Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 201 (2008) 31–44
www.elsevier.com/locate/entcs

Reduction and Refinement

Eerke Boiten and Dan Grundy[1,2]

Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, UK

Abstract

In this paper we explore the relation between refinement and reduction, especially as it is used in the context of cryptography. We show how refinement is a special case of reduction, and more interestingly, how reduction is an instance of a novel generalisation, "refinement with context".

Keywords: Refinement, reduction, cryptography, complexity, IO-refinement.

1 Introduction

This paper forms part of ongoing work by the authors on the application of techniques from formal methods and "mathematics of program construction" to the security area [11,4]. A continuing frustration of ours is the nature of the proofs of security of modern[14] [3] cryptographic constructions.

Roughly speaking, cryptography is the study of constructions where some of the computations involved are deliberately easy (i.e., can be carried out in polynomial-time), while others are deliberately hard (i.e., cannot be carried out in probabilistic polynomial-time). Rather than relying on tacit assumptions that a cryptosystem is "secure", the goal is to prove that breaking it, under a suitable definition of what it means to "break" the cryptosystem, is

[1] Email: E.A.Boiten@kent.ac.uk, WWW: www.cs.kent.ac.uk/~eab2.
[2] Email: dcg2@kent.ac.uk.
[3] I.e., typically it may involve some degree of probabilism, the black-box encryption assumption does not hold, and the notion of security is concerned with leakage of information rather than being all-or-nothing (e.g., revealing complete plaintexts).

doi:10.1016/j.entcs.2008.02.014

computationally intractable; i.e., the goal is to prove that the hard computations are indeed hard.

Unfortunately, high level composable sound abstractions are very thin on the ground in this area. Consequently, a typical proof obligation is that all algorithms in a given probabilistic complexity class have a particular property. However, since there are no useful induction theorems for these classes of algorithms, the only available proof technique is by *reduction to contradiction*: the proof that every algorithm D in class C satisfies P is by taking a hypothetical algorithm D satisfying $\neg P$, and using it as a subroutine ("oracle") in an algorithm for efficiently solving an intractable problem; by contradiction, then, all D satisfy P. In the context of cryptography, we show that any attack is at least as hard as some other problem assumed to be intractable, by demonstrating that the existence of an attack would imply the ability to solve a computationally intractable problem, and so conclude that any such attack must also be computationally intractable.

(We used the word "assumed" in the last sentence because of unresolved questions about the complexity classes, in particular the infamous $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ problem. However, for all relevant complexity classes there are problems strongly believed to be outside them, and disproving these beliefs would take a large section of computer science down with them, so we will not worry about icebergs here.)

To give a simple example, the well-known RSA cryptosystem [17] is — roughly speaking — based on the difficulty of factoring integers: it is easy to show that if factoring is easy then breaking RSA is easy [4]. However, this is an upper bound on the difficulty of attacking RSA: it asserts that attacking RSA is no harder than factoring; far more useful would be to show that attacking RSA is at least as hard as factoring, by showing how the existence of an attack against RSA would imply the ability to factor.

Due to the intricacy and complex nature of the constructions, proofs of this kind are rarely performed at the detailed level of formality that would allow them to be mechanised or machine-checked. (Even experts in this field have a low confidence in the full correctness of such proofs in general, and have suggested mechanisation [12].) By contrast, the field of formal program development, or "refinement", has a well-developed understanding of mechanisable correctness proofs. In this paper we explore the relation between the concepts of *reduction* and *refinement*, our aim being to discover whether there is a set of generalisations and enhancements to (mechanisable) refinement theory that can be used to support reduction proofs.

[4] Here, breaking RSA amounts to computing a private key from a public key, which would be considered a "complete" break of RSA.

In Section 2 we define the concept of reduction and briefly explain its use in complexity theory and cryptography. In Section 3 we explore the mutually dependent concepts of refinement and implementation. In Section 4 we examine how refinement can be viewed as a special case of reduction; the more useful and general view of reduction as a special case of generalised refinement is set out in Section 5. Finally, we draw some conclusions and indicate areas for further research.

2 Reduction

Informally, a problem P *reduces* to another problem Q if any solution to Q can be used to construct a solution to P; more specifically, a reduction is a program that solves P by using a solution to Q as a subroutine. Often the problems involved are decision problems, which can be phrased as testing membership of a certain set (which may or may not be computable). In this very general sense, every computable problem P reduces to every other problem, as a solution can always be constructed — namely, one which does not actually use the solution to Q. However, the more relevant specialised definition in complexity theory assigns a complexity to the construction in order to relate the complexity classes that P and Q belong to.

Definition 2.1 (Reduction) Given specifications P_n and Q_n, indexed with the size of their input, P reduces to Q with complexity $c(n)$ if there exists an algorithm $A_{n,O}$ such that:

- $A_{n,O}$ uses an oracle O for Q_n as a subroutine; counting each call to this oracle as a single step, it has time complexity $c(n)$. The notation $A_{n,B}$ indicates the replacement of the oracle by an actual algorithm B, resulting in an algorithm which has time complexity at least $c(n)$.
- for all algorithms B we have that:

$$B_n \textbf{ implements } Q_n \Rightarrow A_{n,B} \textbf{ implements } P_n$$

With no further constraints on A, the reduction is called a *Turing reduction*. If P_n and Q_n are decision problems, and $A_{n,O}$ is constrained to a single call to O at the end, the reduction is a called a *many-one reduction*[5].

As alluded to above, in order to compare the relative complexities of P and Q we need to consider the complexity of the reduction. It is, for example, easy to reduce an $\mathcal{NP}$-complete problem to a trivial (e.g., constant time)

[5] Many-one reductions are so-named since the transformation need not be injective; the term "one-reduction" is sometimes used to describe injective (many-one) reductions.

problem if we allow the reduction algorithm to solve the $\mathcal{NP}$-complete problem in exponential time[6]. However, such a reduction is of little interest, since performing the reduction is at least as hard as solving the original problem, and so tells us nothing about the complexity relationship between the two problems. Consequently, in order to relate the complexity of P and Q, we additionally bound the complexity of the construction; that is, we bound the rate of growth of the complexity function $c(n)$.

Additionally, although we consider the oracle to compute its solution in a single step, in order to make useful assertions about the complexity of P on the basis of the (usually assumed) complexity of Q, our bound on the complexity of the reduction must also consider the complexity of instantiations of the oracle, where the bound used will depend on the granularity of comparison we intend to make. For example, if we want to establish that P is tractable if Q is tractable, we require only that the reduction algorithm runs in overall polynomial-time when the oracle for Q is replaced by a tractable (i.e., polynomial-time) implementation. However, if we want to make a finer grained comparison, e.g., by showing that P is quadratic if Q is quadratic, then the reduction may only make a constant number of calls to the oracle, since the reduction algorithm must run in overall quadratic time when the oracle is replaced with a quadratic implementation.

In the context of cryptography, we restrict the reductions to efficient (i.e., tractable) computations, since we restrict our attention to feasible, and hence efficient attacks. In practice this means we restrict our attention to probabilistic polynomial-time, since this captures our notion of "efficient" algorithms. Under this definition it may be possible to attack a cryptosystem in (say) exponential time, but since we consider such computations infeasible (i.e., intractable), if follows that we also consider such an attack infeasible. Of course, a proof that no attack exists even in the face of unbounded resources would constitute a strong proof of security, but this is rarely a realistic requirement.

We mention, in view of the importance of efficient computations, that a "polynomial-time reduction" is a reduction that is computable in polynomial-time. If it is a many-one reduction, it is called a *Karp reduction* ; if it is a Turing reduction (but not a many-one reduction) it is called a *Cook reduction*[7].

Two problems are called *computationally equivalent* if they mutually reduce to one-another. Observe that two problems may be computationally

[6] All $\mathcal{NP}$ problems can be solved in exponential time.

[7] We mention also that other notions of reduction exist (e.g., Levin reductions), but that in practice Karp and Cook reductions are the most common types of reduction one encounters (at least in the context of complexity theory).

equivalent but not in the same complexity class. For example, many natural problems (including all $\mathcal{NP}$-complete problems) are *self-reducible*, meaning the search variant of the problem reduces to the decision variant. Going in the opposite direction, it is clear that if we can solve the search version of a problem then we can solve the decision version. Hence the search and decision versions of self-reducible problems are computationally equivalent, even though the two variants lie in different complexity classes (since complexity classes for decision problems and search problems are disjoint by definition).

A problem is called *C-complete* for a complexity class C if it is a member of C, and every other problem in C can be reduced to it using resources that do not exceed the bounds of the class [8]. For example, a problem is $\mathcal{P}$-complete if it is in $\mathcal{P}$ (the class of decision problems solvable by polynomial-time algorithms) and every other problem in $\mathcal{P}$ polynomially (i.e., Karp) reduces to it. By definition, a solution to any of the complete problems for some class yields a solution to every problem in the class, therefore the complete problems are the hardest in the class. If we view reduction as capturing the notion that one problem is rich enough to encode another, then completeness captures the notion that one problem is rich enough to encode an entire class of problems.

Note that the above assumes languages for problems and their algorithmic solutions, an implicit correctness relation between them, and a notion of time complexity for the algorithmic language. There is no mention of refinement in there yet, but it is just around the corner ...

3 Refinement and Implementation

The definition of reduction given above uses a notion of "implementation". This gives us a notion of refinement either directly or indirectly, depending on whether we consider specifications and implementations to be in separate languages. Readers with a background in program derivation will probably find no big surprises in this section [9].

[8] Cook's [9], and independently, Levin's [13], fundamental contribution to complexity theory (known as the Cook-Levin theorem) was to show the existence of $\mathcal{NP}$-complete problems. By now a large number of natural problems have been shown to be $\mathcal{NP}$-complete; our inability to efficiently solve such a large class of problems offers a strong basis for the common belief that $\mathcal{P} \neq \mathcal{NP}$.

[9] A similar discussion is included in [5] which drives home the point that having semantics, refinement, conformance, or consistency for UML are very closely related issues.

3.1 Wide Spectrum Languages

So-called wide spectrum languages [15,2] include executable programs alongside non-operational specification constructs: programs *are* specifications. In such a language, there is no strict distinction between refinement and implementation. The only distinction one could make is that certain specifications are considered executable, and only those are in the domain of the implementation relation; i.e., (using $==$ for definitional equality)

$$\textbf{implements} \;==\; \textbf{executable} \lhd \textbf{refines} \tag{1}$$

The set **executable** might contain only deterministic specifications, or ones where all non-determinism is due to explicit concurrency, or feasible specifications only, or programs executable within given time limits, etc.

Crucially, implementation inherits most of the properties of refinement: it is reflexive (on implementations) and transitive if refinement is, inherits lattice properties, and is monotonic under an operation if the set of executables is closed under it and refinement is monotonic. All this is immediate from **implements** $\subseteq$ **refines**.

3.2 Specifications and Programs, Separately

If specifications and implementations are expressed in different languages, then a notion of implementation (a.k.a. satisfaction or conformance) must already exist in order to characterise correctness. From such a notion of implementation, we can derive a notion of refinement as "inclusion of models" or "inclusion of properties of interest"; i.e.,

$$s \;\textbf{refines}\; s' \;==\; \forall\, p \bullet (p \;\textbf{implements}\; s) \Rightarrow (p \;\textbf{implements}\; s') \tag{2}$$

Additionally, it may be required that s is feasible, i.e., $\exists\, p \bullet p \;\textbf{implements}\; s$, or feasible if s' is, to avoid trivial refinements.

For example, when the specification language is some kind of logic, and implementation is satisfaction, then one would expect refinement to be implication. (If it is not, then the logic has a very unusual notion of satisfaction.)

We have given definitions of refinement in terms of implementation and vice versa; a sanity check is whether substituting one definition in the other makes sense. Substituting (1) in (2) gives

$$\forall\, s, s' \bullet \quad s \;\textbf{refines}\; s' \equiv (\forall\, p : \textbf{executable} \bullet p \;\textbf{refines}\; s \Rightarrow p \;\textbf{refines}\; s') \tag{3}$$

which suggests a lattice-like property on the refinement ordering, viz. that every minimal upper bound of a set of implementations (if it exists) refines every existing minimal upper bound of every extension of that set. Informally, a specification is equivalent to its set of implementations. If no distinction be-

tween executable and non-executable specifications is made, then transitivity and reflexivity are the only properties required to prove that (3) holds.

Certainly the above characterisation of refinement has a rich set of properties that it inherits from set inclusion. However, the asymmetry in the implementation relation means that it does not provide monotonicity properties for refinement to inherit [10].

In the rest of the paper we consider a generic relation **refines** which is either assumed to be pre-defined or derived from a generic implementation relation using (2). Properties that such relations will normally have include transitivity and reflexivity (allowing for stepwise refinement), and monotonicity under various specification operators (allowing for piecewise refinement).

4 Refinement as Reduction

In this section we first observe that refinement can be viewed as a very particular kind of reduction. We then explore a generalisation of (generic) refinement, viz. IO-refinement, which may cover a wider range of reductions.

4.1 Refinement is Reduction

We have related reduction to implementation in Definition 2.1, and implementation to refinement in properties (2) and (1). The similarity between the defining equation of reduction and property (2) stands out. In particular, we have the following basic result.

Theorem 4.1 *If the relation* **refines** *satisfies property (2) then Q* **refines** *P implies that P reduces to Q with complexity 1. If P and Q are decision problems, this reduction is a many-one reduction.*

Proof. Since Q refines P, and **refines** satisfies (2), every implementation of Q is also an implementation of P, and so any implementation of our oracle for Q is also an implementation of P, consequently our reduction algorithm comprises a single call to the oracle and hence has complexity 1. As it uses the oracle only once at the end, it is a many-one reduction if P and Q are decision problems. □

The precondition of this theorem holds in many circumstances, for example for

[10] The natural generalisation is that $p \oplus p'$ **implements** $s \otimes s'$ if p **implements** s and p' **implements** s', i.e., $\otimes$ is "implemented by" $\oplus$, but the proof that $\otimes$ inherits monotonicity from $\oplus$ would require "iff" rather than "if". Monotonicity of specification operators is much more likely to follow through operators on their sets of implementations rather than operators on individual implementations.

many process algebra and relational refinements and standard Z refinement [10]. However, the implication is really in the less useful direction: it positions the problem we can solve as a special case of the one which we would like to solve. We need, then, to look for the other direction, namely: how we can generalise refinement relations in order to characterise (and ultimately, to verify) a wider class of reductions.

4.2 IO-Refinement

IO-refinement is a generalisation of refinement that allows us to decouple the interfaces of specifications and implementations. For example, the specification may refer to numbers or strings, but IO-refinement would allow it to be implemented in terms of windows and mouse clicks. Another use of IO-refinement occurs when an implementation is available with the right functionality but not quite the right interface for the specification. The IO-transformation then could fix some of the implementation's input values or ignore some of its outputs, for example.

The particular instance of IO-refinement for Z abstract data types is described in [7] and [10, Chapter 10]. To define IO-refinement for our generic notion of refinement, we need notions of input/output transformers, and to assume an operation in our specification language that allows pre-composing a specification with an input transformer, and post-composing it with an output transformer. The input/output transformers, and the composition operation, which we will denote $\fatsemi$, will have different instantiations in different specification notations; e.g., see [7,10] for the Z version of input/output transformers and the $\gg$ operator that is used to compose them with Z operations and with each other.

Definition 4.2 (IO-refinement) For specifications s and s', s **IO-refines** s' using input transformer it and output transformer ot iff

$$it \fatsemi s \fatsemi ot \quad \textbf{refines} \quad s'$$

When it is the identity, the refinement is called an *output refinement*; when ot is the identity the refinement is called an *input refinement*. (If both are the identity we have the usual notion of refinement.)

This normally implies that it is total, i.e., it transforms every possible input of s' into some output of s, and that ot is injective: it transforms outputs of s into outputs of s' while retaining enough information to allow the abstract outputs to be reconstructed.

Of course, IO-refinement is only really useful if the decomposition provided

by separating s' into it, s, and ot is a true decomposition: degenerate cases shift the entire specification into it or ot and leave s trivial; the more interesting cases (including the examples above) are when at least one of these has a known implementation already.

Defining the reduction program by $A_{O,n} := it \fatsemi O \fatsemi ot$ appears to make IO-refinement an instance of reduction. However, the complexity of the reduction is not fixed by this: the input transformer it might not be operational (deterministic, executable); the complexity of the output transformation is even harder to pinpoint as it should relate to the size of the *input* of s' which is not in the scope of ot. Moreover, this would still only account for reductions which use the oracle exactly once. A generalisation of Theorem 4.1 for a limited class of IO-transformers could be given. However, given the limited range of reductions corresponding to refinements that would result, we will look for a joint generalisation of the two concepts instead.

5 Refinement with Context

As stated previously, we would prefer a formalisation that characterises reduction as a special case of (generalised) refinement as this would allow us to apply our knowledge (mechanised or not) of refinement to the problem of reduction. We present such a formalisation based on the commonality present in both IO-refinement and reduction as defined above, viz. that the "concrete" specification is first put in a context, and only then do we check for refinement.

Traditional data refinement is proved using simulations [10]: the refinement conditions posit the existence of a simulation relation (retrieve relation, coupling invariant, ...) that relates concrete and abstract states. However, the role of such a simulation relation is purely existential: no knowledge of the simulation is required in order to use the concrete data type instead of the abstract one. (That said, it does appear in meta-level proofs, e.g., to prove that downward simulation is transitive, where the derived simulation is the sequential composition of the base ones.)

Although we can safely ignore the simulation relation in data refinement once we have proved a refinement step correct, notions of refinement that modify the specification's interface need to maintain extra information that tells clients how the concrete interface they have been given corresponds to the abstract interface they asked for. IO-refinement is just one such notion; [10] describes, in addition, non-atomic refinement and alphabet translation. In order to construct concrete inputs from abstract ones, and to reconstruct abstract outputs from concrete ones, the transformers it and ot need to be pro-

vided to the user. This is reflected in Definition 4.2: it defines IO-refinement modulo *it* and *ot*, where they appear both as a part of the notion and its characterisation, and they are not (explicitly or implicitly) existentially quantified.

A similar issue arises with reduction: it is defined modulo the complexity of the reduction algorithm A. In the refinement-like characterisation of reduction in Definition 2.1, the algorithm A is used in the defining property, and although it is existentially quantified, its complexity is part of the notion. For many uses of reduction, less information is required; for example, it may be sufficient to know whether A belongs to a particular complexity class.

Generalising these two examples, we come to a notion of "refinement with context", where the "concrete" specification is viewed in a specific context, and the refinement relation is decorated with that context. (This is different from refinement *in* context, where the same context is applied to *both* sides.) Below, we define a generalisation, which records an abstraction of the context rather than the context itself.

Informally, for a given set of contexts C, P is refined by Q with context c if P is refined by "Q put in a context c from C". Contexts are often defined as "specifications with a hole"; however, rather than taking this more syntactic approach, we will define them as sets of specification transformers, i.e., functions from specifications to specifications. Allowing *any* function of that type makes for a completely meaningless notion of refinement with contexts; rather, we need to characterise sets of such functions which will guarantee that the derived refinement notion has desirable properties.

Definition 5.1 (Contexts) Given a specification language $\mathcal{L}$ with a reflexive and transitive relation **refines** on $\mathcal{L}$. A *context set* for $(\mathcal{L}, \mathbf{refines})$ is a collection of functions $\mathcal{C} : \mathbb{P}(\mathcal{L} \to \mathcal{L})$ such that

- $\mathcal{C}$ contains the identity function $1_{\mathcal{L}}$ (or one for each type, if $\mathcal{L}$ is typed);
- $\mathcal{C}$ is closed under function composition;
- the elements of $\mathcal{C}$ are monotonic with respect to **refines**.

The presence of identity transformers ensures that the refinement relation with contexts from $\mathcal{C}$ generalises **refines**, and consequently that it is reflexive. $\mathcal{C}$ being closed under composition ensures transitivity. Monotonicity transfers monotonicity properties of the underlying refinement relation [11].

Example 5.2 The context set for Z input refinement is $\{it : IOT \bullet \lambda s \bullet it \gg s\}$ where IOT is the set of all IO-transformers (see [10] for details), and

[11] There appears to be rich categorical structure in here.

$\gg$ is restricted to pairs of schemas with perfectly matching output/input names. The operator $\gg$ is indeed monotonic with respect to refinement in this context; composition of contexts corresponds to composition using $\gg$ of the IO-transformers. Identities also exist for each type.

The context set for Z output refinement is nearly identical, post-composing with an IO-transformer instead. For full IO-refinement in Z, the context set is defined by pre- and post-composition with IO-transformers.

Example 5.3 For reduction, we fix an extended set of specifications as "algorithms using an oracle". By including the trivial algorithm that just makes a single oracle call, we also include basic specifications (such as P and Q in Definition 2.1), which are thus effectively identified with their oracles. The application of a context corresponds to inclusion of algorithms as subroutines in larger algorithms. The trivial algorithm is the identity context, and substitution in programs is closed under composition. The required monotonicity is the basis of top-down programming: correct implementation through the correct implementation of subroutines.

We could now give a definition of refinement with contexts; however, this would decorate the refinement with a specific context, which may be more information than is required. In the case of reduction, for example, it is the *complexity* of the mediating algorithm which matters, rather than the algorithm itself. Generalising this observation, we move towards a refinement notion that is decorated with some abstraction of the context, rather than the context itself. We reiterate a point made for the specific cases of reduction and IO-refinement, namely that the process we are looking at is one of *decomposition*: a problem (specification) is decomposed into a "context" and a "simpler" problem. Such decompositions need not always be useful: for example, they may shift all the difficulty into the context, and declare our problem solved in that way [12]. The following definition reflects that we consider this as "cheating", by introducing a *penalty* function to put a price on (the difficulty of) the work shifted into the context.

Definition 5.4 (Refinement with penalty) Given a context set $\mathcal{C}$ for $(\mathcal{L}, \mathbf{refines})$, and a (fixed) function $f : \mathcal{C} \to \mathcal{D}$, this induces a notion of refinement with penalty as follows. We say that Q *refines* P *with penalty* p iff

$$\exists\, c : \mathcal{C} \bullet c(Q) \ \mathbf{refines}\ P \quad \wedge \quad f(c) = p$$

Where f is the identity function, we say that Q *refines* P *with context* c.

[12] Another example of this is the continuation passing style transformation in functional programming: the continuation being passed around will contain all the "work".

Example 5.5 For IO-refinement (not necessarily just in Z), the penalty function is the identity relation, the input and output transformers themselves represent the additional work.

Example 5.6 Reduction (as in Definition 2.1) is refinement with a penalty determined by the complexity of the reduction algorithm. Note that here the penalty function is not compositional i.e., the penalty of combined reduction steps cannot be computed from the penalties of the individual steps [13]. However, in most cases only the complexity *class* of the reduction algorithm is relevant, and classes of interest (e.g., polynomial) are closed under this substitution.

Note that we have put no constraints on the function p in Definition 5.4. Clearly compositionality would have been nice, but is too much to ask for in view of the above example. Sensible constraints derive from a topological view, with (contexts and) p representing a notion of distance between specifications, and might include properties like $f(c_1 \fatsemi c_2) \geq \mathbf{max}\,(f(c_1), f(c_2))$.

As usual, equivalence relations on specifications can be defined in terms of mutual refinement.

Definition 5.7 (Equivalence with contexts) Given a context set $\mathcal{C}$ for $(\mathcal{L}, \mathbf{refines})$, this induces a notion of equivalence with contexts as follows. We say that Q *is equivalent to* P *with contexts* (c_1, c_2) if Q refines P with context c_1 and P refines Q with context c_2. We say that P *and* Q *are isomorphic* if additionally $c_1 \fatsemi c_2 = c_2 \fatsemi c_1 = 1_{\mathcal{L}}$.

Example 5.8 If IO-refinement holds in both directions with bijective transformers, we have isomorphic specifications. However, it is also possible for refinement in one direction to be achieved using an input transformer, and in the other direction using an output transformer, in which case their composition clearly is not the identity for all specifications.

Example 5.9 For reduction, the induced notion of equivalence is computational equivalence as discussed previously.

[13] Consider a linear time algorithm AP with an oracle for Q. Its unit steps may be due either to oracle calls or to other elementary operations; thus, we abstract away from how many calls it makes to the oracle for Q. If we substitute for the oracle an algorithm AQ that solves Q, we cannot determine the resulting complexity from the individual complexities. For example, if AP uses a single oracle call, and AQ is quadratic, then the overall complexity is quadratic; however, if AP uses n oracle calls, then the resulting complexity with the same AQ is cubic.

6 Further Work

Definition 5.4 represents one way towards achieving our goal: we have defined a novel generalised notion of refinement, phrased in terms of (a generic notion of) refinement, that encompasses both reduction and several generalised notions of refinement.

In terms of our issues with reduction as a proof technique, it does not solve everything. In particular, we have not provided any significant advances to the process of *finding* an algorithm to show that a particular reduction works (i.e., solving the existential quantification in Definition 5.4). However, the characterisation above may help to support the verification of reductions, possibly using (even existing) mechanisations.

We mentioned probabilistic algorithms early on in the paper, but not much after that. These come naturally with a range of refinement and implementation relations. *Perfect* implementation results in identical distributions; *statistical* implementation only requires distributions that are statistically close. *Computational* implementation weakens this further to distributions that cannot be distinguished with a significant advantage by probabilistic polynomial algorithms. The latter is often the realistically achievable notion of security [14]. See [4] for a characterisation of this kind of refinement in the framework of approximate refinement [6]. For the most part, it does not matter that we have glossed over the probabilistic aspect. All of these implementation relations give rise to refinement relations with the usual properties, and so fit our generic scheme. Our discussion of reduction also did not fix the complexity classes of interest, which usually shift from worst-case considerations to average-case ones when moving into the probabilistic world (particularly in the context of cryptography). For details of this, see [11]. A generalisation of the computation of penalties may also be necessary in order to account for the difference allowed by approximate refinement relations.

Further work should take this probabilistic aspect fully into account, and expand the ideas in this paper into practical mechanisms applicable to problems in cryptography. In particular, they should be related and applied to the two main efforts at increased abstraction for cryptographic verification that have arisen in the theoretic cryptography community: "game-hopping" [3,18] and universal composability [8]/ reactive simulability [16]. The latter approach in particular already has a strong formal methods influence, using probabilistic IO-automata; however, their formalisation has not yet matured into one that is communicable and easily mechanisable in general [14].

[14] However, see [1] for a report on a specific verification in this framework, mechanised in PVS.

References

[1] Backes, M., C. Jacobi and B. Pfitzmann, *Deriving cryptographically sound implementations using composition and formally verified bisimulation.*, in: L.-H. Eriksson and P. A. Lindsay, editors, *FME*, Lecture Notes in Computer Science **2391** (2002), pp. 310–329.

[2] Bauer, F., R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing and H. Wössner, "The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L," Lecture Notes in Computer Science **183**, Springer, Berlin/Heidelberg/New York, 1985.

[3] Bellare, M. and P. Rogaway, *The security of triple encryption and a framework for code-based game-playing proofs.*, in: S. Vaudenay, editor, *EUROCRYPT*, Lecture Notes in Computer Science **4004** (2006), pp. 409–426.

[4] Boiten, E., *Commitment is hard: Reconstruction of a cryptographic primitive from a formal methods perspective* (2007), submitted for publication.

[5] Boiten, E. and M. Bujorianu, *Exploring UML refinement through unification*, in: J. Jürjens, B. Rumpe, R. France and E. Fernandez, editors, *Critical Systems Development with UML - Proceedings of the UML'03 workshop*, TUM-I0323 (2003), pp. 47–62.
URL http://www.cs.kent.ac.uk/pubs/2003/1742

[6] Boiten, E. and J. Derrick, *Formal program development with approximations*, in: H. Treharne, S. King, M. Henson and S. Schneider, editors, *ZB 2005*, Lecture Notes in Computer Science **3455** (2005), pp. 375–393.

[7] Boiten, E. A. and J. Derrick, *IO-refinement in Z*, in: A. Evans, D. J. Duke and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop* (1998).
URL http://www.bcs.org/server.php?show=ConWebDoc.4354

[8] Canetti, R., *Universally composable security: A new paradigm for cryptographic protocols*, Cryptology ePrint Archive, Report 2000/067 (2000).
URL http://eprint.iacr.org/

[9] Cook, S., *The complexity of theorem proving procedures*, in: *Proceedings of the third annual ACM Symposium on Theory of Computing (STOC)*, 1971, pp. 151–158.

[10] Derrick, J. and E. Boiten, "Refinement in Z and Object-Z: Foundations and Advanced Applications," FACIT, Springer Verlag, 2001.

[11] Grundy, D., Ph.D. thesis, Computing Laboratory, University of Kent (Forthcoming).

[12] Halevi, S., *A plausible approach to computer-aided cryptographic proofs*, Cryptology ePrint Archive, Report 2005/181 (2005), http://eprint.iacr.org/.

[13] Levin, L., *Universal'nye perebornye zadachi*, Problemy Peredachi Informacii **9** (1972), pp. 265–266, in Russian; an English translation appears in "Universal Search Problems", in B.A. Trakhtenbrot, "A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms", IEEE Annals of the History of Computing, 6(4):384–400, 1984.

[14] Mao, W., "Modern Cryptography – Theory & Practice," Hewlett-Packard Professional Books, Prentice Hall, 2004.

[15] Morgan, C. C., "Programming from Specifications," International Series in Computer Science, Prentice Hall, 1994, 2nd edition.

[16] Pfitzmann, B. and M. Waidner, *Composition and integrity preservation of secure reactive systems.*, in: *ACM Conference on Computer and Communications Security*, 2000, pp. 245–254.

[17] Rivest, R., A. Shamir and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), pp. 120–126.

[18] Shoup, V., *Sequences of games: a tool for taming complexity in security proofs*, Cryptology ePrint Archive, Report 2004/332 (2004).
URL http://eprint.iacr.org/