A computational justification for guessing attack formalisms

Tom Newcomb and Gavin Lowe

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
{tom.newcomb, gavin.lowe}@comlab.ox.ac.uk

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 (deterministic) 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 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 shared password p as a symmetric key:

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

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 that (with high probability) he has guessed p correctly. Of course, this 'guessing' may be automated, by iterating through some suitable dictionary, using each value in turn.

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 also assume 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 [6] 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 [7] 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: (a) v can be produced in two different ways; (b) v is a value the intruder already knew; or (c) v is an asymmetric key, and the intruder knows its inverse. These conditions were formed by inspection of known guessing attacks, and appear slightly ad-hoc: it is stated in [7] 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 [2–5]. However, these are all reformalisations of Lowe's original definition in different frameworks; we expect that results about Lowe's framework can be easily adapted for these other extensions.

This paper. We propose a more intuitive, (deterministic) 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 refer to this definition by saying 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, [7] 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 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. behaviour not optimal or necessary for the guessing attack), which is not permitted in a verifier-production trace; or create malformed terms (e.g. by decrypting a ciphertext with the wrong key), which might be useful in a guessing attack but is not allowed in a verifier-production trace.

Note that we consider a *deterministic* computational model, as opposed to a model based upon probabilistic polynomial-time Turing machines. We leave the relationship between these two computational models for future work.

Contribution. The contribution of our work can be seen as follows. We give a justification for the definition of verifier production: technically, we show that Lowe's algorithm is sound and complete with respect to our more computational model. This gives us a decision procedure (i.e. that given in [7]) for finding guessing attacks for our more natural definition of guess verification. We also expose an ambiguity in the verifier-production definition.

Related Work. The guessing formalism in [2] is shown to be sound with respect to a computational model, but complete only with respect to the spi-calculus, another algebraic model. We are not aware of any other computational studies of verifier-production techniques for analysing guessing attacks.

Organisation. In Section 2, we give the existing definition of guess verification from [7]. 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. We omit or sketch some proofs because of lack of space; the full proofs can be found in [8].

2 Existing definition: verifier production

In this section, we give the definition of guess verification from [7]; 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:

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

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

$$IK \models_{\langle\rangle} IK,$$
$$S \subseteq IK \wedge S \vdash_l f \wedge IK \cup \{f\} \models_{tr} IK' \Rightarrow IK \models_{\langle S \vdash_l f\rangle \frown tr} IK'.$$

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, one of which must produce the verifier v:

$$IK \cup \{g\} \models_{tr} IK', \tag{1}$$
$$S \vdash_l v \;\; \textbf{in} \;\; tr. \tag{2}$$

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:

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

where undoes is defined by the following rules and their symmetric opposites:

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

Alternatively, the guess could be an asymmetric key whose inverse is already known by the intruder:

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

We will say that a value g is *verifier-producing* from knowledge IK if it is verifiable according to the above definition. 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 invokes a procedure that can tell the difference between correct and incorrect guesses. This procedure may utilise values that the intruder 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:

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

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.

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 and outputs it in some numeric form. P will produce a unique output for every guess, but it is certainly not verifying anything. An alternative definition might say that the program P must output 0 for a wrong guess and 1 for a correct guess. Within our framework, these two definitions are equivalent (Theorem 15). Examples like this 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. A suitable program P to distinguish $K \frown \langle g \rangle$ from $K \frown \langle g' \rangle$, for any $g' \neq g$, would: accept the input guess in a formal parameter x; decrypt $\{\!|v|\!\}_g$ with x; compare the result with v; output 1 or 0 if the test is true or false respectively.

4 A language for programs

Here 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.

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. The set of terms is generated by the following grammar:

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

where terminals D are drawn from a 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 deal only with terms that have been fully *reduced*, according to the following rewrite rules[1]:

[1] Recall our assumption that the intruder has no general way of detecting whether he has decrypted a ciphertext with the correct key; we therefore allow decryptions with arbitrary keys. The last rule is a consequence of that assumption.

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

We use a different notation for terms 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 asymmetric.

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 **fst**, **snd**, **dec**, **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).

Programs. A program is a sequence of instructions of the following forms:

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

where i, j, k are natural numbers. The assignment instructions mimic operations on terms. We also have unconditional and conditional jumps, and outputs.

A program P takes as input 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. 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, halting if the PC ever encounters an empty location.

If there is an assignment instruction $r_k := t$ at PC, then the register r_k is updated to the term t' which is formed from t by substituting names of registers with their values. For example, if 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 it halts. Finally, the PC is increased by one.

Unconditional jumps $\mathbf{goto}\ n$ update the PC to n. Conditional jumps $\mathbf{if}\ r_i = r_j\ \mathbf{goto}\ n$ change the PC to n if the terms in r_i and r_j are syntactically identical; otherwise the PC is increased by one. Unconditional jumps are instances of jumps with conditional $r_0 = r_0$, and will therefore not be mentioned in proofs.

An output command $\mathbf{output}\ k$ 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'.

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 form $K_x = K \frown \langle x\rangle = \langle v,\ \mathbf{enc}(v, g),\ x\rangle$, where x is a guess of g. 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.

When P is run with input K_g, it assigns the term v to r_3 and outputs 1 before terminating. With input $K_{g'}$ for any $g' \neq g$ the decryption does not reduce; r_3 instead holds the term $\mathbf{dec}(\mathbf{enc}(v, g), g')$ and the program outputs 0.

In contrast, g is not recognisable with the knowledge $K' = \langle \mathbf{enc}(v, g)\rangle$. A program could attempt to decrypt the term with the guess, as in P above, but it then has no way of telling whether the decryption succeeded.

5 Completeness

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

Theorem 1 *If g is recognisable from knowledge sequence K, then g is either deducible or verifier-producing from knowledge sequence*[2] *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.

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 $P(K \frown \langle g\rangle) \not\simeq P(K \frown \langle g'\rangle)$.

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. 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 $P(K \frown \langle g\rangle) \not\simeq P(K \frown \langle g'\rangle)$.

Proposition 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.*

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

5.1 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'.

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, and then terminates. 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}. Allowing these instructions in our programs adds no expressive power. We can henceforth equivalently consider such programs.

Unravelling. We now 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. It is enough to consider unravelled programs: we can unwind all loops just enough to distinguish K and K'.

Unique register assignments. We define an unravelled program to have the *unique-reg* property if a previously undefined register is assigned to at every assignment instruction; this is easily obtained by adding registers. This 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 write $P(K)$ for the mapping from registers to terms when program P with input K reaches the final instruction.

No-tail-undo. Here we present a program transformation that will later ensure Condition (5) of verifier production is met. A program satisfies the *no-tail-undo* property if it *doesn't* have one of the following shapes (for any i, j, k, q):

$$\ldots \quad r_i := \mathbf{fst}(r_k) \quad \ldots \quad r_q := \mathbf{pair}(r_i, r_j) \quad \mathbf{signal}\, r_k = r_q$$

or

$$\ldots \quad r_j := \mathbf{snd}(r_k) \quad \ldots \quad r_q := \mathbf{pair}(r_i, r_j) \quad \mathbf{signal}\, r_k = r_q$$

and $P(K)$ 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. It's always possible to transform a program into an equivalent one with this property.

Smallest Normalised Programs (SNPs). We define a *smallest normalised program (SNP)* that distinguishes K *from* K' as a program with minimal instructions that satisfies all of the above properties, and gives output $\langle 1\rangle$ for K and $\langle 0\rangle$ for K'. The following theorem shows that it is enough to consider SNPs.

Theorem 3 *If K and K' are distinguishable then they are distinguishable by a Smallest Normalised Program (SNP).*

Lemmas about SNPs. We now state some lemmas about SNPs that will be useful later when comparing recognisability and verifier production.

Lemma 4 *In $P(K)$, all registers have distinct values except (possibly) the register assigned to in the last assignment.*

The idea is that if two such registers have the same value, one is redundant so can be removed, creating a shorter equivalent program.

Lemma 5 *No SNP has two instructions with identical right-hand sides.*

Lemma 6 *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.*

In each case, $r_l = r_i$ in both $P(K)$ and $P(K')$; hence removing r_l produces a shorter program.

Thanks to the no-tail-undo transformation, we can also show that an extra form of undoing does not occur.

Lemma 7 *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.*

5.2 Well-formedness of deductions in SNPs

Suppose program P distinguishes K from K', so gives output 1 on input K, and 0 on K'. We show that P never produces malformed terms in its registers on input K; this will allow us to write down a well-formed Dolev-Yao trace. Throughout this section, all values of registers referred to are values in $P(K)$.

Recall 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* if it has shape $\mathbf{pair}(\ldots)$ and was created with an instruction of the form $r_i := \mathbf{pair}(\ldots)$ with no reduction applying, or similarly for the other term constructors. Note that non-reduced terms cannot be atoms.

Proposition 8 *Any malformed register values in $P(K)$ are unreduced.*

Proof. We prove the proposition by induction over 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 holds for all registers appearing in some initial segment of the program: registers that are malformed are unreduced. Consider the next instruction in the program that produces a malformed term in a register r_i. We need to show that no reduction occurs in r_i for that assignment. We prove one case; other cases are similar.

Case $r_i := \mathbf{pair}(r_j, r_k)$. We suppose that a reduction applies and establish a contradiction. 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 4, registers cannot have the same value like this, so $p = q$. We now have a pattern of instructions $r_j := \mathbf{fst}(r_p)\ \ldots\ r_k := \mathbf{snd}(r_p)\ \ldots\ r_i := \mathbf{pair}(r_j, r_k)$ in the program P. This contradicts the 'no undoing' property proved in Lemma 6.

Proposition 9 *In $P(K)$, malformedness is* hereditary*: a register assigned to by an instruction using a malformed register is itself malformed.*

Proof. Consider an assignment instruction where one of the registers used on the right-hand side is malformed. If a reduction doesn't occur, then 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. We prove one illustrative case.

Case $r_i := \mathbf{pair}(r_j, r_k)$. If a reduction occurs, then $r_j = \mathbf{fst}(t)$ and $r_k = \mathbf{snd}(t)$ for some t. By Proposition 8 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 4 to find that $p = q$, but this means the program is performs an 'undoing' operation, which contradicts Lemma 6.

Theorem 10 *$P(K)$ does not contain any malformed terms.*

Proof. 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 Proposition 9 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 8, 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 4, we must have $p = p'$ and $q = q'$, which contradicts Lemma 5. Cases for other instructions run similarly.

5.3 Dolev-Yao deduction traces for SNPs

The above theorem tells us that we can form a *corresponding D-Y trace* from $P(K)$. 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 10 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.

Let tr be the corresponding D-Y trace of $P(K)$. We show that no deduction in tr undoes any other deduction.

Theorem 11 *For all pairs of deductions $S \vdash_l v$ and $S' \vdash_{l'} v'$ in tr, we do not have $S \vdash_l v$* undoes *$S' \vdash_{l'} v'$.*

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

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 $r_j = f'$. If the assignment to r_p appears before that to r_q in P, then Lemma 4 demands that $p = k$; so we have $r_k := \mathbf{pair}(r_i, r_j)$ and $r_q := \mathbf{fst}(r_k)$ in P, which contradicts Lemma 6. Alternatively, 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, which contradicts Lemma 7.

Suppose value g is recognisable from knowledge K. Recall from Proposition 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 3, we can deduce that they are distinguished by an SNP P. We now show that P distinguishes g from g', as opposed to vice-versa: the signal condition is true for input K_g and false for $K_{g'}$. The following two propositions are proved in a similar way to those of Section 5.2.

Proposition 12 *Under the assumption that the signal condition in $P(K_{g'})$ is true, each register with g' as a subterm is such that: the register has value g' and it is the last input register; or the register contains g' as a strict subterm and it is unreduced.*

Proposition 13 *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.*

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

Proposition 14 *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 $K_{g'}$. Proposition 13 tells us that g' must be a subterm of a register 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 in the input $K_{g'}$.

Hence the signal condition must be of the form $r_i = r_j$. If these registers both have the value g', Proposition 12 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 10.

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

Theorem 15 *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 $g' \neq g$.*

5.4 SNP deductions are verifier producing

We are finally ready to prove that recognisability implies verifier production. Recall that we assume some value g is recognisable from knowledge K, and g is not D-Y deducible from K, i.e. there is no K' such that $K \models K' \ni g$. It remains to show that g is verifier producing in order to prove Theorem 1.

Recall that Proposition 14 says there is an SNP P and some fresh g' such that the signal condition in P is true for $K_g = K \frown \langle g \rangle$, and false for $K_{g'} = K \frown \langle g' \rangle$. By Theorem 10 we 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), which states that S, the set of facts from which v is deduced, is not itself deducible without g. For example, consider the following initial knowledge IK: $\langle v,\ \{\!|v, x|\!\}_g,\ (((v, x), y), z) \rangle$, and suppose we wish to show that g is guessable. The trace T produced by an SNP will be: $\langle \{\{\!|v, x|\!\}_g, g\} \vdash_{\mathsf{dec}} (v, x),\ \{(v, x)\} \vdash_{\mathsf{fst}} v \rangle$. 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 v. We can then deduce v in a way that doesn't require the guess g. Applying this to our example, we end up with the longer trace $\langle \{\{\!|v, x|\!\}_g, g\} \vdash_{\mathsf{dec}} (v, x),\ \{(((v, x), y), z)\} \vdash_{\mathsf{fst}} ((v, x), y),$ $\{((v, x), y)\} \vdash_{\mathsf{fst}} (v, x) \rangle$. Verifier production is now satisfied when (v, x) acts as the verifier, and $S = \{\{\!|v, x|\!\}_g, g\}$. 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 16 *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. If there is no such chain, then the result of the signal is independent of the guess, giving a contradiction.

Proposition 17 *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 16. 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 $\nexists IK'' \cdot (IK \models IK'' \supseteq S)$, and $IK \cup \{g\} \models_T IK_2$ with $S \vdash_l v$ in T (for some IK_2). 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 now establish Condition (4). If $v \in IK \cup \{g\}$ then (4b) holds, so suppose $v \notin IK \cup \{g\}$. Take $S' \vdash_{l'} v$ to be the deduction in tr_1 that produces v. We have $S \neq S'$ 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, which 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 undoes $S \vdash_l v$ or $S' \vdash_{l'} v$. There cannot be a deduction in T that undoes $S \vdash_l v$ by Theorem 11. Finally, suppose there is a deduction in T that 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 16 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 16 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). Otherwise, 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 4, we must have $p = p'$ and $q = q'$. But this contradicts Lemma 5.

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, Condition (5) follows immediately from Theorem 11.

Case 3: There is no chain of facts from Lemma 16 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 last part of this proof reveals a deficiency in the definition of verifier production from [7]. 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} but it is deducible from the initial knowledge without requiring g.

For example, consider a knowledge set $K = \{\{\!|g^{-1}|\!\}_k, k\}$ for a symmetric key k, and suppose that the intruder wishes to verify a correct guess of g. Intuitively, he can extract g^{-1} from K using the trace $\langle\{\{\!|g^{-1}|\!\}_k, k\} \vdash_{\mathsf{dec}} g^{-1}\rangle$. 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 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:

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

In [7] 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 [7] although it is enforced in the FDR implementation described there.

6 Soundness

We now prove the soundness of verifier production with respect to recognisability.

Theorem 18 *If g is deducible or verifier producing from knowledge sequence K, then g is recognisable from K.*

Proof. **Deducible implies recognisable.** 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. For example, for a deduction $\{f_1, f_2\} \vdash_{\mathsf{pair}} (f_1, f_2)$, we add the instruction $r_i :=$ $\mathbf{pair}(r_j, r_k)$, where r_j and r_k are the registers storing f_1 and f_2, respectively.

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 position of g in K_g (and of g' in $K_{g'}$). We have a program which distinguishes K_g and $K_{g'}$ for all g'.

Verifier producing implies recognisable. First we deal with the case that the updated version of Condition (6) is true. Construct the (possibly empty) program P using the trace from $K \models_{tr} K' \ni g^{-1}$, as above, to obtain g^{-1} in r_j. Then append **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. Construct a program P to calculate v using tr. We then add a **signal** instruction depending on which part of Condition (4) is true: if (4a) holds, compare the two registers which hold the different derivations of v; if (4b) holds, compare the register containing v with the register holding the correct piece of initial knowledge; if (4c) holds, 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. One can also show that $P(K_{g'})$ ends up being false for any datum $g' \neq g$: see [8].

7 Conclusions

Summary. 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 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 guess verification.

We have shown that a previous formalisation of guess verification via verifier production [7] 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 [7] is parameterised by a given set of Dolev-Yao style deductions, whereas we have assumed a standard set of such deductions. While it appears easy in most cases to modify our proofs to deal with different deduction sets, proving the more general result seems much harder.

Abadi and Gordon define secrecy in the Spi Calculus [1] 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 a 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). We would like to study the relationship more formally.

Acknowledgements. Thanks are due to Eldar Kleiner and the anonymous reviewers for constructive comments about this paper.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computing*, 148(1):1–70, 1999.
2. T. Chothia. Guessing attacks in the pi-calculus with a computational justification. http://www.lix.polytechnique.fr/~tomc/, May 2004.
3. E. Cohen. Proving protocols safe from guessing. In *Proc. Foundations of Computer Security*, 2002.
4. 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.
5. S. Delaune and F. Jacquemard. A theory of dictionary attacks and its complexity. In *Proc. 17th Computer Security Foundations Workshop (CSFW)*, pages 2–15, 2004.
6. D. Dolev and A. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.
7. G. Lowe. Analysing protocols subject to guessing attacks. *Journal of Computer Security*, 12(1), 2004.
8. T. Newcomb and G. Lowe. A computational justification for guessing attack formalisms. Research Report 05-05, Oxford University Computing Laboratory, 2005. http://web.comlab.ox.ac.uk/oucl/publications/tr/rr-05-05.html.