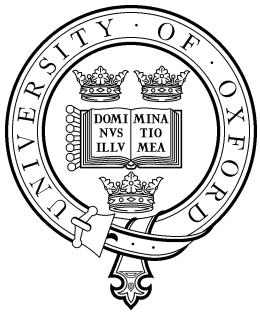
Programming Research Group

A COMPUTATIONAL JUSTIFICATION FOR GUESSING ATTACK FORMALISMS

Tom Newcomb and Gavin Lowe

PRG-RR-05-05



Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Recently attempts have been made to extend the Dolev-Yao security model by allowing an intruder to learn weak secrets, such as poorly-chosen passwords, by *off-line guessing*. In such an attack, the intruder is able to verify a guessed value g if he can use it to produce a value called a *verifier*. In such a case we say that g is *verifier-producing*. The definition was formed by inspection of known guessing attacks.

A more intuitive definition might be formed as follows: a value is verifiable if there exists some computational process that can somehow *recognise* a correct guess over any other value.

We formalise this intuitive definition, and use it to justify the soundness and completeness of the existing definition. Specifically we show that a value is recognisable if and only if the value is either already Dolev-Yao deducible or it is verifier-producing. In order to do this it was necessary to clarify the definition of verifier production slightly, revealing an ambiguity in the original definition.

1 Introduction

Problem Statement. Some security protocols are vulnerable to *guessing attacks*, where an intruder can guess a value not otherwise known to him, and verify the correctness of this guess using messages he has learned. This is a problem particularly for protocols that use user-chosen passwords.

For example, consider the following simple protocol, which aims to authenticate a user a to a server s using a symmetric key p formed from a shared password:

$$\begin{array}{lllll} \text{Message 1.} & s \to a & : & n_s \\ \text{Message 2.} & a \to s & : & \{\!|n_s|\!\}_p. \end{array}$$

The server s sends a fresh nonce n_s to a, who replies by encrypting this nonce with the shared password p.

An intruder overhearing this exchange would be able to guess a value for p, and use it to decrypt the ciphertext from Message 2. If the result is equal to the plaintext from Message 1, the intruder may deduce (with high probability) that he has guessed p correctly. Of course, this 'guessing' may be automated, by iterating through some suitable dictionary, using each value in turn.

Some assumptions are made:

- We assume that values have no entropy: for example, an intruder is not able to test whether a sequence of bits he encounters represents a nonce or a key, and he can never immediately detect if he decrypts a piece of ciphertext with the wrong key.
- We will be assuming that certain values used in protocols have a non-negligible probability of being guessed using a feasible amount of resources, for example, if they appear in a dictionary.
- We consider only *off-line* guessing attacks where the intruder does not require interaction with the protocol in order to check correctness of a guess; on-line attacks can

be detected and prevented using other means, such as blocking multiple incorrect guesses.

These attacks are not captured by the standard Dolev-Yao model [DY83] where an intruder's knowledge at any moment is defined as the closure of directly observed messages under a set of production steps.

Previous work. Lowe [Low02, Low04] has extended the Dolev-Yao model to allow the intruder to perform guessing attacks as follows. At any point in the protocol the intruder can guess a value and then attempt to verify that guess; if the verification is successful he may add the guess to his knowledge and continue.

By Lowe's definition, an intruder verifies a value g if he can use it to produce a *verifier* v satisfying any of:

- v can be produced in two different ways,
- v is a value the intruder already knew, or
- v is an asymmetric key, and the intruder knows its inverse.

These conditions were formed by inspection of known guessing attacks, and appears slightly ad-hoc: it is stated in [Low04] that 'it is hard to be sure that there are no others.' Also, the formalisation of this definition is quite lengthy and contains some unnatural subtleties.

There are other extensions of the Dolev-Yao model that capture guessing attacks [Coh02, DJ04, CMAFE03, Cho04]. However, these are all reformalisations of Lowe's original definition in different frameworks, and we expect that results about Lowe's framework can be easily adapted for these other extensions.

This paper. We propose a more intuitive, computational definition that captures the essence of guess verification:

> An intruder can verify a guess of g if there exists a program that behaves in an observably different way on input g than on any other input.

We formalise our definition and show how it applies to some example protocols. We will refer to guess verification in this form by saying that the value is *recognisable*.

Despite being simpler and more natural, there is a major disadvantage of this definition: it involves a quantification over all programs, making it difficult to automate directly. On the other hand, [Low04] uses verifier production in a decision procedure for the automatic verification of protocols, which is shown to be effective on real-world examples.

We relate these two definitions by proving that a guess is recognisable if and only if it is either already deducible within the Dolev-Yao model or it is verifier-producing. This is a non-trivial result because the arbitrary recognising program may:

- Make use of programming control structures, e.g. conditionals and loops.
- Possess redundancy (i.e. contain behaviour not optimal or necessary for the guessing attack to succeed), which is not permitted in a verifier-production attack trace.

- Create malformed terms, e.g. by decrypting a ciphertext with the wrong key; this might be useful in a guessing attack, but a verifier-production trace does not allow it.

Contributions. We present a simple and intuitive computational definition of guess verification. This is in contrast to previous formalisations which were defined by capturing the characteristic behaviours of known attacks.

We relate these definitions by proving their equivalence; in doing so, we require a slight strengthening of the definition of verifier production. The contribution of our work can be seen as follows:

- It gives a justification for the definition of verifier production: technically, it shows that Lowe's algorithm is sound and complete with respect to our more computational model.
- It provides a decision procedure (i.e. that given in [Low04]) for finding guessing attacks for our more natural definition of guess verification.
- It exposes an ambiguity in the verifier-production definition.

As far as we are aware there have been no attempts to computationally justify such verifier-production techniques for analysing guessing attacks.

Organisation. In Section 2, we give the existing definition of guess verification from [Low04]. We motivate and describe our new definition in Section 3, and formalise it in Section 4. We prove the completeness and soundness of the original definition with respect to our new definition in Sections 5 and 6 respectively. Conclusions and future work are given in Section 7.

2 Existing definition: verifier production

In this section, we give the definition of guess verification from [Low04]; for more explanation and motivation, consult that paper.

First, we describe the standard Dolev-Yao deduction rules. These describe how an intruder may use learned and initially-known facts to deduce new facts:

$$\begin{array}{rll} \{f, f'\} & \vdash_{\mathsf{pair}} & (f, f'), \\ \{(f, f')\} & \vdash_{\mathsf{fst}} & f, \\ \{(f, f')\} & \vdash_{\mathsf{snd}} & f', \\ \{f, k\} & \vdash_{\mathsf{enc}} & \{\!|f|\!\}_k, \\ \{\{\!|f|\!\}_k, k^{-1}\} & \vdash_{\mathsf{dec}} & f. \end{array}$$

A series of deductions $IK \models_{tr} IK'$ is defined by the following rules:

$$\begin{array}{rcl} & & IK \models_{\langle\rangle} IK, \\ S \subseteq IK \wedge S \vdash_l f \wedge IK \cup \{f\} \models_{tr} IK' & \Rightarrow & IK \models_{\langle S \vdash_l f \rangle^\frown tr} IK'. \end{array}$$

We refer to tr as a D-Y trace, and say that IK' (or a value in IK') is D-Y deducible from IK.

We now give the verifier-production definition of guess verification. An intruder verifies a guess g using verifier v from knowledge IK if there exist IK', S, S', l, l' such that either Conditions (1)–(5) hold, or Condition (6) holds.

Firstly, the intruder uses the initial knowledge and the guess to perform a sequence of deductions:

$$IK \cup \{g\} \models_{tr} IK'. \tag{1}$$

One of these deductions must produce the verifier v:

$$S \vdash_l v \ \ \textbf{in} \ \ tr. \tag{2}$$

It must be impossible to obtain the information necessary for the deduction without knowing g:

$$\nexists IK'' \cdot (IK \models IK'' \supseteq S). \tag{3}$$

The verifier must satisfy one of the following properties: (a) it can be produced in a second, different way; (b) the intruder already knew the value; or (c) it is an asymmetric key, and the intruder knows its inverse.

$$\begin{array}{ll} S' \vdash_{l'} v \text{ in } tr \wedge (S,l) \neq (S',l') & \text{(a)} \\ \vee & \\ v \in IK \cup \{g\} & \text{(b)} \\ \vee & \\ v \in ASYMMETRIC\_KEYS \wedge v^{-1} \in IK'. & \text{(c)} \end{array} \tag{4}$$

Finally, deductions that simply undo previous deductions are prohibited. Without this condition, certain false attacks are detected:

$$\begin{array}{ll} \forall (S'' \vdash_{l''} v'') \text{ in } tr \cdot & \neg(S \vdash_l v \text{ undoes } S'' \vdash_{l''} v'') \wedge \\ & \neg(S' \vdash_{l'} v \text{ undoes } S'' \vdash_{l''} v''), \end{array} \tag{5}$$

where,

$$\begin{array}{rcl} \{f, f'\} \vdash_{\mathsf{pair}} (f, f') & \text{undoes} & \{(f, f')\} \vdash_{\mathsf{fst}} f, \\ \{f, f'\} \vdash_{\mathsf{pair}} (f, f') & \text{undoes} & \{(f, f')\} \vdash_{\mathsf{snd}} f', \\ \{(f, f')\} \vdash_{\mathsf{fst}} f & \text{undoes} & \{f, f'\} \vdash_{\mathsf{pair}} (f, f'), \\ \{(f, f')\} \vdash_{\mathsf{snd}} f' & \text{undoes} & \{f, f'\} \vdash_{\mathsf{pair}} (f, f'), \\ \{f, k\} \vdash_{\mathsf{enc}} \{|f|\}_k & \text{undoes} & \{\{|f|\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f, \\ \{\{|f|\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f & \text{undoes} & \{f, k\} \vdash_{\mathsf{enc}} \{|f|\}_k. \end{array}$$

There is an additional way that an intruder can verify a guess which is not covered by the above five conditions:

$$g \in ASYMMETRIC\_KEYS \wedge g^{-1} \in IK. \tag{6}$$

We will say that a value g is *verifier-producing* from knowledge IK if it is verifiable according to the above definition. For convenience, we will overload this definition by allowing the initial knowledge IK to be a sequence.

This definition is quite lengthy and contains some subtleties. It is not unreasonable to have doubts about its correctness. In particular, one might ask whether the three sub-conditions of Condition (4) cover all possible ways of verifying a guess.

3 A new definition: recognisability

In this section we show how guessing can be defined more intuitively. We imagine that any intruder performing an off-line guess verification must have to invoke a procedure that informs him whether his guess is correct or not; such a procedure would need to somehow tell the difference between correct and incorrect guesses. This procedure may utilise values that the intruder has been sent, has overheard, or initially knew.

To formalise this, we say that a guess of g is *recognisable* from a sequence of knowledge K if there exists a program P that behaves in some observably different way when provided with input $K^\frown\langle g\rangle$ than when provided with $K^\frown\langle g'\rangle$ for any value $g' \neq g$. To put this in mathematical notation:

$$\exists P \cdot \forall g' \neq g \cdot P(K^\frown\langle g\rangle) \not\simeq P(K^\frown\langle g'\rangle),$$

where $\simeq$ is *observable equivalence* on programs. Without loss of generality, we may assume that K contains no repetitions, i.e. it corresponds naturally to a knowledge set, as for verifier production. While we discuss and motivate this definition, we hold off formally defining a syntax and semantics for programs until Section 4.

We restrict the intruder so he can only guess atomic data values (i.e. he cannot guess a term built up using encryption or pairing). We also only consider well-formed knowledge sequences K. Such restrictions are also imposed by the verifier-production framework to which we will be relating our definition.

Our definition might be considered too general because although it guarantees that g produces a uniquely recognisable output, we may not *a priori* know what that output is. For example, consider a program P that takes a guess g and outputs it in some numeric form. P will produce a unique output for every guess g, but it is certainly not verifying anything. An alternative definition might say that the program P must produce output $\langle 0\rangle$ for a wrong guess g' and $\langle 1\rangle$ for a correct guess g. We show that, within our framework, these two definitions are equivalent (Theorem 5.20). Examples like the one above are not possible because we prevent programs from inspecting values in this way. We have already made the assumption that values have no entropy so there can be nothing to gain from inspecting values at the bit level.

We finish this section with a simple example. Consider the knowledge sequence $K = \langle v, \{\!|v|\!\}_g\rangle$ where g is a symmetric key, and suppose we want to know whether an intruder could verify a guess of g in the recognisability framework. To show that this is possible, we need to find a program P that produces a different output when provided with input

$K^\frown\langle g\rangle$ than when provided with $K^\frown\langle g'\rangle$ for any $g' \neq g$. A program that would do this would

- accept the input guess in a formal parameter x;
- decrypt $\{\!|v|\!\}_g$ with x;
- compare the result with v;
- outputs 1 or 0 if the test is true or false respectively.

This performs the verification because it outputs a 1 for input g and a 0 for all other inputs. (Alternatively, the program could have encrypted v with x to form $\{\!|v|\!\}_x$, and compared this value with $\{\!|v|\!\}_g$.)

4 A language for programs

In this section we present a formal language for the program P in the previous section. We begin by introducing terms and programs, and finish with an example program that recognises a particular value.

4.1 Terms

We assume a set of atomic *data*. A subset of data is *keys*, which is partitioned into *symmetric* keys and *asymmetric* keys. A symmetric function $\cdot^{-1}$ on asymmetric keys associates k with its inverse key k^{-1}.

Our programs will store *terms* in their registers, which are values from an abstract datatype representing concrete bit sequences. An abstraction here is useful because it allows us to cleanly model the algebraic properties of these terms in the program semantics. It also allows us to abstract away from the precise implementation of program operations such as sequencing and encryption.

The set of terms is generated by the following grammar:

$$\begin{array}{rcl} t & ::= & D \\ & | & \mathbf{pair}(t_1, t_2) \\ & | & \mathbf{fst}(t) \\ & | & \mathbf{snd}(t) \\ & | & \mathbf{enc}(t_1, t_2) \\ & | & \mathbf{dec}(t_1, t_2) \\ & | & \mathbf{enca}(t_1, t_2) \\ & | & \mathbf{deca}(t_1, t_2), \end{array}$$

where terminals D are drawn from the set of atomic data. These terms represent: the pairing of data together; the two ways of unpairing data; symmetric key encryption/decryption; and asymmetric key encryption/decryption.

We will deal only with terms that have been fully *reduced*, according to the following rewrite rules.

$$\begin{aligned}
\mathbf{fst}(\mathbf{pair}(t_1, t_2)) &\rightsquigarrow t_1 \\
\mathbf{snd}(\mathbf{pair}(t_1, t_2)) &\rightsquigarrow t_2 \\
\mathbf{pair}(\mathbf{fst}(t_1), \mathbf{snd}(t_1)) &\rightsquigarrow t_1 \\
\mathbf{dec}(\mathbf{enc}(t_1, t_2), t_2) &\rightsquigarrow t_1 \\
\mathbf{enc}(\mathbf{dec}(t_1, t_2), t_2) &\rightsquigarrow t_1 \\
\mathbf{deca}(\mathbf{enca}(t_1, t_2), {t_2}^{-1}) &\rightsquigarrow t_1 \\
\mathbf{enca}(\mathbf{deca}(t_1, {t_2}^{-1}), t_2) &\rightsquigarrow t_1.
\end{aligned}$$

The reader may recall that the verifier-production setting does not differentiate between symmetric and asymmetric encryption schemes. There, the intended encryption scheme can be inferred from the values themselves (i.e. whether or not the key is a member of *ASYMMETRIC_KEYS*). In our framework we are dealing with programs that can accept a class of inputs. It is not known at the program level which scheme of encryption should be used and it becomes necessary to make it explicit.

We also use a different notation for terms/facts than that used in verifier production. We take (f, f) as syntactic sugar for $\mathbf{pair}(f, f')$, and $\{\!|f|\!\}_k$ as syntactic sugar for $\mathbf{enc}(f, k)$ or $\mathbf{enca}(f, k)$, depending on whether k is symmetric or assymmetric.

Note that while every term in the verifier-production framework has a counterpart in our framework, the converse is not true: there are terms that cannot be mapped backwards in the above translation, such as $\mathbf{fst}(f)$ where f is not a pair. We call these terms *malformed* (as opposed to *well-formed*). We observe that malformed terms are precisely those that contain one of the following:

- **fst**, **snd**, **dec** or **deca**,
- $\mathbf{enc}(\ldots, t)$, where t is not a symmetric key, or
- $\mathbf{enca}(\ldots, t)$, where t is not an asymmetric key.

Recall that the inverse-key function $\cdot^{-1}$ is defined over atomic values; therefore it can never be applied to malformed terms.

We need malformed terms in order to model such arbitrary behaviour as decrypting a ciphertext with the wrong key (for example, the wrong guess) or encryption scheme, or decomposing a non-composite term.

4.2 Programs

A program P is a sequence of instructions, where an instruction is one of:

$r_k := \mathbf{pair}(r_i, r_j)$
$r_k := \mathbf{fst}(r_i)$
$r_k := \mathbf{snd}(r_i)$
$r_k := \mathbf{enc}(r_i, r_j)$
$r_k := \mathbf{dec}(r_i, r_j)$
$r_k := \mathbf{enca}(r_i, r_j)$
$r_k := \mathbf{deca}(r_i, r_j)$
goto k
if $r_i = r_j$ **goto** k
output k

where i, j, k are natural numbers. The assignment instructions mimic terms as they apply the appropriate operations. We also have unconditional and conditional jumps, and outputs.

A program P takes an input, which is a finite sequence of well-formed terms. The input is copied into registers $r_0, \ldots, r_{n-1}$, where n is the length of the input. (Remember: the registers store terms.) All other registers are undefined, except a special integer register called the *program counter* (PC) which starts at 0. We then enter a fetch/execute loop as follows.

If there is no instruction at location PC in the program, then the program will halt. Such programs are ill-written, but we must say how they behave nonetheless.

If there is an assignment instruction $r_k := t$ at PC, then it is executed by updating the register r_k to the term t' which is formed from t by substituting names of registers with their values. For example, if registers r_2 and r_3 hold terms $\mathbf{enc}(v, k)$ and v' respectively, then execution of the instruction $r_1 := \mathbf{pair}(r_2, r_3)$ causes r_1 to subsequently hold $\mathbf{pair}(\mathbf{enc}(v, k), v')$. Immediately after this, a top-level *reduction* may take place in r_k, according to the $\rightsquigarrow$ relation, to ensure the term is in its fully reduced form. If an uninitialised register is encountered on the right-hand side of an assignment, then we consider the program to be ill-written and it halts. Finally, the PC is increased by one.

Unconditional jumps **goto** n update the PC to n. Conditional jumps **if** $r_i = r_j$ **goto** n are executed as follows: if the terms in the registers r_i and r_j are syntactically identical, then the PC is changed to n; otherwise the PC is increased by one. Note that unconditional jumps are instances of jumps with conditional $r_0 = r_0$, and will therefore not be mentioned in proofs.

An output command **output** k sends the number k to the program's output stream, and increases PC by one.

The observable behaviour of a program P with an input K is the (possibly infinite) sequence of numbers that appears on the output steam during execution. We write $P(K) \simeq P'(K')$ to mean that the output of P with input K is identical to that of P' with input K'.

4.3 An example

We present a program P that performs the verification described by the example in Section 3. Recall that the intruder is attempting to guess a symmetric key g with initial knowledge

$$K = \langle v,\ \mathbf{enc}(v, g)\rangle.$$

Therefore the program should expect input of the following form, where x is a guess of g:

$$K_x = K^\frown\langle x\rangle = \langle v,\ \mathbf{enc}(v, g),\ x\rangle,$$

and here is P itself:

0. $r_3 := \mathbf{dec}(r_1, r_2)$
1. **if** $r_0 = r_3$ **goto** 4
2. **output** 0
3. **goto** 5
4. **output** 1
5.

If we run this program with input K_g, it assigns the term v to r_3, and outputs a 1 before terminating. With input $K_{g'}$ for any $g' \neq g$ the decryption does not reduce and r_3 instead holds the term $\mathbf{dec}(\mathbf{enc}(v, g), g')$; the test in line 1 will be false and the program will output a 0.

In contrast, g is not recognisable with the following knowledge:

$$K' = \langle \mathbf{enc}(v, g)\rangle.$$

Even though a program could attempt to decrypt the term with the guess, as in P above, it subsequently has no way of telling whether the decryption succeeded or failed.

5 Completeness

In this section we demonstrate the completeness of verifier production with respect to recognisability by proving the following theorem.

Theorem 5.1. *If g is recognisable from knowledge sequence K, then g is either deducible or verifier-producing from knowledge sequence*[1] *K.*

If g is already deducible from K, the theorem is trivially satisfied. We therefore assume that g is not deducible from K throughout the proof. In particular, this means that g is not a member of K, and hence $K^\frown\langle g\rangle$ contains no repetitions. Here is an outline of the remainder of the proof.

[1] Strictly speaking, the definition of verifier-producing uses a knowledge *set*, as opposed to a sequence; we blur the distinction.

- First we reduce the problem of recognisability to a 'distinguishability' problem: here, we have a program P which distinguishes g from a *single*, *fresh* g'.
- We then present a series of transformation on P, with the aim of simplifying later parts of the proof, and prove some useful lemmas about the transformed P.
- We show that the steps made by P when run on the correct guess g produce terms that are well-formed; this allows us to write down a corresponding Dolev-Yao deduction trace.
- Finally, we show that this trace satisfies the conditions of verifier production given in Section 2.

5.1 Distinguishability

We now show how an instance of the recognisability problem can be reduced to a slightly simpler problem which we call distinguishability. Let's suppose that a guess g is recognisable by the program P with initial knowledge K. That means that for all $g' \neq g$, we have $P(K^\frown\langle g\rangle) \not\approx P(K^\frown\langle g'\rangle)$.

It suffices to consider just one such g' which has the following property: g' does not appear as a subterm in K and is not the inverse of any key appearing as a subterm in K. We say that g' is *fresh* from K. We assume that the type of g is sufficiently large for such a g' to exist. For intuition, freshness is required to ensure that P is actually recognising g as opposed to recognising g'.

We say that $K^\frown\langle g\rangle$ and $K^\frown\langle g'\rangle$ (for g' fresh) are *distinguishable* when there exists a program P such that $P(K^\frown\langle g\rangle) \not\approx P(K^\frown\langle g'\rangle)$. The following proposition summarises the above discussion.

Proposition 5.2. *If a value g is recognisable with knowledge sequence K, then for some g' fresh from K, the knowledge sequences $K^\frown\langle g\rangle$ and $K^\frown\langle g'\rangle$ are distinguishable.*

As an aside, note that our definition of distinguishability should not be confused with some stronger definitions in the literature, where the program has access to both inputs at once. For example, for our purposes the knowledge sequence $K = \langle n_1\rangle$ is not distinguishable from $K' = \langle n_2\rangle$ (where $n_1 \neq n_2$). All any program would see is an arbitrary bit sequence in both cases, and it cannot make any deductions based on that.

5.2 Normalising distinguishing programs

In this section we present a series of program transformations, which convert a program into a normal form from where it is easier to relate its behaviour to a verifier-production guessing attack trace.

The transformations: simplify the output of programs to just a binary signal; unravel programs so they contain no control structures; add extra registers so each register is assigned to at most once; and ensure that a certain form of undoing step cannot occur.

We define our normal form as the smallest such program, in an effort to comply with Condition (5) of verifier production.

We are considering the distinguishability of the two knowledge sequences $K^\frown\langle g\rangle$ and $K^\frown\langle g'\rangle$. However, for conciseness and generality, we consider distinguishability of two arbitrary non-empty knowledge sequences of equal length, without repetitions, which we call K and K'.

5.2.1 Adding signal

Consider the following new instructions:

signal $r_i = r_j$ and **signal** r_i **hasinv** r_j.

Their semantics dictate that the program outputs a 1 if the test is true, and a 0 otherwise. The **hasinv** test is true exactly when the value of r_i is an asymmetric key k and the value of r_j is k^{-1}. Afterwards, execution of the program terminates.

We now show that allowing this instruction in our programs adds no expressive power. We can henceforth equivalently consider such programs.

Proposition 5.3. *K and K' are distinguishable iff they are distinguishable by a program P which may use the* **signal** *instruction.*

Proof. $\Rightarrow$ Instantly true.

$\Leftarrow$ Replace a **signal** $r_i = r_j$ with

if $r_i = r_j$ **goto** l
output 0
goto l_∞
output 1 (This is line l)
goto l_∞.

As this sequence of instructions is longer than the single instruction it replaces, we may also require a renumbering of references to line numbers in the program in the obvious way. The value l_∞ is any line number beyond the last used line in the resultant program.

Replace a **signal** r_i **hasinv** r_j instruction at line n with

$r_i :=$ **enca**(r_j, r_i)
$r_i :=$ **deca**(r_i, r_j)
if $r_i = r_j$ **goto** l
output 0
goto l_∞
output 1 (This is line l)
goto l_∞.

This piece of code tests whether encrypting a value x with r_i and subsequently decrypting with r_j creates x again. (In fact, it uses the value in r_j as x just because we know that register exists, but any value would do.) If so, r_j must be the inverse key of r_i. Again, line reference renumbering may be required. □

5.2.2 Unravelling

In this section we show that we can remove all control structures. We define an *unravelled* program to be a program that has no **if**, **goto** or **signal** instructions, except that it has a **signal** instruction at the end.

Proposition 5.4. *K and K' are distinguishable in the sense of Section 5.1 (i.e. by a program with no* **signal** *instructions) iff they are distinguishable by an unravelled program. Furthermore, the unravelled program will eventually execute this final* **signal** *instruction, i.e. it doesn't prematurely halt due to undefined program lines or uninitialised registers.*

Proof. $\Leftarrow$ Proposition 5.3.

$\Rightarrow$ Suppose K and K' are distinguishable by the program P. Consider the sequences of line numbers that the PC goes through with inputs K and K' on P: they must be different, otherwise the same output sequences would be produced. Therefore, there exists a finite sequence l that is the longest initial subsequence of both PC-sequences.

We can show that the last element of l is the line number of an **if** instruction in P. We argue by contradiction.

- If the program contains no instruction at this point, then the program will terminate here for both inputs K and K'. Therefore l is the entire PC-sequence for P with both inputs K and K', which contradicts them having different PC-sequences.
- Suppose the executing program encounters an assignment using uninitialised registers for the input K. The program execution on input K' has so far visited all the same lines in the program, so must also encounter an uninitialised register. Both programs terminate at this point. (And vice versa.)
- In all other cases (except **if**), after executing this instruction both program executions will fetch the following line in the program.

Now form the unravelled program P' by selecting the instructions from P at the line numbers in l, but ignoring all **if** and **goto** instructions. Add a final instruction **signal** c, where c is the condition in the final **if** mentioned above.

Now, execution of P' with either K or K' will encounter all the same assignment instructions as P up to the final instruction. When executed in P the value of the conditional in the above **if** instruction was different for the two inputs, so the **signal** instruction in P' will output 0 for one input and 1 for the other input. Hence the program P' will distinguish K and K'. □

We write $P(K)$ for the mapping from registers to terms when the execution of program P with input K reaches the final instruction. We now say that an unravelled program P distinguishes K *from* K' if it is $P(K)$ that has the signal condition true (and so outputs 1).

5.2.3 Unique register assignments

The following proposition eases our proofs by allowing us to unambiguously refer to the unique value assigned to a register during the execution of a program, and the instruction that assigned to that register in the program. We define an unravelled program to have the *unique-reg* property if a previously undefined register is assigned to at every assignment instruction.

Proposition 5.5. *Two knowledge sequences K and K' are distinguishable iff they are distinguishable by an unravelled program which also has the unique-reg property.*

Proof. $\Leftarrow$ Proposition 5.3.

$\Rightarrow$ First apply Proposition 5.4 to get an unravelled program. Then, starting at line 0 in the program, if that line assigns to an input register or to a register already assigned to, then rename it and all forward occurrences to a fresh register. Repeat for every line in order. □

5.2.4 No-tail-undo

Here we present a program transformation that will later ensure Condition (5) of verifier production is met. The following example should illustrate why it is necessary.

Suppose we are studying the distinguishability of the following two sequences of knowledge:

$$\begin{array}{rcl} & & \begin{array}{cc} (r_0) & (r_1) \end{array} \\ K & = & \langle \quad \mathbf{enc}(m,k), \quad \mathbf{pair}(k,m) \quad \rangle \\ K' & = & \langle \quad \mathbf{enc}(m',k), \quad \mathbf{pair}(k,m) \quad \rangle. \end{array}$$

A program that will distinguish these sequences performs the following operations: extract the key k from r_1, decrypt r_0 with this key, and compare the plaintext with the result of extracting m from r_1:

$r_2 := \mathbf{fst}(r_1)$
$r_3 := \mathbf{dec}(r_0, r_2)$
$r_4 := \mathbf{snd}(r_1)$
signal $r_3 = r_4$.

An alternative version alters the final assignment as follows. Instead of extracting m from r_1, we pair up k with the value in r_3 and see if this is equal to r_1:

$r_2 := \mathbf{fst}(r_1)$
$r_3 := \mathbf{dec}(r_0, r_2)$
$r_4 := \mathbf{pair}(r_2, r_3)$
signal $r_1 = r_4$.

Both of these programs are optimal in terms of program length. However the second program will eventually yield a sequence of deductions that does not satisfy Condition (5). Notice how that last deduction undoes the first:

$$\langle \quad \mathbf{pair}(k,m) \vdash_{\mathsf{fst}} k, \quad \{|m|\}_k, k \vdash_{\mathsf{dec}} m, \quad k, m \vdash_{\mathsf{pair}} \mathbf{pair}(k,m) \quad \rangle.$$

We conclude that choosing the *smallest* normalised program is not always enough. What is required is a transformation to turn programs like the second one above into programs like the first. We begin by defining the programs of interest.

An unravelled unique-reg program that distinguishes K from K' satisfies the *tail-undo* property if it has one of the following shapes (for some i, j, k, q):

$$\begin{array}{l} \vdots \\ r_i := \mathbf{fst}(r_k) \\ \vdots \\ r_q := \mathbf{pair}(r_i, r_j) \\ \mathbf{signal}\ r_k = r_q \end{array} \quad \text{or} \quad \begin{array}{l} \vdots \\ r_j := \mathbf{snd}(r_k) \\ \vdots \\ r_q := \mathbf{pair}(r_i, r_j) \\ \mathbf{signal}\ r_k = r_q \end{array}$$

and also $P(K)$ has $r_i = t_i,\ r_j = t_j$ and $r_k = r_q = \mathbf{pair}(t_i, t_j)$ for some t_i, t_j. For example, the second program above has this property. An unravelled unique-reg program satisfies the *no-tail-undo* property if it *doesn't* satisfy the tail-undo property.

Proposition 5.6. *If two knowledge sequences K and K' are distinguishable, then they are distinguishable by an unravelled, unique-reg program which also satisfies the no-tail-undo property.*

Proof. $\Leftarrow$ Proposition 5.3.

$\Rightarrow$ Proposition 5.5 gives us the unravelled, unique-reg program that, without loss of generality, distinguishes K from K'. If the program satisfies the no-tail-undo property then we're done, so we suppose it satisfies the tail-undo property. We do only the case for **fst** as the case for **snd** runs symmetrically.

Form the program P' from P by replacing the last two instructions with:

$$\begin{array}{l} r'_q := \mathbf{snd}(r_k) \\ \mathbf{signal}\ r_j = r'_q \end{array}$$

where r'_q is a fresh register. (We keep the assignment to r_i, because other commands in the program might use the result.) This program satisfies no-tail-undo, whilst still satisfying the unravelled and unique-reg properties.

In $P'(K)$, we know that $r'_q = r_j = t_j$, so the program $P'(K)$ has its **signal** condition true. It remains to show that $P'(K')$ has its signal condition false. Note that the values of the registers r_i, r_j and r_k are the same in $P(K')$ and $P'(K')$. We case split on t_k, the value of r_k.

- Suppose t_k is not of the form $\mathbf{pair}(\cdot, \cdot)$. If the signal condition in $P'(K')$ were to be true, we would need $r_j = r'_q = \mathbf{snd}(t_k)$. This means that $r_j = \mathbf{snd}(t_k)$ in $P(K')$ too. This is a contradiction, because $P(K')$ also has $r_i = \mathbf{fst}(t_k)$, so the condition $r_k = r_q$ would be true.

- Suppose t_k is of the form $\mathbf{pair}(s_1, s_2)$. If the signal condition in $P'(K')$ were to be true, we would need $r_j = s_2$. The means that $r_j = s_2$ in $P(K')$ too. This is a contradiction, because $P(K')$ also has $r_i = s_1$, so the condition $r_k = r_q$ would be true. □

5.2.5 Smallest Normalised Programs (SNP's)

We define a *smallest normalised program (SNP)* that distinguishes K from K' as a program with minimal instructions with the following properties:

- it distinguishes K from K' (so the **signal** condition is true for K);
- it is unravelled, i.e. contains no **if** or **goto** instructions, and a **signal** instruction only at the end;
- it assigns to a fresh register at each assignment (the unique-reg property);
- it satisfies the no-tail-undo property.

The following theorem follows immediately from Proposition 5.6.

Theorem 5.7. *If K and K' are distinguishable then they are distinguishable by a Smallest Normalised Program (SNP).*

5.3 Lemmas about SNPs

Here we prove some lemmas about SNPs that will be useful later when comparing recognisability and verifier production. We assume the existence of an SNP P that distinguishes the knowledge sequence K from K'.

Lemma 5.8. *In $P(K)$, all registers have distinct values except (possibly) the register assigned to in the last assignment.*

Proof. By assumption, the input registers of $P(K)$ are distinct. Let r_i and r_j be registers with the same values in $P(K)$, such that (without loss of generality) r_j is assigned to after r_i receives its value (so r_i is either an input register, or is assigned a value before r_j is assigned a value) and r_j is not the register of the last assignment. If they have equal values in $P(K')$ too, we can remove the production of r_j and rename all occurrences of r_j to r_i; this creates a smaller SNP. So they must have different values in $P(K')$. By assumption, r_j is not produced in the last assignment, so we can create a smaller SNP by replacing the last assignment with **signal** $r_i = r_j$, and removing the previous signal instruction. □

Lemma 5.9. *We can't have two instructions in an SNP with identical right-hand sides.*

Proof. Suppose we do. Then we create the same value in the two registers r_i and r_j, in both $P(K)$ and in $P(K')$; we can therefore remove the later assignment, to r_j say, and replace references to r_j with r_i. This gives a smaller SNP. □

Lemma 5.10. *The program P does not perform any 'undoing.' Specifically, it does not contain all of the instructions in any of the following sets (with any instantiation of the register names).*

- $r_k := \mathbf{pair}(r_i, r_j)$, $r_l := \mathbf{fst}(r_k)$.

- $r_k := \mathbf{pair}(r_j, r_i)$, $r_l := \mathbf{snd}(r_k)$.
- $r_j := \mathbf{fst}(r_i)$, $r_k := \mathbf{snd}(r_i)$, $r_l := \mathbf{pair}(r_j, r_k)$.
- $r_j := \mathbf{enc}(r_i, r_k)$, $r_l := \mathbf{dec}(r_j, r_k)$.
- $r_j := \mathbf{dec}(r_i, r_k)$, $r_l := \mathbf{enc}(r_j, r_k)$.
- $r_j := \mathbf{enca}(r_i, r_n)$, $r_l := \mathbf{deca}(r_j, r_m)$, *where* $P(K)$ *has* $r_n = k$ *and* $r_m = k^{-1}$ *for some asymmetric key* k.
- $r_j := \mathbf{deca}(r_i, r_m)$, $r_l := \mathbf{enca}(r_j, r_n)$, *where* $P(K)$ *has* $r_n = k$ *and* $r_m = k^{-1}$ *for some asymmetric key* k.

Proof. For all cases except the last two, it is clear from the semantics of instructions that $r_l = r_i$ in both $P(K)$ and $P(K')$. Remove the assignment to r_l, and replace any references to r_l with r_i. The program still distinguishes K and K' and is shorter than P, contradicting the minimality of P.

For the last two cases, suppose $P(K')$ does not have r_m as the inverse key of r_n. Then, end the program at the r_l assignment with the instruction **signal** r_k **hasinv** r_m to create a smaller SNP — a contradiction. Otherwise, we conclude that the encryption and decryption cancel out in both $P(K)$ and $P(K')$, and proceed as before. □

5.4 Well-formedness of deductions in SNPs

We show that a program P that distinguishes knowledge sequence K from K' never produces any malformed terms in its registers on the input K. This will allow us to take the execution of $P(K)$ and write down a well-formed Dolev-Yao trace.

Recall, from Section 4.1 that a term is malformed if it contains **fst**, **snd**, **dec** or **deca**, or $\mathbf{enc}(\ldots, t)$ or $\mathbf{enca}(\ldots, t)$ where t is not an appropriate key.

We say that a term is *unreduced* as follows: if the term has shape $\mathbf{pair}(\ldots)$ then it was created with an instruction of the form $r_i := \mathbf{pair}(\ldots)$ with no reduction applying; and similarly for the other term constructors **fst**, **snd**, **enc**, **dec**, **enca**, **deca**. Note that non-reduced terms cannot be atoms.

Proposition 5.11. *If a malformed term appears in a register* r_i *in* $P(K)$, *then it is unreduced.*

Proof. In this proof, all values of registers referred to are values in $P(K)$.

We prove the proposition by induction over the number of registers used in initial segments of the program. For the empty initial segment of P, we only need to consider input registers. These are not malformed so there is nothing to prove.

We now suppose the induction hypothesis for all registers appearing in some initial segment of the program: registers that are malformed are unreduced. We encounter the next instruction in the program that produces a malformed term in a register r_i. To re-establish the induction hypothesis we need to show that no reduction occurs in r_i for that assignment. For each instruction we assume that a reduction does apply and establish a contradiction.

- $r_i := \mathbf{pair}(r_j, r_k)$. For a reduction to apply, we must have $r_j = \mathbf{fst}(t)$ and $r_k = \mathbf{snd}(t)$ for some t. These are malformed, so by induction are unreduced and must have been created by instructions $r_j := \mathbf{fst}(r_p)$ and $r_k := \mathbf{snd}(r_q)$, where r_p and r_q both have value t. By Lemma 5.8, registers cannot have the same value like this, so $p = q$. We now have a pattern of instructions

  $$r_j := \mathbf{fst}(r_p) \quad \ldots \quad r_k := \mathbf{snd}(r_p) \quad \ldots \quad r_i := \mathbf{pair}(r_j, r_k).$$

  in the program P. This contradicts the 'no undoing' property proved in Lemma 5.10.

- $r_i := \mathbf{fst}(r_j)$. If a reduction were to apply, we would have $r_j = \mathbf{pair}(t, t')$. Recall that we are assuming that r_i is malformed, in which case t must be malformed. Therefore r_j is malformed and therefore unreduced, so there must be an instruction $r_j := \mathbf{pair}(r_p, r_q)$. This constitutes 'undoing' as prohibited by Lemma 5.10.

- $r_i := \mathbf{snd}(r_j)$. Analogous to above.

- $r_i := \mathbf{dec}(r_j, r_k)$. If a reduction is to take place, we must have $r_j = \mathbf{enc}(t_1, t_2)$ and $r_k = t_2$. We conclude that $r_i = t_1$. Recall again our assumption that r_i is malformed, so t_1 is malformed. As r_j contains t_1 as a subterm, it is also malformed. The induction hypothesis implies the existence of an instruction $r_j := \mathbf{enc}(r_p, r_q)$ with $r_p = t_1$ and $r_q = t_2$. By Lemma 5.8 we have $q = k$ which contradicts Lemma 5.10.

- $r_i := \mathbf{enc}(r_j, r_k)$. As above, swapping **dec** and **enc**.

- $r_i := \mathbf{deca}(r_j, r_k)$. We are supposing that a reduction takes place, so must have $r_j = \mathbf{enca}(t_1, t_2)$ and $r_k = {t_2}^{-1}$. Hence $r_i = t_1$. Remember that r_i is malformed by assumption, so r_j is malformed. By induction, we have an instruction $r_j := \mathbf{enca}(r_p, r_q)$ with $r_q = t_2$. This is not allowed by Lemma 5.10.

- $r_i := \mathbf{enca}(r_j, r_k)$. As above, swapping **deca** and **enca**, and also t_2 and ${t_2}^{-1}$. □

Proposition 5.12. *In $P(K)$, malformedness is* hereditary *in the following sense: a register assigned to by an instruction using a malformed register is itself malformed.*

Proof. Consider an assignment instruction $r_i := \ldots$ where one of the registers used on the right-hand side is malformed. If a reduction doesn't occur on r_i, then the proposition is trivially true as the new term will contain the malformed one as a substring. So, for each instruction we assume a reduction does apply and reach a contradiction.

- $r_i := \mathbf{pair}(r_j, r_k)$. If a reduction occurs, then $r_j = \mathbf{fst}(t)$ and $r_k = \mathbf{snd}(t)$ for some t. By Proposition 5.11 we know these register are unreduced, so there must be previous instructions $r_j := \mathbf{fst}(r_p)$ and $r_k := \mathbf{snd}(r_q)$ where $r_p = t$ and $r_q = t$. Apply Lemma 5.8 to find that $p = q$, but this means the program is performs an 'undoing' operation, which contradicts Lemma 5.10.

- $r_i := \mathbf{fst}(r_j)$. As a reduction occurs, we must have $r_j = \mathbf{pair}(t, t')$. By assumption, r_j is malformed. Proposition 5.11 tell us r_j was created by an instruction $r_j := \mathbf{pair}(r_p, r_q)$, contravening Lemma 5.10.

- $r_i := \mathbf{snd}(r_j)$. As above.

- $r_i := \mathbf{dec}(r_j, r_k)$. As a reduction occurs, we must have $r_j = \mathbf{enc}(t_1, t_2)$ and $r_k = t_2$. By semantics of **dec**, the value of r_i will be t_1. By assumption at least one of r_j and r_k is malformed: this must be because at least one of t_1 and t_2 is malformed. Therefore, in either case, r_j is malformed. By Proposition 5.11, r_j is unreduced and there is some instruction $r_j := \mathbf{enc}(r_p, r_q)$ where $r_p = t_1$ and $r_q = t_2$. By Lemma 5.8 we have $k = q$, contradicting Lemma 5.10.

- $r_i := \mathbf{enc}(r_j, r_k)$. Analogous to the case above.

- $r_i := \mathbf{deca}(r_j, r_k)$. If a reduction occurs, we must have $r_j = \mathbf{enca}(t_1, t_2)$ and $r_k = {t_2}^{-1}$. We know that at least one of r_j and r_k is malformed, but r_k is just a key ${t_2}^{-1}$, so r_j must be malformed. But t_2 is just a key, so t_1 must be malformed. By semantics of **deca**, $r_i = t_1$, and so r_i is malformed.

- $r_i := \mathbf{enca}(r_j, r_k)$. Analogous to the case above. □

Theorem 5.13. *$P(K)$ does not contain any malformed terms.*

Proof. Once again, values of registers referred to in the proof are their values in $P(K)$.

Suppose there is a malformed term in a register. As the input is well-formed, this must be a register created in an assignment. All assignment instructions must 'contribute' to the condition in the signal instruction, else they are redundant and can be removed. We can then apply the 'hereditary of malformedness' property (Proposition 5.12) to deduce that one of the two registers used in the signal instruction must be malformed.

Now consider the final signal instruction itself. A r_i **hasinv** r_j condition could never be true if one of r_i and r_j is malformed, so the condition must be of the form $r_i = r_j$. By Proposition 5.11, we know that r_i and r_j were created in the same way. For example, if they are both pairs then they were created with instructions $r_i := \mathbf{pair}(r_p, r_q)$ and $r_j := \mathbf{pair}(r_{p'}, r_{q'})$, such that no reduction applies for these instructions in $P(K)$. So, in order that $r_i = r_j$, we must have had $r_p = r_{p'}$ and $r_q = r_{q'}$. By Lemma 5.8, we must have $p = p'$ and $q = q'$, which contradicts Lemma 5.9. Cases for other instructions run similarly. □

5.5 Forming Dolev-Yao deduction traces from SNPs

The above theorem tells us that we can form a *corresponding D-Y trace* from $P(K)$, where P is an SNP which distinguishes the knowledge sequence K from K'.

For each instruction we form a deduction in the natural way. For example, from $r_k := \mathbf{pair}(r_i, r_j)$ we get $f, f' \vdash_{\mathsf{pair}} (f, f')$, where registers r_i, r_j and r_k have the values f, f' and $\mathbf{pair}(f, f')$ respectively. Deductions for other instructions are produced analogously.

Theorem 5.13 guarantees well-formed Dolev-Yao deductions in each case, resulting in a valid D-Y trace.

Now that this translation is well defined, we will use it implicitly in proofs.

5.6 SNP deductions respect undoes

Carrying on from the last section, let tr be the corresponding D-Y trace of $P(K)$. We now show that no deduction in tr undoes any other deduction. First we prove a small lemma.

Lemma 5.14. *For any i, j, k, q, if $P(K)$ has $r_i = t$, $r_j = t'$, and $r_q = r_k = \mathbf{pair}(t, t')$ then we cannot have both $r_i := \mathbf{fst}(r_k)$ and $r_q := \mathbf{pair}(r_i, r_j)$ in P; neither can we have both $r_j := \mathbf{snd}(r_k)$ and $r_q := \mathbf{pair}(r_i, r_j)$ in P.*

Proof. We do only the case for **fst**, as the case for **snd** runs symmetrically. Suppose we have $r_i := \mathbf{fst}(r_k)$ and $r_q := \mathbf{pair}(r_i, r_j)$, where registers have values as given in the lemma statement. Lemma 5.8 then tells us that the assignment to r_q is the final assignment. The **signal** condition must use r_q, or else this final assignment is redundant and could be removed. Proposition 5.6 then tells us that the **signal** condition is not of the form $r_k = r_q$, so it must be of the form $r_p = r_q$, for some other register r_p equal to both r_k and r_q. But now we have two registers, neither assigned to in the final assignment, with equal values, contradicting Lemma 5.8. □

Theorem 5.15. *For all pairs of deduction $S \vdash_l v$ and $S' \vdash_{l'} v'$ in tr, we do not have $S \vdash_l v$ undoes $S' \vdash_{l'} v'$.*

Proof. We suppose, for a contradiction, $S \vdash_l v$ undoes $S' \vdash_{l'} v'$, for each case of undoes in turn. All register values are in $P(K)$.

- Suppose $\{f, f'\} \vdash_{\mathsf{pair}} (f, f')$ and $\{(f, f')\} \vdash_{\mathsf{fst}} f$ both in tr. This means we have $r_p := \mathbf{pair}(r_i, r_j)$ and $r_q := \mathbf{fst}(r_k)$ in P, with $r_p = r_k = \mathbf{pair}(f, f')$, $r_i = r_q = f$, and $r_j = f'$. We now case split depending on which instruction appears first:
  - Suppose the assignment to r_p appears before that to r_q in P. Then Lemma 5.8 demands that $p = k$. So we have $r_k := \mathbf{pair}(r_i, r_j)$ and $r_q := \mathbf{fst}(r_k)$ in P. This contradicts Lemma 5.10.
  - If the assignment to r_q appears first, then $i = q$. We have $r_i := \mathbf{fst}(r_k)$ and $r_p := \mathbf{pair}(r_i, r_j)$ in P. This contradicts Lemma 5.14.
- Case for **pair**/**snd** runs analogously.
- Suppose $\{f, k\} \vdash_{\mathsf{enc}} \{|f|\}_k$ and $\{\{|f|\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f$ both in tr, where k is a symmetric key. This means we have $r_p := \mathbf{enc}(r_i, r_j)$ and $r_q := \mathbf{dec}(r_m, r_n)$ in P, with $r_p = r_m = \mathbf{enc}(f, k)$, $r_i = r_q = f$, and $r_j = r_n = k$. If the assignment to r_p appears first, then Lemma 5.8 means $p = m$ and $j = n$, so we contradict Lemma 5.10. If the assignment to r_q appears first, then Lemma 5.8 means $q = i$ and $j = n$, also contradicting Lemma 5.10.

- Suppose $\{f, k\} \vdash_{\mathsf{enc}} \{\!|f|\!\}_k$ and $\{\{\!|f|\!\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f$ both in tr, where k is an asymmetric key. This case is analogous to the previous. □

5.7 SNP deductions recognise the correct guess

Suppose value g is recognisable from knowledge K. Recall from Proposition 5.2 that this means there is some fresh g' such that $K_g = K^\frown\langle g\rangle$ and $K_{g'} = K^\frown\langle g'\rangle$ are distinguishable. By Theorem 5.7, we can deduce that they are distinguished by an SNP P. In this section we show that P distinguishes g from g', as opposed to vice-versa: the signal condition is true for input K_g and false for $K_{g'}$.

In this section we assume, for a contradiction, that the signal condition is true for $P(K_{g'})$. Under this assumption, the lemmas of Section 5.3 become applicable to $K_{g'}$, and this allows us to reuse the proofs from Section 5.4. Where values of registers are mentioned we mean their values in $P(K_{g'})$ unless stated otherwise.

Proposition 5.16. *Under the assumption that the signal condition in $P(K_{g'})$ is true, if an assignment causes a register to contain g' as a subterm, then that register is unreduced. Recall that unreduced means it was created using the assignment instruction corresponding to the term's outer constructor with no reduction applying. In particular, this implies that g' is a* strict *subterm of the register.*

Proof. The proof runs very much like that of Proposition 5.11. For example, consider the case $r_i := \mathbf{fst}(r_j)$. If a reduction were to apply, we would have $r_j = \mathbf{pair}(t, t')$. Recall that we are assuming that r_i contains g', in which case t must contain g'. Therefore r_j contains g', and is therefore unreduced, so there must be an instruction $r_j := \mathbf{pair}(r_p, r_q)$. This constitutes 'undoing' as prohibited by Lemma 5.10. □

Proposition 5.17. *Under the assumption that the signal condition in $P(K_{g'})$ is true, each register with g' as a subterm is such that:*

1. *the register has value g' and it is the last input register; or*
2. *the register contains g' as a strict subterm and it is unreduced.*

Proof. The base case follows from the fact that we are considering $P(K_{g'})$, and g' is fresh from K (Proposition 5.2). This result then follows easily from the previous proposition. □

Proposition 5.18. *Under the assumption that the signal condition in $P(K_{g'})$ is true, the property of containing g' as a subterm is hereditary in the following sense: a register assigned to by an instruction using a register with g' as a subterm will also contain g' as a subterm.*

Proof. The proof is very similar to that of Proposition 5.12. □

The following proposition essentially tells us that P is actually recognising g rather than g'.

Proposition 5.19. *The program P distinguishes K_g from $K_{g'}$: the signal condition in P is true for input K_g and false for input $K_{g'}$.*

Proof. We suppose for a contradiction that the signal condition is true for $K_{g'}$. This makes the above propositions applicable.

The register holding g' must 'contribute' to the condition in the signal instruction, otherwise P could not distinguish K_g and $K_{g'}$. Hence, applying Proposition 5.18, we see that g' must be a subterm of one of the registers used in the signal instruction in $P(K_{g'})$.

This signal condition can't be r_i **hasinv** r_j. We know one of these registers must contain g', and to be a key it would therefore have to actually be g'. In order that this **hasinv** condition is true, the other register must be g'^{-1}; this is impossible because g' is fresh in K, so g'^{-1} doesn't appear anywhere in the input $K_{g'}$.

Hence the signal condition must be of the form $r_i = r_j$. It might be the case that these registers both have the value g'; if so, Proposition 5.17 says we must have $i = j$ and $P(K_g)$ could never be false. Otherwise, we know that both r_i and r_j are unreduced, and we end up with the same argument as in the last paragraph of the proof of Theorem 5.13. □

As an aside, we note that this proposition validates our comment about the generality of our definition of recognisability made back in Section 3. We state this as a theorem.

Theorem 5.20. *A value g is recognisable with knowledge K if and only if it is recognisable by a program that outputs $\langle 1 \rangle$ for g, and $\langle 0 \rangle$ for any $g' \neq g$ (i.e. any wrong guess).*

5.8 SNP deductions are verifier producing

We are finally ready to prove that recognisability implies verifier production. We repeat our assumptions from the start of Section 5:

1. Some value g is recognisable from knowledge K.
2. The value g is not D-Y deducible from K, i.e. there's no K' such that $K \models K' \ni g$.

It remains to show that g is verifier producing in order to prove Theorem 5.1.

Recall from Proposition 5.19 that this means that there is an SNP P and some fresh g' such that the signal condition in P gives true for $K_g = K^\frown\langle g \rangle$, and false for $K_{g'} = K^\frown\langle g' \rangle$. So by Theorem 5.13 we can get the corresponding D-Y trace of $P(K_g)$, which we denote T,

Ideally we would use T directly as the guess attack trace tr in the definition of verifier producing, but unfortunately this doesn't always work due to the exact statement of Condition (3). To ease discussion, we reproduce some of that definition here:

$$S \vdash_l v \ \in \ tr. \tag{2}$$

$$\nexists IK'' \cdot (IK \models IK'' \supseteq S). \tag{3}$$

Condition (3) states that S, the set of facts from which v is deduced, is not itself deducible without g. However, consider the following initial knowledge IK

$$\langle\ v,\ \{|v, x|\}_g,\ (((v, x), y), z)\ \rangle,$$

and suppose we wish to show that the symmetric key g is guessable. The trace T produced by an SNP will be:

$$\begin{array}{lrll} \langle & \{\{\!|v,x|\!\}_g, g\} & \vdash_{\mathsf{dec}} & (v,x), \\ & \{(v,x)\} & \vdash_{\mathsf{fst}} & v \qquad \rangle. \end{array}$$

In this trace, v acts as the verifier because it is also contained in the initial knowledge (i.e. Condition (4b) applies). However, we are forced to set $S = \{(v,x)\}$, which does not satisfy Condition (3).

In such a case we need to extract the initial portion of T that produces the first already-known term, and make this the verifier. We can then deduce this verifier in a way that doesn't require the guess g. In our example we end up with a trace like this:

$$\begin{array}{lrll} \langle & \{\{\!|v,x|\!\}_g, g\} & \vdash_{\mathsf{dec}} & (v,x), \\ & \{(((v,x),y),z)\} & \vdash_{\mathsf{fst}} & ((v,x),y), \\ & \{((v,x),y)\} & \vdash_{\mathsf{fst}} & (v,x) \qquad \rangle. \end{array}$$

In this instance, verifier production is satisfied when (v,x) acts as the verifier, and $S = \{\{\!|v,x|\!\}_g, g\}$. Note that a normalised distinguishing program could produce this trace, but we deliberately chose a *smallest* normalised program in order to satisfy Condition (5). This example shows that attacks produced by verifier production are not necessarily optimal.

In the rest of this section we formalise this procedure and show that the resulting trace is a suitable witness for g being verifier producing.

Define a relation $f \rightsquigarrow f'$ iff there exists a deduction $S \vdash_l f'$ in T with $f \in S$.

Lemma 5.21. *There exists a sequence*

$$f_0 \rightsquigarrow f_1 \rightsquigarrow f_2 \rightsquigarrow \cdots \rightsquigarrow f_n,$$

for some $n \geq 0$, such that $f_0 = g$ and f_n is the value of one of the registers in the signal instruction of P.

Proof. Apart from the final input register, all the input registers are the same in K_g and $K_{g'}$. If the program P exhibits different behaviours on these inputs then it must use this register at some point. Now we can take a maximal chain $g \rightsquigarrow f_1 \rightsquigarrow \cdots f_n$ for some $n \geq 0$, and suppose f_n is stored in register r_i.

As this chain is maximal, we know that r_i does not appear in any subsequent assignment. If it doesn't appear in the signal instruction either, then r_i is never used at all, and its production can be removed from the program without affecting the program's behaviour, thus creating a smaller SNP. □

Proposition 5.22. *The guess g is verifier-producing from initial knowledge sequence K.*

Proof. Let IK be K converted from a sequence to a set. Take a chain of facts from Lemma 5.21, choosing one with length greater than 1 if possible. We perform a case analysis.

Case 1: There is some f_i that is D-Y deducible from IK, i.e. for some tr_1 and IK_1, we have $IK \models_{tr_1} IK_1$ and $f_i \in IK_1$.

Pick the lowest such i, and let $v = f_i$. We know $i > 0$ as by assumption g is not D-Y deducible, so f_{i-1} is not D-Y deducible. We conclude that there is a deduction $S \vdash_l v$ in T such that:

- $\nexists IK'' \cdot (IK \models IK'' \supseteq S)$,
- $IK \cup \{g\} \models_T IK_2$ with $S \vdash_l v$ in T (for some IK_2).

By the above construction, we have satisfied conditions (1), (2), and (3) in the definition of verifier production with $tr = tr_1 ^\frown T$ and $IK' = IK_1 \cup IK_2$.

We show that Condition (4) is true. If $v \in IK \cup \{g\}$ then (4b) holds. So suppose $v \notin IK \cup \{g\}$, and take $S' \vdash_{l'} v$ to be the deduction in tr_1 that produces v. We have $(S, l) \neq (S', l')$ because S is not D-Y deducible from IK whereas S' is. Hence (4a) holds.

We are left with Condition (5).

- By assumption, tr_1 stops at the first production of v. This means that v never appears on the left-hand side of a deduction rule in tr_1. Therefore there's no deduction in tr_1 that $\mathsf{undoes}\ S \vdash_l v$ or $S' \vdash_{l'} v$.
- There cannot be a deduction in T that $\mathsf{undoes}\ S \vdash_l v$ by Theorem 5.15.
- Finally, suppose there is a deduction in T that $\mathsf{undoes}\ S' \vdash_{l'} v$. This deduction must produce something s in S'. This deduction can therefore be removed as s is already deduced earlier in the trace. It is important to realise that removing this deduction doesn't invalidate anything we've already shown: in particular, this deduction can not be $S \vdash_l v$ because s is used to *deduce* v so can't *be* v.

Case 2: There is a chain from Lemma 5.21 of length greater than 1, but none of the f_i is D-Y deducible from IK.

Let $v = f_n$, $tr = T$, and let IK' be such that $IK \cup \{g\} \models_T IK'$. Then there is a deduction in tr that produced v of the form $S \vdash_l v$. Note that this fulfils conditions (1), (2), and (3). Now case split on the final instruction in P:

- Case **signal** $r_i = r_j$. From Lemma 5.21 we know that one of these registers, say r_i, has value v in $P(K_g)$; therefore so does r_j. If r_j is an input register, then $v \in IK \cup \{g\}$, and we have fulfilled Condition (4b). If r_j is not an input register then r_j is produced in a D-Y deduction $S' \vdash_{l'} v$ in tr. To satisfy (4a) we now show $(S, l) \neq (S', l')$ by supposing for a contradiction that $l = l' = \mathsf{pair}$ (other cases run analogously). Then P must contain instructions $r_i := \mathbf{pair}(r_p, r_q)$ and $r_j := \mathbf{pair}(r_{p'}, r_{q'})$, where the values of r_p and $r_{p'}$ are identical, similarly r_q and $r_{q'}$. By Lemma 5.8, we must have $p = p'$ and $q = q'$. But this contradicts Lemma 5.9.
- Case **signal** r_i **hasinv** r_j. Register r_i, which has value v in $P(K_g)$, must be a key, and r_j must hold the inverse v^{-1}. Also, v is an asymmetric key from the semantics of **hasinv**, thus satisfying Condition (4c).

Finally we need to satisfy Condition (5). This follows immediately from Theorem 5.15.

Case 3: There is no chain of facts from Lemma 5.21 with length greater than 1. This means that the input register holding g or g' is used directly in the **signal** instruction. Call this register r_i, and the other register r_j. Note that r_i is not used in the production of r_j without breaking the assumption about chain length. So we have $IK \models IK' \ni r_j$ for some IK'. We now split cases depending on the type of **signal** instruction in P.

- Case **signal** $r_i = r_j$. In $P(K_g)$, for this condition to be true we must have $r_j = g$ also. This would mean that $IK \models g$, breaking one of the main assumptions of this section: that g is not already D-Y deducible from IK.
- Case **signal** r_i **hasinv** r_j. This means that $IK \models IK' \ni g^{-1}$ and $g \in ASYMMETRIC\_KEYS$. This does not quite satisfy Condition (6) of verifier production which asks only that $g^{-1} \in IK$. We assume the stronger version of Condition (6) stated below. □

The very last part of this proof reveals a deficiency in the definition of verifier production given in [Low04]. Condition (6) was designed to capture the possibility that an intruder could use g as the verifier without performing any deductions because he already knows g^{-1}. However, it doesn't allow for the fact that he may not directly know g^{-1} because it is deducible from the initial knowledge without requiring g.

For example, consider an initial knowledge set $\{\{|g^{-1}|\}_k, k\}$ where k is a symmetric key, and suppose that the intruder has guessed g correctly and wishes to verify it. Intuitively, he can extract g^{-1} from the initial knowledge using the D-Y trace:

$$\langle \quad \{\{|g^{-1}|\}_k, k\} \vdash_{\mathsf{dec}} g^{-1} \quad \rangle$$

This doesn't satisfy Condition (6) because g^{-1} is not in the initial knowledge. It doesn't satisfy Conditions (1)–(5) either, because the deduction that produces the verifier in Condition (2) would have to be the one deduction in the trace, so $S = \{\{|g^{-1}|\}_k, k\}$; but S doesn't satisfy Condition (3), that S is not deducible without g.

A weakening of Condition (6) is required. We rewrite it as:

$$g \in ASYMMETRIC\_KEYS \wedge \exists IK' \cdot IK \models IK' \ni g^{-1}. \tag{6}$$

In [Low04] an implicit assumption was made that the set IK is already closed with respect to the deduction operators. Under this assumption, the two versions of Condition (6) are identical. This assumption is not explicitly stated in [Low04] although it is enforced in the FDR implementation described there.

We have chosen to fix this problem here by changing the definition rather than adding this assumption, because it is interesting to note that dropping this quite strong assumption about IK requires a relatively minor change to the definition.

6 Soundness

In this section we prove the soundness of verifier production with respect to recognisability:

Theorem 6.1. *If g is deducible or verifier producing from knowledge sequence K, then g is recognisable from K.*

The proof is broken into the following two propositions.

Proposition 6.2. *If g is deducible from knowledge sequence K, then g is recognisable from K.*

Proof. If $K \models K' \ni g$ then there exists a sequence of D-Y deductions that produces g using K. We use these deductions to construct a program by converting each deduction in turn into an instruction. Simultaneously we build up a mapping from terms to registers, starting with the mapping $F(t) = r_i$ for all t in K, where i is t's position in K.

For example, for a deduction $\{f_1, f_2\} \vdash_{\mathsf{pair}} (f_1, f_2)$, we add the instruction $r_i :=$ **pair**$(F(f_1), F(f_2))$, where r_i is a register unused so far. Now add $F((f_1, f_2)) = r_i$ to the mapping F. Other types of deductions run analogously.

Eventually we have a program P such that $P(K)$ computes g, say in register r_i; if $g \in K$, then this program is empty and i is the position of g in K. Now add the instruction **signal** $r_i = r_j$, where j is the length of K (and hence the offset of g and g' in K_g and $K_{g'}$ respectively), and we have a program which distinguishes K_g and $K_{g'}$ for all g'. We can use Proposition 5.3 to remove the signal instruction. □

Proposition 6.3. *If g is verifier producing from knowledge sequence K, then g is recognisable from K.*

Proof. First we deal with the case that Condition (6) of verifier production is true. We use our updated version of this condition discussed at the end of Section 5. Construct the (possibly empty) program P using the D-Y trace tr from $K \models_{tr} K' \ni g^{-1}$, in the same way as the proof of Proposition 6.2, to calculate g^{-1} in r_j. Then add on the instruction **signal** r_i **hasinv** r_j, where r_i contains the inputted guess. This condition will be true when the guess is g, and false for any other value.

We now deal with Conditions (1)–(5). Take a verifier-producing trace tr of minimal length; let other values be as mentioned in the definition of Conditions (1)–(5). Construct a program P to calculate v using tr in the same way as the proof of Proposition 6.2. We then add onto the program a **signal** instruction depending on which part of Condition (4) is true. Proposition 5.3 can then be used to remove the **signal** instruction.

(4a) We compare the two registers which hold the different derivations of v.

(4b) Compare the register containing v with the register holding the correct piece of initial knowledge.

(4c) Use the **signal** ... **hasinv** ... instruction to test the registers holding v and v^{-1}.

It is clear that the program $P(K_g)$ ends up with the equality test being true. What is less clear is that $P(K_{g'})$ ends up being false for any datum $g' \neq g$. The rest of this proof focuses on this.

Not surprisingly, this proof has parallels with that of Proposition 5.19. The main difference is that P is not necessarily an SNP — instead it was created from the verifier-producing trace tr. This means that we no longer have access to the Lemmas from Section 5.3 and must recreate them for P.

Firstly, we show that all the registers in $P(K)$ have distinct values, except perhaps the value produced in the last assignment (cf. Lemma 5.8). From the assumption that tr is of minimum length, it is clear that the final deduction in tr produces v, or else that deduction is unnecessary. It is also clear that tr will contain at most one other production of v: one to satisfy Condition (2) and one to satisfy Condition (4a) if it is needed. Any extra productions would causes redundancy in tr. Aside from the term v, no two deductions in tr produce the same value, or else the latter such deduction is unnecessary. The result then follows from the construction of P.

Next, we show that P doesn't contain two assignments with identical right-hand sides (cf. Lemma 5.9). This would correspond to two identical deductions in the trace tr. One of these could be easily removed without falsifying any of conditions (1)–(5).

Further, P does not contain instructions that 'undo' each other (cf. Lemma 5.10). Condition (5) bans the corresponding patterns of deductions from appearing in tr.

We can now proceed exactly as in the proof of Proposition 5.19. □

7 Conclusions

Summary. In this paper we have presented a new, natural way of capturing off-line guess verification. Central to our definition is the existence a computational process that can somehow recognise the guess, thereby performing the verification. This is in contrast to previous verifier-production definitions which detect behaviour assumed, by inspection of known attacks, to be characteristic of a guess verification.

We show that a previous formalisation of guess verification via verifier production [Low04] is equivalent to our recognisability definition. Aside from resolving an ambiguity in this previous definition, the contributions of this can be seen in two ways: it gives justification for the verifier-production definitions of guessing attacks; and it provides a decision procedure for our more natural definition.

Future work. This paper is not complete in its comparison with Lowe's original work. The guessing definition in [Low04] is parameterised by a given set of Dolev-Yao style deductions, whereas we have assumed a standard fixed set of such deductions.

While it appears easy in most cases to modify the proofs in this paper to deal with different deduction sets, proving the more general result seems much harder: that a value is verifier-producing with some given Dolev-Yao deductions iff it is recognisable by programs with access to instructions corresponding to the given Dolev-Yao deductions.

Abadi and Gordon define secrecy in the Spi Calculus [AG99] as follows: a value x is secret if a protocol run using x is testing-equivalent to a run using a different value x. This is clearly very similar in spirit to our definition of recognisability. However, Spi seems a little too strong to test for guessing attacks: it allows the intruder to test whether an message was encrypted with a particular key, even if the result of the decryption contains

nothing recognisable (cf. the example at the end of Section 4.3). We would like to study the relationship more formally.

There are still some unanswered questions about formalisations of guessing attacks. We list a couple here.

The decision procedure in [Low04] makes the following extra assumption: that during a guessing attack, the intruder would never need to generate a term which is not a subterm of something in the initial knowledge or of something that could be used in the currently executing protocol. Soundness results justifying such assumptions are frequently used in standard analysis of security protocols. Is this assumption still safe in the presence of guessing attacks?

The use of terms to abstract away from actual bit sequences and encryption schemes is common in protocol analysis. Problems with this abstraction have been noted before [CDL05], but a novel problem arises in the context of guess verification when different terms represent bit sequences of different lengths. We illustrate this with an example.

Suppose an intruder knows the identity of an agent A and subsequently overhears a message $\{|A, n_A|\}_k$. Suppose also that A is a 32-bit value, whereas the symmetric key k is 56 bits long. In our framework a guess of k could be verified by decrypting the message with the guess and checking that A appears in the first 32 bits of the resulting plaintext. While this procedure would correctly spot k, it would also mistakenly spot incorrect guesses where the corrupted plaintext happens to have A in the first 32 bits. There would be roughly $2^{(56-32)} = 2^{24}$ of these, making such a verification procedure too inaccurate to be of any use.

A solution might involve annotating abstract values with their bit lengths and devising a function that computes the probability that a guess verification is actually accurate. For the moment we content ourselves with the fact that these attacks are false positives and can be dismissed manually.

8 Acknowledgements

Thanks are due to Eldar Kleiner for constructive comments about this paper.

References

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computing*, 148(1):1–70, 1999.

[CDL05] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 2005. To appear.

[Cho04] T. Chothia. Guessing attacks in the pi-calculus with a computational justification. `http://www.lix.polytechnique.fr/~tomc/`, May 2004.

[CMAFE03] R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess what? Here is a new tool that finds some new guessing attacks (extended abstract). In *Workshop on Issues in the Theory of Security (WITS)*, pages 62–71, 2003.

[Coh02] Ernie Cohen. Proving protocols safe from guessing. In *Proceedings of Foundations of Computer Security*, 2002.

[DJ04] S. Delaune and F. Jacquemard. A theory of dictionary attacks and its complexity. In *Proceedings of the 17th Computer Security Foundations Workshop (CSFW)*, pages 2–15. IEEE Computer Society Press, 2004.

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.

[Low02] G. Lowe. Analysing protocols subject to guessing attacks. In *Workshop on Issues in the Theory of Security (WITS)*, 2002.

[Low04] G. Lowe. Analysing protocols subject to guessing attacks. *Journal of Computer Security*, 12(1), 2004.