2006

Towards a Notion of Quantitative Security Analysis

Iliano Cervesato
Tulane University of Louisiana, iliano@cmu.edu

Published In
.

Towards a Notion of Quantitative Security Analysis

Iliano Cervesato*

Department of Mathematics, Tulane University
New Orleans, LA 70118, USA
iliano@math.tulane.edu

Abstract. The traditional Dolev-Yao model of security limits attacks to "computationally feasible" operations. We depart from this model by assigning a cost to protocol actions, both of the Dolev-Yao kind as well as non traditional forms such as computationally-hard operations, guessing, principal subversion, and failure. This quantitative approach enables evaluating protocol resilience to various forms of denial of service, guessing attacks, and resource limitation. While the methodology is general, we demonstrate it through a low-level variant of the MSR specification language.

1 Introduction

Security protocols have classically been analyzed with respect to the Dolev-Yao intruder [8, 14], a model which gives the attacker complete access to the network, but limits its decryption capabilities to messages for which he possesses the appropriate keys. There is consensus among practitioners that the basic problems of protocol verification, namely secrecy and authentication, are by now solved for this model, as the most recent tools sweep through the standard Clark-Jacob benchmark [6] in mere milliseconds. Recent research has moved in two directions: apply the current tools to the much larger protocols used in the real world, and investigate intruder models that rely on capabilities beyond Dolev-Yao gentlemen correctness. We follow the latter path.

The three tenets of the Dolev-Yao model are (1) the symbolic representation of data, so that a key k is seen as an atomic object rather than a bit-string, (2) the unguessability of secret values such as nonces and keys, and (3) black-box cryptography, by which a message m encrypted with k can be recovered only by a principal in possession of $k^{(-1)}$. All three have been weakened in the last few years. Approaches have taken the bit length of messages and keys into account. Within the symbolic abstraction, effort has been undertaken to include guessing in the intruder's toolkit [11], and to let recurrent algebraic operations, in particular XOR and Diffie-Hellman exponentiation, out of the black box, allowing the intruder to use them to mount an attack (within the accepted computational bounds, *e.g.*, taking a discrete logarithm is not permitted) [4, 7].

The present work takes a symbolic view of data, but allows the intruder to guess values and perform computationally hard operations. That is, if he is willing and able to pay the price. Indeed, we are not so much interested in a *lucky* intruder breaking

* Partially supported by NRL under contract N00173-00-C-2086 and by ONR under contract number N000149910150.

the protocol, but in a *tenacious* one, who will spend Herculean effort in order to gain Alice's confidence or learn Bob's secret. The proposed methodology assigns a cost to both Dolev-Yao and non-standard intruder operations. Depending on the intended use, this cost can be a physical measurement, such as time, space or energy, a complexity class, or simply one of the two values 0 and ∞ in a purely Dolev-Yao model. This work directly extends Meadow's quantitative assessment of denial-of-service [13] and Lowe's analysis of verifiable guesses [11]. It is also related to [16].

Potential applications of this approach include:

- Provide a way for standard analysis methodologies to take intruder effort levels into account. For example, weak secrets are usually modeled as either unguessable or public values. Assigning them an appropriate cost and estimating the resources (or the persistence) of the intruder may help decide whether this secret is too weak for practical purposes. Intruder cost thresholds can be easily integrated into many model checking tools for example.
- Monitoring of network activity, by either an intruder or a law enforcement agency. This entity may then compute the cost of mounting an action against particular communicating agents, using the result to estimate the needed resources.
- Gauge the resilience of a system against denial-of-service scenarios. Cost functions have been used for this purpose [13, 15], but mostly limited to legal Dolev-Yao intruder operations.
- Assess the vulnerabilities of agents meant to operate in a potentially hostile environment with very limited resources in terms of computational power, bandwidth, battery life, etc [5], *e.g.*, , smart cards, PDAs and cellular phones.
- When building a system, compare protocols providing desired functionalities, with respect to their resilience to particular forms of attacks. During the development phase of a protocol, compare alternative designs or parameter choices for optimal resistance to certain attacks. In particular, proposals for denial-of-service protection, *e.g.*, , Juels and Brainard's client puzzles [10] or even the network level proposal of Gunter et al. [9], are good application candidates for this methodology.

As it allows asking "How secure is this protocol?", rather than "Is it secure or not?", the proposed methodology can also be seen as complementing performance and quality of service with an additional quantitative dimension on which to evaluate protocols.

We rely on a Fine-Grained variant of the MSR rule-based specification formalism [1, 2] as a vehicle to introduce this work. Fine-Grained MSR isolates individual verification operations and accounts for the possibility of failure. This is achieved by dividing rule application in a pre-screening phase that commits to a rule, and a more thorough check that fully assesses its applicability. Further details can be found in [3].

2 Background

In MSR, a protocol is specified as a number of *roles*. A role corresponds to the abstract sequence of actions executed by each participating principal. Roles are also used to describe the intruder capabilities. A role itself is given as a sequence of *multiset rewrite*

rules, which describe each individual action. Each rule represents a local transformation of the execution state. It has a *left-hand side* that describes what should be taken out of the state, and a *right-hand side* denoting what it should be replaced with. State objects are modeled using first-order atomic predicates. They include messages in transit ($N(m)$), public information ($M_*(\boldsymbol{m})$), private data of a principal ($M_A(\boldsymbol{m})$), and a record of the status of every executing role ($L^v(\boldsymbol{m})$ — v acts as a program counter, and $\boldsymbol{m}$ is synchronization data). The right-hand side of a rule can additionally mention existentially bound variables to model the creation of nonces and other fresh data. See [3] for details. Example of rules will be shown later in this document.

Simplifying somewhat from [3], the execution semantics of MSR operates by transforming *configurations* of the form $\langle S \rangle_{\Sigma}^{R}$, where the *state* S is a multiset of ground predicates, the *signature* Σ keeps track of the symbols in use, and the *active role set* $R = (\rho_1^{a_1}, \ldots, \rho_n^{a_n})$ records the remaining actions of the currently executing roles (ρ_i), and who is executing them (a_i). In order to add costs to this framework, it is useful to take an even higher-level view of execution. This will also act as an abstract interface where other formalisms can experiment with the techniques in this paper.

An *abstract execution step* is a quadruple $C \xrightarrow{r,\iota} C'$, where C and C' are consecutive configurations, r identifies the rule from $\mathcal{P}$, and ι stands for the instantiating substitution. An abstract execution step is just a compact yet precise way to denote rule application. It is reasonable to think about it as a partial function from C, r and ι to C'. We say that r is *applicable* in C is there is are a substitution ι and a configuration C' such that $C \xrightarrow{r,\iota} C'$ is defined. A *trace* $\mathcal{T}$ is then a sequence of applications

$$C_0 \xrightarrow{r_1,\iota_1} C_1 \xrightarrow{r_2,\iota_2} \cdots \xrightarrow{r_n,\iota_n} C_{n+1}$$

While we rely on the notion of sequence here, this definition could be generalized to a lattice with minimum C_0 and maximum C_n to account for action independence. We will however stick to sequences for simplicity.

A *protocol requirement* for a safety property such as secrecy or authentication is simply given by a set $\mathcal{S}_{\mathcal{I}}$ of *initial configurations* and a set $\mathcal{S}_{\mathcal{A}}$ of *attack configurations*, or some finite abstraction of them. A *verification procedure* decides, for a given protocol, whether there exists a valid trace from an initial to an attack configuration.

A *script* is a parametric sequence of actions $(r_1, \sigma_1), \ldots, (r_n, \sigma_n)$, where the codomain of the σ_i's may mention variables. A script is *realizable* if there are configurations $C_0, \ldots, C_{n+1}$, and grounding substitutions $\gamma_1, \ldots, \gamma_n$ such that $C_0 \xrightarrow{r_1,\iota_1} \cdots \xrightarrow{r_n,\iota_n} C_{n+1}$ is a trace. Scripts describe patterns of execution.

In general, there are two types of scripts of interest: the ones corresponding to the expected runs of the protocol (written $\mathcal{T}_{ER}$), and the scripts that an intruder devises to mount an attack. For our purposes, the latter are more interesting, and we shall extend their syntax for flexibility. An *attack script* is then given by the following grammar:

$$\begin{array}{lll} \mathcal{A} ::= \cdot & & \textit{(Empty script)} \\ \quad \mid \mathcal{A}\ (r, \sigma) & & \textit{(Extension with an action)} \\ \quad \mid !_n\ \mathcal{A} & & \textit{(Script iterated } n \textit{ times)} \\ \quad \mid \mathcal{A} + \mathcal{A} & & \textit{(Alternative scripts)} \end{array}$$

We are particularly interested in attack scripts that are realizable in an initial configuration and end in an attack configuration. We further distinguish *any-time scripts*, which

where honest principals are just responding to intruder solicitation, and *opportunistic scripts*, where the intruder takes advantage of moves initiated by honest principals.

3 Fine-Grained MSR

It is possible to use the definitions in Section 4 to endow MSR with a notion of cost. This would however not be a very precise model, in particular as far as rule application failure is concerned. Therefore, we dedicate this section to defining a finer-grained version of MSR. We isolate the verification operations implicit in an MSR left-hand side as separate rules. During execution, we split rule application into two steps: *pre-screening* commits to a rule, while *left-hand side verification* decides if it should succeed or fail (typically when messages have been tempered with). For space reasons, we describe the compilation of an MSR specification into fine-grained MSR only intuitively.

Fine-grained MSR inherits its language of messages from MSR. However, it makes two changes to the set of available predicates. First, it extends the network predicate with a header h, giving it the template $N^h(m)$. The header is meant to identify precisely a message within a protocol instance: it will typically contain the postulated sender and intended recipient, the name and version number of the protocol and a step locator. An attacker can alter the header at will. The second change is the introduction of predicates $R^v(m)$, which will act as *local registers* during a verification step. Similarly to the local state predicates $L^v(\_)$, the dynamically created superscript v is intended to prevent confusion.

An MSR rule application consists of two distinct phases: the left-hand side mandates a number of verification operations on incoming and retrieved messages, while the right-hand side prescribes how to construct out-going or archived messages. Both are represented succinctly in MSR, yet they can be very complex. Fine-Grained MSR replaces each MSR rule $r = (lhs \rightarrow rhs)$ with a number of *verification rules*, each corresponding to an individual verification step in lhs, and a single *building rule*, which produces rhs. Reducing rhs to atomic steps is not necessary since construction cannot fail once verification has succeeded. Registers are used to serialize these rules in a collection that we call *rule target*.

In order to account for failure, we must split rule application into two stages. During the *pre-screening phase*, a rule is selected based uniquely on the predicate names (including headers and superscripts) appearing in its left-hand side and in the current configuration. In particular, the arguments are not considered. We commit to the selected rule. Then, the *verification phase* checks whether the arguments have the expected form. In case of success, the next configuration is computed as in MSR. In case of failure, the clean-up clause is invoked and the entire role this rule belonged to is removed. See [3] for a formalization of these ideas.

The intruder capabilities traditionally considered for security protocol verification follow the well-known Dolev-Yao model [8, 14]: the intruder can intercept and generate network traffic, take apart and construct messages as long as it has all the elements to do so in a proper way (*e.g.*, it should know the appropriate key in order to perform a

cryptographic operation). This model disallows guessing unknown values and performing operations that are considered "hard" (*e.g.,* recovering a key from a ciphertext). Let I be a memory predicate belonging to the intruder, so that $I(m)$ indicates that it knows (or has intercepted) the message m (this most simplistic setting can be considerably refined). Then, the Dolev-Yao model can be expressed by the following rules:

$$\begin{array}{rlrl} & N^h(x) \to I(x) & & I(x) \to N^h(x) \\ & M_*(\boldsymbol{x}) \to \boldsymbol{I(x)}, M_*(\boldsymbol{x}) & & \cdot \to \exists x.I(x) \\ \boldsymbol{I(y)}, I(\mathsf{op}_{\boldsymbol{y}}(\boldsymbol{x})) & \to \boldsymbol{I(x)} & & \boldsymbol{I(x)} \to I(\mathsf{op}(\boldsymbol{x})) \\ & I(x) \to I(x), I(x) & & I(x) \to \cdot \end{array}$$

The first line corresponds to network interception and injection. The second is access to public information and data generation (when allowed). The third abstractly expresses dismantling and constructing messages (of course, some combinations are disallowed). The fourth line contains administrative rules. Note that, unsurprisingly, these capabilities correspond very closely to the rules of the Fine-Grained MSR [3]. The correspondence would be even more exact if we had reduced the right-hand side to atomic constructions.

The Dolev-Yao model allows the intruder to perform "easy" operations. Once we explicitly assign cost to actions, we can introduce and reason about intermediate degrees between "easy" and "impossible", which is really what the Dolev-Yao restrictions boil down to. Indeed, we will allow attacks that involve performing "hard" operations, guessing values, and subverting principals. We will also be able to quantify "easy", "hard" and levels in between.

The *subversion* of a principal is easily modeled by another intruder memory predicate, $X(A)$. The first row of rules below represent subversion and rehabilitation of a principal A. The others stand for access to A's private data and for the intruder covering its traces.

$$\begin{array}{rlrl} \cdot \to & X(A) & & X(A) \to \cdot \\ X(A), M_A(\boldsymbol{x}) & \to X(A), \boldsymbol{I(x)} & & X(A), \boldsymbol{I(x)} \to X(A), M_A(\boldsymbol{x}) \end{array}$$

We model *"hard" operations* by simply extending the set of patterns allowed in rule template $(\boldsymbol{I(y)}, I(\mathsf{op}_{\boldsymbol{y}}(\boldsymbol{x})) \to \boldsymbol{I(x)})$ to represent non Dolev-Yao inferences. For example, taking a discrete logarithm is expressed as:

$$I(g), I(g^x) \to I(x).$$

Clearly there are limitations to this method as it applies only to the inversion of bijections. Other "hard" operations, such as finding hash collisions, can be modeled as guessing problems.

The trivial *guessing rule* $(\cdot \to I(x))$ is unrealistic and hard to work with from a cost accounting point of view. Therefore, following the pioneering work of Lowe [11], we require that every guess be backed up by a *verification procedure*. We express both the guess and its verification as an MSR role of the following form:

$$\exists u, v_1, v_2. \left[\begin{array}{r} \ldots \rightarrow G^u(x), \ldots \\ \ldots \\ \ldots \rightarrow \ldots, V_1^{v_1}(m_1) \\ \ldots \rightarrow \ldots, V_2^{v_2}(m_2) \\ V_1^{v_1}(y), V_2^{v_2}(y), G^u(x) \rightarrow I(x) \end{array}\right] \begin{array}{l} \} \text{ Guess} \\ \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \text{Verification} \end{array}$$

On the right, G, V_1 and V_2 are local state predicates (generically called L in Section 2) that hold the guess and two constructions (the *verifiers*) that should produce the same value if the guess is correct. See [11] for conditions required of acceptable verifiers. The exact format can vary, as illustrated below. Guessing roles are protocol specific, in general.

As a concrete example, the following role expresses the guess of a Diffie-Hellman exponent:

$$\exists u, v. \left[\begin{array}{r} I(g^x) \rightarrow G^u(x'), V^v(g^x, g^{x'}) \\ G^u(x), V^v(y, y) \rightarrow I(x) \end{array}\right]$$

Note that, although this role is functionally equivalent to the discrete logarithm specification above, the exponent is explicitly guessed here rather than reverse-engineered as above.

Our final example describes the guess of the shared key k in the toy protocol informally described to the right of this text.

$$\begin{array}{l} A \rightarrow B: \ \{n_a\}_k \\ B \rightarrow A: \ n_a \end{array}$$

$$\exists u, v. \left[\begin{array}{r} \cdot \rightarrow \exists n. G^u(k), N^h(\{n\}_k), V^v(n) \\ G^u(k), V^v(n), N^{h'}(n) \rightarrow I(k) \end{array}\right]$$

Here, the intruder generates a nonce n, makes a guess for k and sends the expected message to B (we ignored header-formatting issues for simplicity). This copy of n is the first verifier and is memorized in the predicate V. The second verifier is simply the response from B: if the guess was correct, they will be equal, otherwise it will either come back as a different bit-string, or be dropped by B altogether if the forgery attempt is uncovered.

4 Cost Model

Traditional approaches to protocol analysis are only interested on whether an action is applicable in a given state. Actions that are not applicable, either because they cannot succeed or because "computationally infeasible", are unobservable. In this paper, we are concerned with the cost of successful and failed applications. Cost will be measured in terms of whatever resource of interest changes as a result of attempting the action. Primary focuses are time and storage, but other parameters, such as energy, or the lowered randomness of some quantity (that may be used for side-channel attacks, for example) can also be used.

4.1 An Algebra of Cost

We will now define a generic infrastructure for expressing cost. The details of the resulting algebra shall be application specific.

We want to associate a value to each type of cost incurred by a principal. A *type*, denoted with τ, describes a resource of interest, time, space and energy are typical, but more refined types, *e.g.*, verification vs. construction time, can also be expressed. A *cost base* relates a cost type τ to a principal a. We write it as τ^a.

How much of a given cost type is incurred by a principal takes the form of a *scalar value*, which we will denote with s. Scalars can be abstract quantities (*e.g.*, Meadow's "*cheap*", "*medium*", "*expensive*", etc. [13], or just 0 and ∞ in a Dolev-Yao setting), numbers (in $\mathbb{N}$ or $\mathbb{R}$ for example), or even complexity bounds in $\mathcal{O}$-notation. It is useful that some form of addition (written $+$) and a unit (0) be defined on scalars. These could be just free symbols, but $+$ can also be an actual operation. It is also very useful to have a comparison relation (written $<$, with the usual variants) among scalars within a cost base. Note that some forms of cost never decrease and $+$ should be monotonic with respect to $\leq$ for them. Time or energy are examples. This is the only case considered in [13]. Other costs, in particular space, do not need to be monotonic, and this restriction does not apply.

A *cost item* is a cost base τ^a together with a scalar value s. We denote it as $s\tau^a$. We extend the scalar comparison operators to cost items only when the base is the same. Such an extension rarely makes sense if the cost type is different, and should be evaluated on a case by case basis when the principals are not the same: one byte is one byte for everybody, but performing a decryption will generally take different amounts of time when hardware or implementation varies.

At this point, a *cost vector* $\mathcal{C}$ is simply a collection of cost items $s_1\tau_1^{a_1}, \ldots, s_n\tau_n^{a_n}$, which we write $\sum_i s_i\tau_i^{a_i}$. Given a cost vector $\mathcal{C}$, we write $\mathcal{C}^a$, $\mathcal{C}_\tau$, and $\mathcal{C}_\tau^a$ for its projections relative to principal a, cost type τ, and their combination, respectively. For example, $\mathcal{C}^a = \sum_{s\tau^a \in \mathcal{C}} s\tau^a$. It should be noted that a cost vector can be seen as a generalization of the notion of multiset.

4.2 Cost Assignment for Protocol Operations

In spite of their apparent simplicity, cryptographic protocols comprise a large number of operations and action classes. We will now examine them and comment on their characteristics in term of cost. Most of the issues are discussed relative to Fine-Grained MSR, and transpire also at the level of MSR. Needless to say, similar considerations apply to other specification languages.

Network: The network operations observable in MSR are receiving and sending a message. We denote their associated cost as $\kappa_{N\Rightarrow}$ and $\kappa_{\Rightarrow N}$, respectively. This generally includes time and storage components. Accounting for other transmission costs such as network latency could be easily accommodated through a simple refinement of (Fine-Grained) MSR.

Storage: Each of public (M_*), private (M_a) and local (L) storage has a temporal and a spatial component. Storage operations include allocating and recording data (*e.g.*, $\kappa_{\Rightarrow M_a}$), disposal ($\kappa_{M_a\perp}$) and look-up (κ_{M_a}). Notice in particular that the spatial component of storage disposal is negative. Note also that some values may be easier to look-up than others, and so κ_{M_a} depends on the actual predicate M.

Registers: We do not associate any cost with register management, preferring to fold it into the operations they participate in.

Constructor operations: Each constructor op has a number of operations associated with it. We consider its use as a building block of a message ($\kappa_{\Rightarrow \mathsf{op}}$) and during verification. In the latter case, we distinguish between the cost of success ($\kappa_{\mathsf{op}\surd}$) and failure ($\kappa_{\mathsf{op}\perp}$). The cost of performing Dolev-Yao and non Dolev-Yao operations is computed in the same way in our model. What will change is likely to be the magnitude of the scalar values.

Data operations: Atomic values are subject to generation (using $\exists$ in MSR), and generic values can be tested for equality. We write $\kappa_{\zeta\exists}$ and $\kappa_{\zeta=}$ respectively, where ζ represents some notion of type of a value.

Subversion: We write $\kappa_{?a}$ for the cost of subverting principal a and $\kappa_{!a}$ for the cost of its rehabilitation.

Guessing: The cost of a guessing attack can be modeled in two ways. At a high level of abstraction, we can associate a cost to a verification procedure ρ_G as a whole, which accounts for the cost of the expected number of guesses and verifications until one is successful. We write κ_{ρ_G} for this omnibus, MSR-oriented, cost. Alternatively and at a much lower-level level of detail, we can compile the verification procedure to Fine-Grained MSR, obtaining a role $\bar{\rho}$, assign a cost to the individual guess itself (κ_G), compute the cost of each guess and verification, $\mathcal{C}(\bar{\rho})$, as outlined below, and estimate the number of attempts it may take until a successful guess is produced. In general, this type of accounting will have the form $f(n)\,\mathcal{C}(\bar{\rho})$, where f is a function and n is a parameter such as the length of the data to be guessed.

Each of these operations, with sometimes the exception of guessing, are executed by a single principal, say a (which may also be the intruder). Each will in general involve several cost components. Therefore, $\kappa_{\_}$ corresponds to a cost vector relative to a. Guess verification can be performed locally by the intruder, or require exchange of messages with one or more principals. In the latter case, the cost vector will have appropriate components for each of the involved parties.

In general, the accuracy of a cost-based analysis directly depends on the precision of the cost associated with each basic action. For example, a classification into "*cheap*" and "*expensive*" forms the basis for a Dolev-Yao investigation, while adding an intermediate "*medium*" value already provides a setting in which one can start analyzing denial-of-service situations [13]. Moving to numerical classes adds flexibility, but non-trivial problems quickly emerge as accurate physical measurements can be difficult to gather and work with when dependent on hardware, implementation and system load. In this paper, we provide a flexible framework for taking cost into consideration, but have little to say at this stage about how to best determine the granularity and magnitude of basic costs.

4.3 Cost Calculation in MSR

The notion of cost naturally extends from individual operations to traces. First, we define the cost of a Fine-Grained MSR rule by simply adding up the cost of each operation occurring in it. There is little to do in the case of the verification rules, while building rules involve some work. Some rule have the option of failing, and therefore both a success and a failure cost is associated with them: we shall consider them as if they were different operations.

Consider now a trace $\mathcal{T} = C_0 \xrightarrow{r_1,\iota_1} \cdots \xrightarrow{r_n,\iota_n} C_{n+1}$. Let a_j be the principal executing (r_j, ι_j), and $\mathcal{C}^{a_j}(r_j) = \sum_i s_{ij}\tau_{ij}^{a_j}$ its cost. The cost of the trace is then given by

$$\mathcal{C}(\mathcal{T}) = \sum_j \mathcal{C}^{a_j}(r_j) = \sum_j \sum_i s_{ij}\tau_{ij}^{a_j}$$

The cost calculation for a trace extends naturally to the cost of a script since substitutions do not play any role when computing a cost. The presence of alternatives in an attack script forces us to define cost for them over (multi-dimensional) intervals rather than points. We have the following definition:

$$\begin{array}{lcl} \mathcal{C}(\cdot) & = & I_0 \\ \mathcal{C}(\mathcal{A}\ (r,\sigma)) & = & \mathcal{C}(\mathcal{A}) + \mathcal{C}^{a_{[\sigma]r}}(r) \\ \mathcal{C}(!_n\ \mathcal{A}) & = & n\,\mathcal{C}(\mathcal{A}) \\ \mathcal{C}(\mathcal{A}_1 + \mathcal{A}_2) & = & [\min\{\mathcal{A}_1, \mathcal{A}_2\}, \max\{\mathcal{A}_1, \mathcal{A}_2\}] \end{array}$$

Here, I_0 is some fixed interval, typically $[0, 0]$. We extend scalar product and addition to intervals by applying these operations to its endpoint, *i.e.*, $n[a, b] = [na, nb]$ and $[a, b] + [c, d] = [a + c, b + d]$.

Since most tools for security protocol analysis rely, often symbolically, on traces, the infrastructure we just outlined is compatible with their underlying methodology. Indeed, systems based on explicit model checking can immediately take costs into account, while symbolic approaches need to have the cost model indirectly encoded as part of the problem description. Similar considerations applies to analysis based on theorem proving. In general, how easy it is to extend a tool with cost computation capabilities depends on how deeply the intruder model is ingrained in their implementation. The required modifications include tracking cost and allowing for non Dolev-Yao intruder actions.

Note that any tool natively supporting cost calculation (or even retrofitted to do so) can still perform traditional verification by assigning cost ∞ to non Dolev-Yao intruder actions and abandoning any attack trace as soon as its cost reaches ∞.

5 Quantitative Security Analysis

A first-class notion of cost leads to protocol analysis opportunities that lay far beyond the traditional Dolev-Yao feasibility studies. In this section, we will examine some of the possibilities related to time and space, well aware that many more lay out there, waiting for the imaginative mind to grab them. We elaborate on two non Dolev-Yao forms of verification: *threshold analysis* tries to determine what attacks are possible given a bound on the resources available to the intruder alone; *comparative analysis* studies attack opportunities when the resource bounds of all involved parties are taken into consideration. Denial-of-service attacks are a prime example.

5.1 Threshold Analysis

A rather trivial use of cost is to first ascertain that a protocol is secure relative to the Dolev-Yao model, and then compute the amount of resources it requires. This may

be useful already in situations characterized by limited capacities, such as protocols implemented on smart-cards. If κ_{HW} is an inventory of the available resources, this problem is abstractly stated as "$\mathcal{C}(\mathcal{T}_{ER}) \leq \kappa_{HW}$?".

Dually, an intruder can pre-compute the cost of mounting an attack on a discovered vulnerability. This is generally not very interesting in a Dolev-Yao setting where an attack uses the same kind of operations as the protocol itself, and the intruder is implicitly assumed to have access to resources similar to honest principals. This becomes crucial when the intruder experiments with "computationally infeasible" operations, principal subversion, guessing, or a combination of these non Dolev-Yao operations. Indeed, some protocol analysis tools already allow principals to "lose keys" [12], but do not assign any special status to this operation. The intruder can then calculate the cost of a candidate attack and compare it with its available resources (dictionary attacks on passwords are the simplest instance), in symbols "$\mathcal{C}(\mathcal{A}) \leq \kappa_I$?". A protocol verification tool can similarly discard attack traces as soon as their cost exceeds a predetermined amount of intruder resources.

A protocol designer can go one step further by keeping aspects of the cost calculation as parameters. He can then determine value ranges that would require extravagant amounts of resources from an intruder in order to implement the attack (given foreseeable technology): "$\min x.\mathcal{C}(\mathcal{A}(x)) \gg \kappa_I$?". This is how key lengths and other parameters of cryptographic algorithms have traditionally been set. The approach we are promoting extends this form of safe parameter determination in that it takes into account the whole protocol rather than an isolated cryptographic primitives. This is particularly valuable as modern ciphers offer the option of variable key lengths.

5.2 Comparative Analysis

A cost infrastructure can be useful to a designer to choose a protocol among two candidates based on resource usage "$\mathcal{C}(\mathcal{T}^{P_1}) > \mathcal{C}(\mathcal{T}^{P_2})$?", or on their resilience to a certain type of attacks: "$\mathcal{C}(\mathcal{A}_1) > \mathcal{C}(\mathcal{A}_2)$?". By the same token, an attacker or law enforcement agency can evaluate attack strategies based on their cost.

Denial-of-service (DoS) attacks operate by having a possibly distributed intruder waste a server's resources with fake requests to the point where legitimate uses cannot be serviced in any useful time frame (or the server crashes). It stresses the bounds on the server's resources, typically time (or service rate) and storage capacity. A precise cost analysis, like the one proposed here, helps compute actual values for the resources used by both the intruder and the server at different stages of the protocol execution. The statement here is "$\mathcal{C}^B(\mathcal{A}) > \mathcal{C}^I(\mathcal{A})$?". Given assumptions about performance and buffer sizes, it can help determine how many requests can be handled concurrently and in particular by how many compromised hosts. The same calculation can be used to determine the amount of resources needed to withstand a given target level of attack.

Consider the abstract protocol below (left), where a client C initially contacts the server S with some message m_1, is given a challenge m_2, and receives the requested service m_4 only after it has provided an adequate response m_3 to m_2:

$$
\begin{array}{lcccccl}
 & C & & & S & & \\
C \to S : m_1 & s_1^C & t_{b1}^C & \xrightarrow{m_1} & t_{v1}^S & s_1^S & \\
S \to C : m_2 & s_2^C & t_{v2}^C & \xleftarrow{m_2} & t_{b2}^S & s_2^S & \\
C \to S : m_3 & s_3^C & t_{b3}^C & \xrightarrow{m_3} & t_{v3}^S & s_3^S\ [= -(s_1^S + s_2^S)] & T \\
S \to C : m_4 & s_4^C & t_{v4}^C & \xleftarrow{m_4} & t_{b4}^S & s_4^S\ [= 0] &
\end{array}
$$

The exchange on the right shows the time (t_{xi}^a) and space (s_i^a) cost incurred by each principal. Let us measure time in seconds and space in bytes. We wrote t_{bi}^a for the time a spent building message m_i and $t_{vi}^{a'}$ for the time a' spent verifying it. For simplicity, we assume that the time incurred in a failed verification is also t_{vi}^a. Our approach allows for a much more precise model. It is reasonable to assume that the server will not allocate any buffer space upon sending m_4, hence $s_4^S = 0$ and that it releases any used buffer space as soon as it has verified m_3, *i.e.*, $s_3^S = -(s_2^S + s_3^S)$. We further assume that the server will time-out after T seconds if it does not receive message m_3 from C. In this case it will deallocate the space $s_2^S + s_3^S$.

This simple protocol template is susceptible to three forms of time DoS, and one form of space DoS:

- An attacker can induce the server to waste time unsuccessfully verifying a fake message m_1. This time is at most t_{v1}^S. The server's verification rate is therefore at least $1/t_{v1}^S$, which must be matched by the intruder in order to successfully attack S. While this is easily achieved as a fake m_1 can be an arbitrary string, t_{v1}^S will often be comparable to networking overhead in a protocol designed with DoS attacks in mind.

  As a concrete example, a simple initial request containing the client's name, a timestamp, a nonce and a checksum will take under 1μs to verify on fast hardware. Therefore, the server can process at least 1,000,000 requests per second. Assuming that the server has a 1 Gbit/s network interface and that request packets are 50 bytes long (*i.e.*, 400 bits), the network layer will be able to deliver 2,500,000 packets per second to the protocol.

  A dedicated attacker may match these numbers. He may also perform the attack through a number of compromised hosts, which will typically have more limited computing power and bandwidth. While an arbitrary string can be put together in 1μs on many home computers, typical outbound network speeds are less than 4 Mbit/s. Therefore, the attacker will need to synchronize 250 compromised hosts to overwhelm the server with a simultaneous attack.
- A time-out waiting for the reception of m_3 leads to another potential point of DoS. In this case, the server has spent $t_{v1}^S + t_{b2}^S$ while the attacker has incurred a cost t_{b1}^C. Again, this gives us a way to compare the attacker's and the server's rate.

  Continuing our example, $t_{v1}^S + t_{b2}^S$ may amount to 100μs as the server's response will generally involve the generation of a nonce or of cryptographic material. Therefore, the resulting rate may be 10,000 replies per second.
- Another option for time DoS is the reception of a fake message m_3 by S. Here S needs to spend $t_{v1}^S + t_{b2}^S + t_{v3}^S$ seconds, while the attacker's cost amounts to t_{b1}^C plus the minimal time it takes to produce the counterfeit m_3 (the intruder is likely to ignore m_2). This strategy wastes more server time, but it will release storage

earlier unless carefully timed. Moreover, the reception of a large number of garbled message may trigger countermeasures on the server.
Looking at our example again, the verification of a fake message m_3 will typically involves substantial use of cryptography, often expensive asymmetric cryptography. We can then take $t^S_{v1} + t^S_{b2} + t^S_{v3}$ to be 10 milliseconds, which results in a rate of just 100 exchanges per second.

In all these situations, the resilience of the server is given by comparing the service rate as measured above, with the individual attack rate multiplied by the number of attackers. Our methodology can give useful ranges as it takes into account the exact structure of the messages involved, including that of the messages faked by the intruder.

– A time-out on m_3 is also the target of a space DoS. Let B be the size in bytes of the buffer where S stores received bits of m_1 and generated fragments of m_2. Then, S can serve at most $n(B) = B/(s^S_1 + s^S_2)$ concurrent requests: the larger B, the larger the number of parallel attacks the system can withstand. The space allocation rate is given by $(s^S_1 + s^S_2)/(t^S_{v1} + t^S_{b2})$ bytes per second relative to an individual attacker, while the space reclamation rate is at least $(s^S_1 + s^S_2)/(T + t^S_{v3})$.
Now, given B, we can calculate optimal values for the time-out T. First, T should be large enough for all legitimate usage pattern to complete: $T > t_{\min}$. On the other hand, it should not be so large that an attacker coalition may file more than $n(B) - 1$ fake service requests while waiting for time-out on any initial exchange: $T \leq (t^S_{v1}+t^S_{b2})\times(n(B)-1)$. We are looking for the maximum value of T satisfying these bounds.
Concretely, if $s^S_1 + s^S_2 = 128$ bytes, $t^S_{v1} + t^S_{b2} = 10$ milliseconds, $t_{\min} = 90s$, and the maximum number of expected parallel attacks is $10,000$, we deduce that B should be at least 1.28 Mb, and that T can be about 1 minute and 40 seconds. If this value is too low, then B should be increased (which would make the system resilient to more concurrent attacks).

References

1. I. Cervesato. A specification language for crypto-protocol based on multiset rewriting, dependent types and subsorting. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proceedings of the Workshop on Specification, Analysis and Validation for Emerging Technologies — SAVE'01*, pages 1–22, Paphos, Cyprus, 2001.
2. I. Cervesato. Typed MSR: Syntax and examples. In V. Gorodetski, V. Skormin, and L. Popyack, editors, *First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security — MMM'01*, pages 159–177, St. Petersburg, Russia, 2001. Springer-Verlag LNCS 2052.
3. I. Cervesato. Fine-Grained MSR Specifications for Quantitative Security Analysis. In *Fourth Workshop on Issues in the Theory of Security — WITS'04*, pages 111–127, 2004.
4. Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the Security of Protocols with Diffie-Hellman Exponentiation and Products in Exponents. In *FSTTCS 2003: Foundations of Software Technology and Theoretical Computer Science — 23rd Conference*. Springer-Verlag LNCS 2914, 2003.

5. K. Christou, I. Lee, A. Philippou, and O. Sokolsky. Modeling and analysis of power-aware systems. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems — TACAS 2003*, pages 409–425, Warsaw, Poland, 2003. Springer-Verlag LNCS 2619.
6. J. Clark and J. Jacob. A survey of authentication protocol literature. Technical report, Department of Computer Science, University of York, 1997. Web Draft Version 1.0 available from `http://www.cs.york.ac.uk/~jac/`.
7. H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *Proceedings of the Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 261–270. IEEE, Computer Society Press, 2003.
8. D. Dolev and A. C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.
9. C. Gunter, S. Khanna, K. Tan, and S. Venkatesh. DoS protection for reliably authenticated broadcast. In M. Reiter and D. Boneh, editors, *Proceedings of the 11th Networks and Distributed System Security Symposium — NDSS'04*, San Diego, CA, 2004.
10. A. Juels and J. Brainard. Client puzzles: a cryptographic defence against connection depletion attacks. In S. Kent, editor, *Proceedings of the 5th Networks and Distributed System Security Symposium — NDSS'99*, pages 151–165, San Diego, CA, 1999.
11. G. Lowe. Analysing protocols subject to guessing attacks. In J. Guttman, editor, *Second Workshop on Issues in the Theory of Security — WITS'02*, Portland, OR, 2002.
12. C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
13. C. Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1/2):143–164, 2001.
14. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
15. D. Tomioka, S. Nishizaki, and R. Ikeda. Cost estimation calculus for analysing denial-of-service attack resistance. Pre-proceedings of ISSS'03, Tokyo, Japan, Nov. 2003.
16. R. Zunino and P. Degano. A note on the perfect encryption assumption in a process calculus. In I. Walukiewicz, editor, *7th International Conference on Foundations of Software Science and Computation Structures — FoSSaCS'04*, pages 514–528, Barcelona, Spain, 2004. Springer-Verlag LNCS 2987.