory allocation. All the routines have been modified so that there is no data-dependent branching and memory indexing as explained, for example, in [11]. This means that all functions in the library execute in constant time and access the same memory addresses, regardless of the input data. Integers are stored in statically declared arrays of pre-defined size, which means that the maximum range of the integers that will be manipulated by the program must be known in advance, and must be provided at the time of compilation. For our example, we have aimed at 4096-bit RSA keys, represented using 137 words. We note that fixing the maximum length of integers is consistent with our assumption that the security parameter is known at compile time, and the formal semantics that we have added in CompCert captures exactly the behaviour of our library. Although the functionality of our library is not formally verified, we have used a Frama-C plug-in [8] to check that the side-channel countermeasures are correctly deployed.

Dealing with side-channel leakage The last extension to CompCert concerns the need to satisfy the part of Definition 4 imposing that traces generated by the assembly code in the PC model can be simulated from the leakage model we adopted for C programs.

The annotation mechanism of CompCert allows us to extend standard C code with dummy statements that, when evaluated, give rise to events that externalize arbitrary constant identifiers making them visible in the observable behaviour of the program. This means that a direct translation of the EasyCrypt code into CompCert taking advantage of these annotations gives rise to a C program whose observable behaviour includes (among the other events that may be signalled) a trace of all the conditional jumps taken by the program. Furthermore, CompCert guarantees that the same exact traces will be observable in the assembly program.

It is obvious that, by inspecting the sequence of events that reports which conditional jumps were taken, one can reconstruct the entire control flow of the C program. However, the same cannot be said for the control flow of the assembly program: indeed, CompCert is only guaranteed to preserve the observable behaviour of the C program, and many possible assembly programs can achieve

[3] This is rather a matter of presentation, since in order to establish the result on the main function, CompCert needs to establish equivalent results for the remaining functions of the program.

[4] In the case of our example, the inputs and outputs are simply octet strings of various lengths, which means that the read and write functions can be also implemented using the external function mechanism described above.

[5] http://gmplib.org

[6] In fact, we have used that mechanism to specify all the GMP low-level integer API.

[7] http://www.win.tue.nl/~klenstra/

this. In particular, there may be assembly programs which are insecure in the PC model (i.e. that leak sensitive information via the control flow) and still have an observable behaviour that matches that of the original C program. In other words, it may not be possible to fully determine the sequence of program counter values taken by an assembly implementation solely from the observable conditional jumps taken by the C program. However, in order to achieve security-aware compilation, this is precisely what we require.

Instead of proving that each of the compiler passes does not introduce spurious branching, we have implemented a simple static analysis on the generated assembly program that establishes the desired property. Our analysis is formalized in Coq as a translation validation that checks, for every conditional branch instruction in the assembly code, that i. all execution paths arising from that instruction go through an annotation; and ii. that these annotations give rise to events that externalize (pairwise) distinct constant identifiers. This is sufficient to ensure that the observable behaviour of the program fully reflects the choice of the execution path, and we have formalized and proved in Coq the following theorem that establishes the soundness of the translation validation.

Theorem 4 (PC trace simulatability)**.** *Let S be an assembly program that passes the translation validation described above. Let also s_1 and s_2 be memory states s.t. $s_1 \equiv^{\mathsf{PC}} s_2$; and, let B_1 and B_2 be behaviours s.t. $B_1 \sim^{\mathsf{ann}} B_2$, $\langle S; s_1\rangle \Downarrow B_1$, and $\langle S; s_2\rangle \Downarrow B_2$. Then, for any states s_1', s_2' and traces t_1, t_2 we have that*

$$s_1 \xrightarrow{t_1} s_1' \wedge s_2 \xrightarrow{t_2} s_2' \quad \Longrightarrow \quad s_1' \equiv^{\mathsf{PC}} s_2' .$$

Here, states s_1, s_2 are PC-equivalent ($s_1 \equiv^{\mathsf{PC}} s_2$) when they agree on the value stored in the PC register and have the same call stack, and behaviours B_1, B_2 are annotation-equivalent ($B_1 \sim^{\mathsf{ann}} B_2$) when they exhibit the same (possibly infinite) sequence of annotation events. The above theorem shows that the sequence of PC values in the evaluation of an assembly program that passes the translation validation is fully determined by the sequence of constant identifiers revealed via annotations in the observable behaviour. More precisely, the theorem expresses this result in the style of a non-interference result: if a program is validated by the test, then any two instances of it that exhibit the same annotations on their behaviour, are guaranteed to proceed in lockstep, i.e., the next PC value can always be determined from the observable trace.

The above theorem treats the execution of external calls and other compiler builtins as atomic steps. The soundness of the validation depends on the assumption that these external functions have precisely the same property (a compiler warning collects the identifiers of all these functions to remind the user of this fact). From the perspective of the end-user, the test is triggered by a new command-line option -max-annot. When the validation fails, no executable is produced and an error is emitted pointing to the branch of the (bad) assembly program that fails the check.

The following corollary relates the theorem above to the notion of semantic preservation in Definition 4.

Corollary 5 (Informal)**.** *Consider a PKE implementation that, when it is compiled with CompCert, gives rise to an assembly program that passes the translation validation check. Then, the compilation performed by CompCert enforces Definition 4.*

The proof of this corollary follows directly from the fact that the correctness theorem for CompCert guarantees that Definition 4 is satisfied, provided that simulator $\mathcal{S}$ can be constructed. The theorem above guarantees the correctness of the trivial simulator that looks ahead to the potential executions of the assembly program, until it finds the annotation that reveals the correct execution paths.

Using* CompCert *for security-aware compilation To summarize the above discussions, we have extended CompCert with a number of features and adapted the correctness result of the compiler to accommodate these extensions. This means that CompCert will preserve the observable behaviours of source C programs that rely on an arbitrary *TrustedLib*. We have also shown that the translation validation step that we have added to the compiler guarantees that simulation of PC traces is possible for accepted assembly programs. Putting these two results together, we conclude that our version of CompCert provides security-aware compilation by guaranteeing semantic preservation according to Definition 4. This means that, by the Theorems proved in Section 4, compiling a cryptographic implementation from C code to assembly, one obtains the following guarantees:

1. Assuming that the C implementation is secure in a side-channel aware security model such as the one described in Section 2;
2. Compiling the C implementation with the generic main entry point using CompCert and activating the translation validation stage;
3. Assuming that the *TrustedLib* functions are instantiated with a secure and correct library that satisfies the requirements specified in the security proof;
4. Then, if compilation does not fail, the assembly implementation is correct and secure against real-world adversaries that attack the scheme in the PC model.

Experimental results We have performed an evaluation of the performance of the assembly code produced by CompCert when used for security-aware compilation. Our goals were three-fold: i. to evaluate whether or not the translation validation check might reject the assembly produced by CompCert (in which case the compiler might not be preserving the side-channel countermeasures); ii. to evaluate whether the annotation of C source-code with the leakage tags might damage the performance of the code produced by CompCert; and iii. to compare the efficiency of the secure code produced by CompCert when compared to GCC. We have conducted our evaluation in a standard PC with an IA32 architecture. In addition to the PKCS implementation described here, we have also evaluated the entire NaCl library core [11].

Our findings were the following. We have not encountered any example where the assembly code generated from properly annotated and secure C code was rejected due to transformations performed by CompCert. On the other hand, there were several cases where the transformation validation stage led us to identify points in the C code where there might be potential leakage problems (mostly associated with the compilation of composed Boolean expressions). Furthermore, when comparing the performance of validated assembly code with that produced by CompCert from non-annotated C code, there were no significant deviations in performance. These findings indicate that CompCert behaves well in the preservation of the class of side-channel countermeasures that we considered in this paper, when they are deployed at the C level.

Regarding the overall performance of generated code, Figure 7 shows some selected benchmarking results, normalized with respect to the performance of non-optimized GCC output. Our findings are consistent with the known reports on CompCert benchmarking when comparing CompCert with unoptimized GCC code: the former outperforms the latter by roughly a factor of 2. When comparing with GCC at optimization level 1, we have found that CompCert is at least 30% slower. These results are slightly worse than previously reported values [27], which put this value at roughly 15%. We attribute this discrepancy to the domain-specific nature of our code, namely to side-channel countermeasures and intensive use of arithmetic and bit-wise operations. In the case of PKCS, we have also considered the case where the trusted library is pre-compiled and linked with the outputs of CompCert and GCC. In this case, CompCert performs as well as GCC at optimization

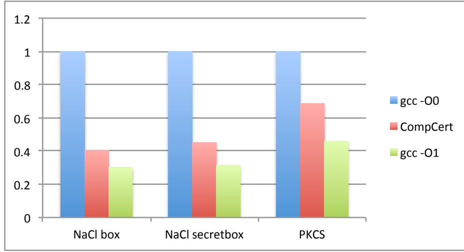

Figure 7. Benchmarking results

level 1, which shows that most of the speedups of GCC are being achieved in the optimization of the trusted library.

6. Concluding remarks

We have developed library extensions to EasyCrypt that enable the development of cryptographic security proofs directly on a large subset of the C language, in an extended security model where the adversary is given access to execution traces modelling PC security. We have extended the CompCert certified compiler with a mechanism for reasoning about programs relying on trusted libraries, as well as a translation validation stage based on CompCert's annotation mechanism. We have shown that these mechanisms along with a trusted library providing arithmetic operations and instantiations of idealised operations are enough to preserve correctness and PC security guarantees from a source C program down to its compiled assembly executable. We have also shown the independent value of the new CompCert extensions for compiling third-party C programs whilst preserving their claimed PC security properties.

Related work Our work lies at the intersection of computer-aided cryptography and certified compilation; we refer the readers to [13] and [26] for recent accounts of these fields, and focus this related work section on the verification of cryptographic implementations, and the formal treatment of side-channels.

Machine-checked correctness proofs of implementations of cryptographic primitives have been well-studied, using techniques ranging from equivalence checking [33], to verifying compilation [30], to deductive program verification [5] and interactive theorem proving [1]. However, these techniques are focused on functional correctness and do not attempt to formally carry provable security guarantees to the implementations.

Some proposals have been made towards obtaining computational security guarantees of implementations of cryptographic primitives and protocols. These are based on deductive verification [18], code generation [16], model extraction [4], refinement type systems [20], or static information-flow analysis [25]. However, these techniques focus on source program verification and do not explicitly address executable code, nor side-channel attacks.

In addition to being a core area of research in practical cryptography, side-channel attacks and countermeasures have been studied extensively in related areas, namely in the setting of programming languages [2, 3, 24, 35] and of theoretical cryptography [6, 19]. These works provide a more general account of side-channel attacks, either by considering a more precise computational model, e.g. with caches, or by providing a more abstract treatment of side-channels. However, they typically reason in a single setting—source code, assembly code, or an abstract model of computation. In contrast, we precisely relate the leakage properties of primitives to the security of algorithms and their executable implementations.

Our work is also related to ongoing efforts to formalize programming languages and compilers. In particular, the idea of harnessing a general-purpose verification tool with the CompCert compiler appears in the Verified Software Toolchain [7], and extensions of CompCert with arithmetic libraries are considered in [14].

Directions for further work We intend to leverage the developments of this paper to build a verified software toolchain for cryptographic implementations.

A first step towards this goal is to provide automated support for the C mode of EasyCrypt; we are confident that the additional complexity introduced by low-level considerations can be managed automatically to a large extent. Automation is also instrumental for the feasibility of security proofs in alternative leakage models. For example, we would like to investigate stronger leakage models in which the adversary could observe the list or set of memory addresses accessed during the execution of an algorithm, as well as weaker leakage models in which the adversary could observe the number of operations performed during the execution of an algorithm. Dealing with these alternative models would also require extending EasyCrypt with further libraries and to extend the new translation validation stage in CompCert to guarantee the preservation of countermeasures adequate for these leakage models.

Finally, we did not tackle the correctness properties of the big-integer trusted library that we use to extend C. We leave it as an interesting challenge for future work to evaluate the impact of side-channel countermeasures on the feasibility of formally verifying the correctness of such a multi-precision arithmetic library.

Acknowledgements

The fourth author is supported by an Amarout II grant (FP7 Marie Curie Actions-COFUND 291803). This work is part-financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ENIAC/2224/2009 and by ENIAC Joint Undertaking under grant agreement number 120224.

References

[1] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Sci. Comput. Program.*, 77(10-11):1058–1074, 2012.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of POPL'00*, pages 40–53, 2000.

[3] Johan Agat and David Sands. On confidentiality and algorithms. In *IEEE Symposium on Security and Privacy*, pages 64–77. IEEE Computer Society, 2001.

[4] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *ACM Conference on Computer and Communications Security*, pages 712–723. ACM, 2012.

[5] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. *Innovations in Systems and Software Engineering*, 6(3):203–218, 2010.

[6] Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Survey: Leakage resilience and the bounded retrieval model. In Kaoru Kurosawa, editor, *ICITS*, volume 5973 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[7] Andrew W. Appel. Verified software toolchain - (invited talk). In *ESOP'11*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[8] Manuel Barbosa, editor. *Deliverable 5.4: Certified shared library core*. Computer Aided Cryptography Engineering (CACE FP7 EU Project), 2011. http://www.cace-project.eu.

[9] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Heidelberg, 2011. Springer.

[10] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella-Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196, Heidelberg, 2011. Springer.

[11] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin Heidelberg, 2012.

[12] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[13] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2012.

[14] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *Arith - 21st IEEE Symposium on Computer Arithmetic*, pages 107–115. IEEE, 2013.

[15] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2012.

[16] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally secure protocol specifications. In *POST*, volume 7796 of *Lecture Notes in Computer Science*, pages 63–82. Springer, 2013.

[17] Jean Paul Degabriele, Kenneth Paterson, and Gaven Watson. Provable security in the real world. *Security Privacy, IEEE*, 9(3):33–41, may-june 2011.

[18] François Dupressoir, Cédric Fournet, and Andrew D. Gordon. Proving computational security with a general-purpose C verifier. In submission.

[19] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 293–302, Washington, 2008. IEEE Computer Society.

[20] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *ACM Conference on Computer and Communications Security*, pages 341–350. ACM, 2011.

[21] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2001.

[22] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[23] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[24] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-Channels. In *Proc. 24th International Conference on Computer Aided Verification (CAV '12)*, pages 564–580. Springer, 2012.

[25] Ralf Küsters, Tomasz Truderung, and Juergen Graf. A framework for the cryptographic verification of Java-like programs. In *CSF*, pages 198–212. IEEE, 2012.

[26] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54, New York, 2006. ACM.

[27] Xavier Leroy, editor. *The CompCert C verified compiler: Documentation and user's manual*. INRIA Paris-Rocquencourt, 2013.

[28] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238, Heidelberg, 2001. Springer.

[29] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.

[30] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *ACL2*, pages 1–10. ACM, 2006.

[31] Phillip Rogaway. Practice-oriented provable security and the social construction of cryptography. Unpublished essay, 2009.

[32] Sabine (formerly Fischer) Schmaltz. Formal verification of a big integer library including division. Master's thesis, Saarland University, 2007.

[33] Eric Whitman Smith and David L. Dill. Automatic formal verification of block cipher implementations. In *FMCAD*, pages 1–7. IEEE, 2008.

[34] Falko Strenzke. Manger's attack revisited. In Miguel Soriano, Sihan Qing, and Javier López, editors, *Information and Communications Security*, volume 6476 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2010.

[35] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 99–110. ACM, 2012.

A. Proof of Theorem 2

Proof. The proof is a direct reduction, where we show that any adversary $\mathcal{A}$ that contradicts the correctness of the assembly implementation would imply an adversary $\mathcal{A}'$ that contradicts the correctness of Π. Let us take $\mathcal{A}'$ to be $\mathcal{A}$ itself.

By the correctness of Π, we know that for all possible interactions with Rand and RO, all honestly generated pk and sk, and all valid m that may be output by $\mathcal{A}'$ we have that encryption is always successful and that, for some ciphertext c

$$(\mathsf{T}, \mathsf{c}) = \mathsf{Enc}^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk}) \quad \wedge \quad (\mathsf{T}, \mathsf{m}) = \mathsf{Dec}^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk})$$

Now suppose that $\mathcal{A}$ succeeds in the correctness game for the assembly implementation. Then it must be the case that, for some interaction with Rand and RO, honestly generated pk and sk, and some valid m output by $\mathcal{A}$, we would have that encryption fails or that, for some ciphertext c

$$(\mathsf{T}, \mathsf{c}) = \mathsf{Enc}^{\mathsf{Rand},\mathsf{RO}}(\mathsf{m}, \mathsf{pk}) \quad \wedge \quad (\mathsf{T}, \mathsf{m}) \neq \mathsf{Dec}^{\mathsf{RO}}(\mathsf{c}, \mathsf{sk})$$

The semantic preservation of π tells us that, conditioning on each possible interaction with Rand and RO, and on any inputs, whatever output is observed by $\mathcal{A}$ in Enc' and Dec' would also be observable in the C implementations. However, this would contradict the correctness assumption on Π, which concludes the proof. $\square$

B. Proof of Theorem 3

Proof. The proof is a direct reduction, where we show that any adversary $\mathcal{A}$ that contradicts the security of the assembly implementation would imply an adversary $\mathcal{A}'$ that contradicts the security of the C implementation. We present algorithm $\mathcal{A}'$ in Figure 8. For simplicity we omit the simulation of the random oracle, in which $\mathcal{A}'$ simply relays the adversaries queries to its own oracle. This means that the proof actually holds in the standard model.

The intuition is the following: algorithm $\mathcal{A}'$ uses simulator $\mathcal{S}$ to translate the leakage obtained at the C level to that expected by $\mathcal{A}$ at the assembly level. Apart from that, $\mathcal{A}$ simply passes the outputs of $\mathcal{A}'$ to its own environment, and returns the results without change. Note that this is possible because the semantic domains of both C and assembly programs is the same.

Adversary $\mathcal{A}'_1(\mathsf{pk})$:

$(\mathsf{m}_0, \mathsf{m}_1, \mathsf{st}) \leftarrow_\$ \mathcal{A}_1^{\mathsf{Decrypt}(\cdot)}(\mathsf{pk})$
Return $(\mathsf{m}_0, \mathsf{m}_1, \mathsf{st})$

Adversary $\mathcal{A}'_2(\mathsf{c}^*, t_\Pi, \mathsf{st})$:

$t_\pi = \mathcal{S}(\mathsf{Enc}, \mathsf{Enc}', t_\Pi)$
$b' \leftarrow_\$ \mathcal{A}_2^{\mathsf{Decrypt}(\cdot)}(\mathsf{c}^*, t_\pi, \mathsf{st})$
Return b'

oracle $\mathsf{Decrypt}(\mathsf{c})$:

Call $\mathsf{IND\text{-}CCA}_{\Pi,\mathcal{A}}.\mathsf{Decrypt}(\mathsf{c})$ to get $(\mathsf{rc}, \mathsf{m}, t_\Pi)$
$t_\pi = \mathcal{S}(\mathsf{Dec}, \mathsf{Dec}', t_\Pi)$
Return $(\mathsf{rc}, \mathsf{m}, t_\pi)$

Figure 8. An attacker against the C implementation

Our first observation is that the input to $\mathcal{A}_1$ is identically distributed to that in the real attack game against the assembly implementation. We aim to show that the simulation that $\mathcal{A}'$ offers of the decryption oracle and the challenge ciphertext are perfect, so that $\mathcal{A}$'s advantage is directly translated into an attack on the specification. We begin with the decryption oracle. We show that, for any ciphertext produced by the adversary in the attack against the assembly implementation, $\mathcal{A}'$ responds with the correct reply. Consider any given query c placed by the adversary. We know $\mathcal{A}$ is expecting the answer to be computed as

$$(\mathsf{rc}, \mathsf{m}) \leftarrow \mathsf{Dec}'^{\mathsf{RO}}_\lambda(\mathsf{c}, \mathsf{sk}) \rightsquigarrow t_\pi$$

Now, the fact that π preserves the semantics of Π means that, conditioning on any interaction with the environment, the C implementation will produce exactly the same result $(\mathsf{rc}, \mathsf{m})$. Since we know that $\mathcal{A}'$ computes its answer by using its own oracle (note that the legitimacy of $\mathcal{A}$ implies that $\mathcal{A}'$ is also legitimate in its decryption oracle queries) it follows that $(\mathsf{rc}, \mathsf{m})$ are correctly simulated. Furthermore, we also see that $\mathcal{A}'$ can provide $\mathcal{S}$ with all the inputs it requires to simulate the correct PC trace for $\mathcal{A}$, and so the simulation of the decryption oracle is perfect.

At this point we know that the messages m_0 and m_1 produced by $\mathcal{A}$ are identically distributed to what would occur in the real attack game against π. In particular, these messages can be output by $\mathcal{A}'$ as legal outputs in its attack against Π.

To complete the proof, it suffices to show that the challenge ciphertext is correctly simulated by $\mathcal{A}'$. This follows from a similar argument to that used for the simulation of the decryption oracle. We know $\mathcal{A}$ expects the challenge ciphertext to be computed as

$$\mathsf{c}^* \leftarrow_\$ \mathsf{Enc}'^{\mathsf{Rand},\mathsf{RO}}_\lambda(\mathsf{m}_b, \mathsf{pk}) \rightsquigarrow t_\pi$$

Again, the semantic preservation property gives us that the output produced at the C level for the same inputs is distributed identically to what $\mathcal{A}$ is expecting and, furthermore, that $\mathcal{S}$ provides the correct PC trace. Now, since the view of $\mathcal{A}'$ is perfectly simulated by $\mathcal{A}$, it follows that $\mathcal{A}$ will succeed in breaking Π with the same advantage as $\mathcal{A}'$. This concludes the proof. ☐

C. PKCS#1 v2.1 implementation

```
fun OS2IP(res : bigint, x : octet const[], len : int):
  var rc : rcode;  var aux : bigint;  var counter : int;
  octet_to_bigint(res, x [0]);  counter = 1;
  while (counter ≤ len + 1) { Leak(Event_True);
    bigint_lshift_8(res, res);
    octet_to_bigint(aux, x [counter]);
    bigint_add(res, res, aux);
    counter = counter + 1;  } Leak(Event_False);
  return RC_Success;

fun I2OSP(res : octet[], x : bigint const, l : int):
  var rc : rcode;  var aux, x' : bigint;  var counter : int;
  bigint_copy(x', x);  counter = 1;
  while (counter ≤ l) { Leak(Event_True);
    res[l − counter] = bigint_to_octet(res);
    bigint_rshift_8(x', x');
    counter = counter + 1;  } Leak(Event_False);
  bigint_zero(aux);
  if (bigint_lt(aux, x')) { Leak(Event_True);
    rc = RC_IntegerTooLarge;  }
  else { Leak(Event_False);  rc = RC_Success;  }
  return rc;

fun RSAEP(res : bigint, pk : pkey const, m : bigint const):
  var rc : rcode;  var n, e : bigint;
  rc = OS2IP(n, modulus_p(pk), k);
  rc = OS2IP(e, exponent_p(pk), k);
  if (bigint_lt(m, n)) { Leak(Event_True);
    bigint_modexp(res, m, e, n) ⇝ τ;  rc = RC_Success;  }
  else { Leak(Event_False);  bigint_zero(res);
    rc = RC_MessageRepresentativeOutOfRange;  }
  return rc;

fun RSADP(res : bigint, sk : skey const, c : bigint const):
  var rc : rcode;  var n, e : bigint;
  rc = OS2IP(n, modulus_s(sk), k);
  rc = OS2IP(d, exponent_s(sk), k);
  if (bigint_lt(c, n)) { Leak(Event_True);
    bigint_modexp(res, c, d, n) ⇝ τ;  rc = RC_Success;  }
  else { Leak(Event_False);  bigint_zero(res);
    rc = RC_CipherTextRepresentativeOutOfRange;  }
  return rc;
```

```
fun OAEP_Decode(res : octet[], len : int out, EM : octet const[]):
  var rc : rcode;  var b : bool;  var i, j : int;
  var seed : octet[hLen];  var DB : octet[dbLen];
  MGF1(seed, 0, hLen, EM, 1 + hLen, dbLen);
  i = 0;
  while (i < hLen) { Leak(Event_True);
    seed[i] = EM [1 + i] ⊕ seed [i] ; ;
    i = i + 1;  } Leak(Event_False);
  MGF1(DB, 0, dbLen, seed, 0, hLen);
  i = 0;
  while (i < dbLen) { Leak(Event_True);
    DB[i] = EM [1 + hLen + i] ⊕ DB [i] ;
    i = i + 1;  } Leak(Event_False);
  i = hLen; j = hLen; b = true;
  while (i < dbLen − 1) { Leak(Event_True);
    b = b ∧ DB [i] = 0x00;
    j = j + bool_to_int(b);
    i = i + 1;  } Leak(Event_False);
  i = 0; b = true;
  while (i < hLen) { Leak(Event_True);
    b = b ∧ DB [i] = dLHash [i] ;
    i = i + 1;  } Leak(Event_False);
  if (b ∧ DB [j] = 0x01 ∧ EM [0] = 0x00) { Leak(Event_True);
    *len = dbLen − j − 1; i = 0;
    while (i < *len) { Leak(Event_True);
      res[i] = DB [i + j + 1] ;
      i = i + 1;  } Leak(Event_False);
    rc = RC_Success;  }
  else { Leak(Event_False);
    *len = 0;  rc = RC_DecryptionError;  }
  return rc;
```

```
fun OAEP_Decrypt(res : octet[], rOff : int, rLen : int out,
                 sk : skey const,
                 c : octet const[], cOff : int, cLen : int):
  var rc : rcode;  var i : int;  var ic, im : bigint;
  var EM : octet[k];  var m : octet[maxMLen];
  if (cLen = k) { Leak(Event_True);
    rc = OS2IP(ic, c, cOff, k);
    rc = RSADP(im, sk, ic);
    if (rc ≠ RC_Success) { Leak(Event_True);
      rc = RC_DecryptionError;  }
    else { Leak(Event_False);
      rc = I2OSP(EM, 0, im, k);
      *rLen = maxMLen;
      rc = EME_OAEP_Decode(m, rLen, EM);
      i = 0;
      while (i < *rLen) { Leak(Event_True);
        res[rOff + i] = m [i] ;
        i = i + 1;  } Leak(Event_False);  }
  else { Leak(Event_False);  rc = RC_DecryptionError;  }
  return rc;
```