Using Equivalence-checking to verify robustness to Denial of Service

Stéphane Lafrance

Département de génie informatique
École Polytechnique de Montréal
P.O. Box 6079, Station Centre-ville
Montréal (Québec), Canada, H3C 3A7

Abstract

In this paper, we introduce a new security property which intends to capture the ability of a cryptographic protocol being resistant to denial of service. This property, called impassivity, is formalised in the framework of a generic value-passing process algebra, called Security Protocol Process Algebra, extended with local function calls, cryptographic primitives and special semantics features in order to cope with cryptographic protocols. Impassivity is defined as an information flow property founded on bisimulation-based non-deterministic admissible interference. A sound and complete proof method, based on equivalence-checking, for impassivity is also derived. The method extends results presented in a previous paper on admissible interference and its application to the analysis of cryptographic protocols. Our equivalence-checking method is illustrated throughout the paper on the TCP/IP connection protocol and on the 1KP secure electronic payment protocol.

Key words: Equivalence-checking, Denial of service, Protocols, Process Algebra, Admissible interference, Bisimulation

1 Introduction

The sudden expansion of electronic commerce has introduced an urgent need to establish strong security policies for the design of security protocols. Formal validation of security protocols has since become one of the primary tasks in computer science. In recent years, equivalence-checking has proved to be useful for formal verification of security protocols [1–3]. The main idea behind...
this approach is to verify a security property by testing whether the process (specifying the protocol) is equivalent to its intended behaviours. Many other methods, coming from a wide range of approaches, have been proposed in the literature to analyse security protocols, but most are devoted to validation of confidentiality and authentication policies. Little attention has been paid to denial of service (DoS) up to now. However, the inability to clearly establish a formal definition for DoS has made this type of attack a growing concern for protocol designers. The main contribution of this paper is an equivalence-checking method for detecting DoS vulnerabilities in security protocols.

The basic idea behind the validation method developed in this paper consists on proving that no intruder may use the protocol to send, with little effort, a fake message in order to force costly actions like memory allocation, decryption or signature verification. Such a flaw could be exploited by an intruder in order to launch a DoS attack by wasting the victim’s valuable resources. These requirements guarantee that the protocol provides protection against an intruder spending minimal resources. Robustness to DoS is therefore formalised by the security property which requires that no intruder can use the protocol to interfere with costly actions of the defender, in order to cause resource exhaustion DoS. Non-interference properties [4] capture any causal dependency between intruder’s behaviours and other principal’s behaviours. Non-interference was also successfully applied to cryptographic protocols analysis [5–7]. However, many practical security concerns go beyond the scope of non-interference. For instance, we do not want to detect interference coming from an intruder behaving correctly (e.g. an intruder initiating a protocol run with its true identity or responding honestly to a request). Admissible interference [8,9] properties allows us to ensure that interference occurs only through such honest behaviours.

The first step toward validation of security protocols is to find a language that may express both the protocols and the security policies we want to enforce. Process algebra has been used for some years to specify protocols as a cluster of concurrent processes, representing principals participating to the protocol, which are able to communicate in order to exchange data. Process algebra CSP was one of the first language to be successfully used in this matter [10,11]. In cryptographic based process calculi like Abadi & Gordon’s spi calculus [12] and Focardi & Martinelli’s CryptoSPA [6], cryptographic manipulations are formalised in a parallel inference system. Therefore, they are not directly observable from the process semantics. For instance, a principal sending a message $m$ encrypted with a key $k$ is simply modelled as an output action “$c(\{m\}_k)$” whenever $\{m\}_k$ can be inferred from the principal’s current knowledge. However, information flow properties like non-interference and admissible interference usually require such manipulations to be observable. For that purpose, we establish our theoretical framework within a generic process algebra with value-passing called Security Protocol Process Algebra (SPPA),
with extensions to allow viewing local function calls made by a principal (process) as visible actions and using marker actions to keep track of information exchanges, otherwise lost during communication. In SPPA, a principal $A$ sending a message $m$ encrypted with a key $k$ is modelled as the action “$\text{enc}_{id_A}$” (the principal $A$ encrypting the message) followed by the output action “$\text{cid}_A(m_k)$”, where $id_A$ is the principal’s identifier. Moreover, every action is assigned to a cost describing the quantity of CPU and memory resources required to execute it. This cost-based framework allows us to analyse the information flow of security protocols in terms of CPU and memory spent by each principal. Formal validation is founded on algebraic specifications, using this extension of SPPA, of both the protocol and the DoS attacks. Hence, intruder capabilities are explicitly captured through regular SPPA processes, each pursuing a specific attack.

There have been a number of different formalisations of non-interference, many of them in terms of process algebra [13,14]. These formalisations are usually accompanied with equivalence-checking-based proof methods (traditionally called unwinding theorems). Bisimulation and trace equivalence are the most common equivalence relations used to compare processes. Bisimulation offers an expressive semantics to process calculi that refines trace equivalence. Moreover, a polynomial-time algorithm was designed for bisimulation, while trace equivalence is known to have exponential complexity (EXPSPACE-complete). In this paper, we consider a formalisation in SPPA of admissible interference called bisimulation-based non-deterministic admissible interference (BNAI) [3]. We then proceed to introduce an interpretation of BNAI in the context of DoS robustness called impassivity which requires no causal dependency between low-cost enemy’s behaviours and high-cost actions (hence, potentially exhausting actions) of other principals. We show that impassivity is satisfied whenever the costly behaviours (in terms of CPU or memory) of the principal being attacked are bisimilar to the costly behaviours of the principal within a regular protocol run (not being attacked). Roughly speaking, if the principal behaves differently when it is being attacked, then we conclude that the protocol is unsafe. The protocol’s costly actions are identified simply by establishing cost levels, called capacities, defined in such a way that running simultaneously several actions, with cost over the defender’s capacities, could cause a DoS.

Yu & Gligor [15] proposed a formal specification and verification method for the prevention of DoS that relies essentially on an access control policy called user agreement. Using temporal logic, they introduced fairness, simultaneity and finite-waiting-time policies, combined to a general resource allocation model. Yu & Gligor argued that DoS may be viewed as liveness problems (some users prevent some other users from making progress within the service for an arbitrary long time) and safety problems (some users make some other users receive incorrect service). Millen [16] extended the Yu-Gligor resource
allocation model by explicitly representing the passage of time. By doing so, Millen can support policies of a probabilistic nature, like a maximum-waiting-time policy. Cuppens & Saurel [17] introduced a similar approach using an availability policy formalised within temporal logic and deontic logic.

The approaches developed around Yu & Gligor’s frameworks do not offer protection against attacks occurring before parties are authenticated to each other, as it is the case for distributed denial of service attacks. In network DoS attacks on protocols establishing authenticated communication channels, the identity of the intruder is generally unknown because authentication has not yet been completed. A design technique for making protocols more resistant to such attacks consist on using a sequence of authentication mechanisms, arranged in order of increasing cost (to both parties) and increasing security. With this approach, an intruder must be willing to complete the earlier stages of the protocol before it can force a system to expend resources running the later stages of the protocol. Meadows [18] proposed a cost-based framework for analysing such protocols. Her framework is based on Gong & Syverson’s fail-stop protocol model [19]. Meadows interprets this fail-stop model by requirements specification based on Lamport’s causally-precedes relation setting which events should causally-precede others in a protocol. The idea behind this framework is that although an intruder is capable to break some of the (weak) authentication steps of a protocol, it will cost him an unworthy amount of efforts. It offers prevention against multiple exploitations of a single flaw which could lead to DoS.

The paper is organised as follows. Process algebra SPPA, which copes with architecture of cryptographic protocols, is described in Section 2. In Section 3, a short introduction to the notions of non-interference and admissible interference is given together with unwinding theorems. Our information flow method for the detection of resource exhaustion DoS vulnerabilities is presented in Section 4, along with a sound and complete proof method based on bisimulation (Definition 4). Substantial applications on the TCP protocol and the 1KP secure electronic payment protocol are given throughout the paper and in Section 5. In Section 6, we discuss related and future works.

### 1.1 The SYN flooding attack

In recent years, several Internet sites have been subjected to DoS attacks. One of the most famous DoS is the SYN flooding attack [20] on the TCP/IP protocol. Since 1996, this resource exhaustion attack has been launched at several occasions by intruders having the capabilities to initiate, with little effort, a large number of protocol runs. This DoS on the TCP/IP protocol is possible due to the facility to forge an identity, thus the difficulty for the victim
to identify an intruder. Other DoS attacks have been perpetrated on Internet e-commerce sites, including Yahoo, Ebay and E*trade in February 2000, and Microsoft in January 2001. The most lethal distributed DoS attacks have also caused their share of mayhem. Some tools devoted to distributed DoS [21,22] were developed for the analysis of such attacks based on specific malicious applications like Trin00, TFN2K and Stacheldraht.

The Transmission Control Protocol (TCP) provides a reliable connection-oriented data stream delivery service for applications. The TCP connection protocol will serve as our main illustration throughout this paper. A TCP connection commonly uses memory structures to hold data related to the local end of the communication link, the TCP state, the IP addresses, the port number, the timer, the sequence number, the flow control status, etc. (A full description of TCP in terms of a state machine is given by Schuba & al. [20].) Before starting the transmission of data between a source (client) A and a destination (server) B, TCP relies a connection establishment protocol called the three-way handshake. The three-way handshake is achieved with the following steps:

Message 1: \( A \xrightarrow{SYN_n} B \)

Message 2: \( B \xrightarrow{SYN_m,ACK_n} A \)

Message 3: \( A \xrightarrow{ACK_m} B \).

First, A initiates the connection by sending to B a synchronisation packet \( SYN_n \), containing a fresh sequence number \( n \) along with the IP addresses of A and B. Next, B acknowledges the first message and continues the handshake by sending packets \( ACK_n \) and \( SYN_m \) to A. Finally, A acknowledges B’s packets by replying \( ACK_m \). Whenever B receives a SYN packet, data structures are allocated.

The SYN flooding resource exhaustion attack on the TCP protocol is initiated as follows:

newId: the intruder generates a fake identifier (IP address) \( id \);
newNumber: the intruder generates a random number \( n \);
makeSYN: the intruder creates a synchronisation packet \( SYN_n \) containing the sequence number \( n \) along with the source and destination’s IP address (in that case, fake address \( id \) and the server’s address);
output(\( SYN_n \)): the intruder sends \( SYN_n \) to the server.

Upon receiving the intruder’s SYN packet, the server processes it as follows:

store: the server allocates data structures in which packet \( SYN_n \) is stored;
makeACK: the server creates an acknowledgement packet \( ACK_n \) for \( SYN_n \);
makeSYN: the server generates a random number \( m \) and creates a synchronisation packet \( SYN_m \);

\( \text{output}(SYN_m, \text{ACK}_n) \): the server sends \( SYN_m \) and \( ACK_n \) to the fake IP address \( id \).

The intruder then repeats this attack with different fake identifiers and different SYN packets, but without completing the protocol run. This attack is possible because generating fake identifiers (IP addresses) and SYN packets require little resources. Resource exhaustion DoS occurs because the server allocates expensive data structures upon receiving a SYN packet (specified above as the action store). In order to analyse this DoS attack, we need to consider both the cost for the intruder to launch its attack and the cost for the server to process it. If the cost of the attack (i.e. \( \text{cost}(\text{newId}) + \text{cost}(\text{makeSYN}) + \text{cost}(\text{output}) \)) is much less than the cost of processing it (i.e. \( \text{cost}(\text{store}) + \text{cost}(\text{makeACK}) + \text{cost}(\text{makeSYN}) + \text{cost}(\text{output}) \)), then the protocol obviously has a flaw since an intruder could launch, with very few resources, multiple attacks that can waste a great amount of the server’s resources. If the wasted resource is over his capacity, then the server has no more resource available and it must deny any new legitimate protocol run.

Once the costly actions identified, not each of their occurrences should be considered as an attack; only those that are direct consequence of the intruder’s behaviour. Indeed, a server’s costly behaviour could have been caused by some other (honest) principal. In the case of TCP protocol, where only the action “store” is assumed to be costly, we prove (see Example 3) that the server’s costly behaviour (i.e. \( \text{store} \xrightarrow{} \text{makeACK} \xrightarrow{} \text{makeSYN} \xrightarrow{} \text{output} \)) is a direct consequence of the intruder’s attack (i.e. \( \text{newId} \xrightarrow{} \text{makeSYN} \xrightarrow{} \text{output} \)).

2 Security Protocol Process Algebra

In this section, we introduce a generic value-passing process algebra called Security Protocol Process Algebra (SPPA). It allows the specification of local function calls and introduces marker actions which are used to tag message exchanges between processes. Up to these extensions tailored just to fit to the ideas presented here, SPPA is similar to Milner’s CCS [23]. The purpose here is not to introduce a new process algebra but just to define a generic process algebraic framework as well-suited as possible to analyse cryptographic protocols.
2.1 Message Algebra

SPPA uses a message algebra that relies on disjoint syntactic categories of numbers, principal identifiers and variables respectively ranging over sets $\mathcal{N}$, $\mathcal{I}$ and $\mathcal{V}$. The set of terms $\mathcal{T}$ is constructed as follows:

$$ t ::= n \ (\text{number}) \mid id \ (\text{identifier}) \mid x \ (\text{variable}) \mid (t, t) \ (\text{pair}) $$

$$ \mid \{t\} \ (\text{encryption}) \mid [t] \ (\text{signature}) \mid h(t) \ (\text{hashing}) $$

For any term $t$, we denote $\text{fv}(t)$ the set of variables occurring in $t$ and we say that $t$ is a message whenever it contains no variable. The set of all messages is denoted by $\mathcal{M}$. For sake of clarity, we discriminate a subset $\mathcal{K} \subseteq \mathcal{M}$ of messages that may be used as encryption key. Note that the exact content of the set $\mathcal{K}$ depends on the cryptosystems used by the protocol.

We consider a finite set $\mathcal{F}$ of functions $f : \mathcal{M}^n \rightarrow \mathcal{M}$.\(^1\) If the function $f$ has arity $n \geq 1$, we write $f(x_1, \ldots, x_n)$ with $x_1, \ldots, x_n \in \mathcal{V}$ and we use $\text{dom}(f)$ to denote its domain.\(^2\) A function with arity $n = 0$ is called a generating function. They are extremely useful for the specification of security protocols requiring fresh nonces, fresh keys and random numbers, and for the specification of intruders generating fake messages. Given a generating function $f \in \mathcal{F}$, we use $\text{im}(f)$ to denote its image (range).\(^3\)

For instance, consider the following functions:

- $\text{pair}(a, a') = (a, a')$ (pairing function with domain $\mathcal{M} \times \mathcal{M}$);
- $\text{enc}(k, a) = \{a\}_k$ (encryption function with domain $\mathcal{K} \times \mathcal{M}$);
- $\text{hash}(a) = h(a)$ (hash function with domain $\mathcal{M}$);
- $\text{sign}(k, a) = [a]_k$ (signature function with domain $\mathcal{K} \times \mathcal{M}$);
- $\text{newMessage}(-)$ (message generator with image $\mathcal{M}$);
- $\text{newNumber}(-)$ (number generator with image $\mathcal{N}$);
- $\text{newId}(-)$ (identifier generator with image $\mathcal{I}$);
- $\text{newKey}(-)$ (key generator with image $\mathcal{K}$).

For practical purpose, we extend the pairing function to deal with any $n$-tuple; thus, we consider function $\text{pair}(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$.

---

\(^1\) For some $n \geq 0$, where $n$ denotes $f$’s arity.

\(^2\) $\text{dom}(f) \subseteq \mathcal{M}^n$.

\(^3\) $\text{im}(\text{new}) \subseteq \mathcal{M}$. 

7
2.2 Syntax of SPPA

We consider a finite set $C$ of public channels. Public channels are used to specify message exchanges between principals (commonly, there is one channel for every step of a protocol run). Every public channel $c$ has a predetermined domain $\text{dom}(c)$ of messages which can be sent and received over $c$, but, for simplicity purpose, we shall assume that $\text{dom}(c) = \mathcal{M}$ for every $c \in C$. Hence, any message can be sent or received over public channels. Intuitively, SPPA has two syntactic levels. On one hand, an agent intends to specify the internal behaviour of each principal. On the other hand, the interaction of principals over public channels is formalised through the notion of processes. Operational semantics is defined only for processes.

The agents of SPPA are constructed from the following grammar:

$$S ::= \emptyset \quad \text{(nil)} \mid \text{let } (x, y) = t \text{ in } S \quad \text{(pair splitting)}$$
$$\mid \overline{c}(t).S \quad \text{(output)} \mid \text{let } x = f(t) \text{ in } S \quad \text{(function call)}$$
$$\mid c(x).S \quad \text{(input)} \mid \text{case } t \text{ of } \{ x \} \nu \text{ in } S \quad \text{(decryption)}$$
$$\mid [t = t'] S \quad \text{(match)} \mid \text{case } t \text{ of } [t'] \nu \text{ in } S \quad \text{(signature verification)}$$
$$\mid S + S \quad \text{(sum)} \mid S|S \quad \text{(parallel composition)}$$

Whenever $f$ is a generating function, we usually write $\text{let } x = f(-) \text{ in } S$. Given an agent $S$, we define its set of free variables, denoted by $\text{fv}(S)$, as the set of variables $x$ appearing in $S$ that are not in the scope of an input prefix $c(x)$, a pair splitting $\text{let } (x, y) = t \text{ in } S$, a function call $\text{let } x = f(t) \text{ in } S$ or a decryption $\text{case } t \text{ of } \{ x \} \nu \text{ in } S$; otherwise the variable $x$ is said to be bound.

Given a free variable $x \in \text{fv}(S)$ and a term $t$, we consider the substitution operator $S[t/x]$ where every free occurrence of $x$ in $S$ is set to $t$. A closed agent is an agent $S$ such that $\text{fv}(S) = \emptyset$.

A SPPA principal is a couple $(S, id)$ where $S$ is a closed agent and $id \in \mathcal{I}$ is an identifier. The purpose of this notation is to relate a SPPA agent $S$ and its sub-agents, to their unique owner (principal) via its identifier $id$. When no confusion is possible, we often use $A$ as a reference to the principal $(S_A, id_A)$ where $S_A$ is the initial agent of $A$ i.e., the closed agent specifying the principal’s entire behaviour within the protocol. Moreover, we commonly make use of the identifier $id_A$ as a message containing its address, while we simply use $A$ to refer to the protocol’s entity. For simplicity, given $A_1 = (S_1, id)$ and $A_2 = (S_2, id)$ (they must have the same identifier) we often write:

8
Following a classic approach [24,6,11], security protocols are specified in SPPA as concurrent principals. Given a principal $A$, SPPA processes are constructed as follows:

$$P ::= A \text{ (principal)} \mid A \parallel P \text{ (protocol)}$$

$$\mid P \backslash L \text{ (restriction)} \mid P / O \text{ (O-observation)}.$$

where $L$ is a set and $O$ is a partial mapping (both to be clarified in Section 2.4 and 2.5).

**Example 1** The (one run) TCP’s three-way handshake connection establishment protocol is specified as SPPA process $A \parallel B$ with public channels $C = \{c_1, c_2, c_3\}$, and principals $A = (S_A, id_A)$ and $B = (S_B, id_B)$, where initial agents $S_A$ and $S_B$ are defined in Fig. 1 and Fig. 2, in which $S_{A'}$ and $S_{B'}$ are agents specifying the remainder of the TCP protocol.

$$S_A ::= \text{let } x_1 = \text{newNumber}(-) \text{ in let } x_2 = \text{makeSYN}(id_A, id_B, x_1) \text{ in}$$

$$\text{let } x_3 = \text{store}(x_2) \text{ in } c_1(x_2). c_2(x_4). \text{let } (x_5, x_6) = x_4 \text{ in}$$

$$\text{let } x_7 = \text{makeACK}(x_2) \text{ in } [x_7 = x_6] \text{ let } x_8 = \text{makeACK}(x_5) \text{ in}$$

$${\overline{c_3}}(x_8).S_{A'}$$

Fig. 1. TCP’s principal $A$ specified in SPPA

Note that SYN$_n$ and ACK$_n$ packets may be viewed as tuples $(\text{header}, id, id', n)$ from our message algebra where $id, id' \in \mathcal{I}$ are the source and the destination identifiers, and $n \in \mathcal{N}$ is a sequence number. For the purpose of this specification, these packets are formalised as the 4-tuples $\text{SYN}_n = (0, id, id', n)$ and $\text{ACK}_n = (1, id, id', n)$ (header “0” denotes a SYN packet, while header
$S_B := c_1(y_1). \text{let } (y_2, y_3, y_4, y_5) = y_1 \text{ in } [y_4 = id_B] \text{ let } y_6 = \text{store}(y_1) \text{ in }
\text{let } y_7 = \text{makeACK}(y_1) \text{ in let } y_8 = \text{newNumber}(-) \text{ in }
\text{let } y_9 = \text{makeSYN}(id_B, y_3, y_8) \text{ in let } y_{10} = \text{store}(y_9) \text{ in }
\text{let } y_{11} = \text{pair}(y_9, y_7) \text{ in }
\overline{c_2}(y_{11}). \overline{c_3}(y_{12}). \text{let } y_{13} = \text{makeACK}(y_9) \text{ in } [y_{13} = y_{12}] \ S'_B$

Fig. 2. TCP’s principal $B$ specified in SPPA

"1" denotes an ACK packet, although the header is larger in practice since it contains other information related to the protocol’s run). This specification requires the following functions:

- $\text{makeSYN}(id, id', n) = (0, id, id', n)$ (SYN packet constructor with domain $I \times I \times N$);
- $\text{store}(a) = a$ (storing function with domain $M$);
- $\text{makeACK}(0, id, id', n) = (1, id, id', n)$ (ACK packet constructor with domain $\{0\} \times I \times I \times N$).

Intuitively, $\text{store}(a)$ really stands for the address where the message is being stored. Since TCP does not rely on cryptographic manipulations, we only need to consider the set of functions

$$\mathcal{F} = \{\text{pair}, \text{newNumber}, \text{makeSYN}, \text{store}, \text{makeACK}\}.$$

A destination principal $B$ (e.g. server) may usually proceed to several connection establishments at the same time. If $N$ denotes the maximal number of concurrent half-open TCP connections allowed (before a reset), then a destination $B$ is specified by principal $B^N = (S_B \mid \ldots \mid S_B, id_B)$ ($N$ times). Therefore, the ($N$ runs) TCP connection establishment process is given by process $TCP ::= A \parallel B^N$.

2.3 Actions

The actions of SPPA are defined in Fig. 3, where $a \in M$.

For instance, function call $\text{enc}_{id_A}(a)$ stands for principal $A$ encrypting some message $a$ with some key $k$; output action $\overline{c_{id_A}}(\{a\}_k)$ stands for principal $A$ sending message $\{a\}_k$ over the public channel $c$; and decryption action $\overline{c_{id_A}}$ stands for principal $A$ successfully decrypting some message $\{a\}_k$. The silent action $\tau$ is used to express non-observable behaviours. We often use $C$ to denote both the set of public channels and the set of output and input actions.
Marker actions are introduced in an attempt to establish an annotation upon the semantics of a SPPA process; they do not occur in the syntax of processes and their specific semantics restrict their occurrence in order to tag communications between principals. In value-passing process algebra, communication is commonly expressed by replacing matching output action and input action by the silent action $\tau$, which leads to a great loss of information on the content of the exchanged values and the parties involved. A marker action has three parameters: a principal identifier, a channel and a message. Roughly speaking, the occurrence of an output marker action $\delta_{c\ id\ A}(a)$ stands for "the principal $A$ has sent message $a$ over the channel $c$", and the occurrence of an input marker action $\delta_{c\ id\ A}(a)$ stands for "the principal $A$ has received message $a$ over the channel $c$".

We write $\text{Act}$ to denote the set of all actions and we consider the set $\text{Act}_A \subseteq \text{Act}$ of actions that may be launched by some principal $A$, defined by:

$$\text{Act}_A = \{c_{id\ A}(a), \overline{c_{id\ A}(a)}, \overline{\delta_{c\ id\ A}(a)}, \overline{\delta_{c\ id\ A}(a)} \mid c \in C \text{ and } a \in M\}$$

$$\cup \{f_{id\ A}, \overline{f_{id\ A}} \mid f \in F\}$$

$$\cup \{\overline{\text{dec}_{id\ A}}, \text{signv}_{id\ A}, \overline{\text{fail}_{id\ A}}, \overline{\text{fail}_{id\ A}}, \overline{\text{fail}_{id\ A}}\}$$

### 2.4 Observation Criteria

An observation criterion is a mapping $O : \text{Act}^* \rightarrow \wp(\text{Act}^*)$ which intends to express equivalence between process behaviours. Two sequences of actions $\gamma_1$ and $\gamma_2$ are said to carry out the same observation $\alpha$ whenever $\alpha \in O(\gamma_1) = O(\gamma_2)$. Given a subset $L \subseteq \text{Act} \setminus \{\tau\}$, we consider the observation criterion $O_L$ defined as follows:

---

$\text{Act}^*$ denotes the set of every finite sequence of actions form $\text{Act}$ and $\wp(\text{Act}^*)$ denotes its power set.
Only behaviours from the set $L$ are observable through this observation criterion. In particular, we have a natural observation criterion $O_{Act_A}$, often denoted by $O_A$, describing the actions observable by a principal $A$.

Note that observation criterion $O_L$ is such that, given $\alpha_1, \alpha_2 \in Act$ and $\gamma \in Act^*$, $\alpha_1 \in O_L(\gamma)$ and $\alpha_2 \in O_L(\gamma)$ implies $\alpha_1 = \alpha_2$. Note that $\tau \in O_L(\gamma)$ implies that $\gamma$ is not observable from $O_L$, i.e. it contains no actions form the set $L$.

### 2.5 Semantics of SPPA

The operational semantics of SPPA processes is defined in Fig. 4 and Fig. 5, where $a \in M$, $L \subseteq Act$, $A, A', B$ are principals and $P, P', Q, Q'$ are processes. The **Sum**, **Parallel**, **Protocol** and **Synchronisation** rules are assumed to be both associative and commutative (i.e. process $P + (Q + R)$ behaves as $(P + Q) + R$, process $Q + P$ behaves as $P + Q$, etc.).

The **Output** rule allows to output messages over public channels, while the **Input** rule needs to consider every possible message that may be received over a public channel. The **Split** rule allows to extract pairs, and is extended, in a natural way, to deal with any $n$-tuple splitting. The **Function** rule allows the specification of local function calls made by principals, and the **Fail-Function** rule generates a fail action whenever a function is called on a message outside its domain. The **Decryption** rule allows to decrypt messages, and the **Fail-Decryption** rule generates a fail action whenever decryption is attempted with the wrong key. The **Signature-Verif** rule allows the validation of signed messages, although without recovering the signed message since it is assumed to be hashed, and the **Fail-Signv** rule generates a fail action whenever signature verification fails (either because of the wrong key or the wrong content). The **Match** rule allows the verification of (syntactic) equality between two messages and the **Fail-Match** deals with the case where the equality is not satisfied. (Note that every fail action leads to the nil process 0, although we could define special processes for different type of failure.)

The **Sum** and **Parallel** rules allows the specification of non-deterministic sum and parallel product of principals (with matching identifier). The **Protocol** and **Synchronisation** rules allows the specification of protocols, where the operator $|$ is similar to parallel product, but through which communication

---

5 $\epsilon$ denotes the empty sequence.
between principals is achieved (and forced) through public channels. When two processes $P$ and $Q$ communicate, they enter an intermediary state $P \# Q$ ("busy" state) that cannot be accessed otherwise. This formalisation of communication insures us that every output marker action is always followed by a matching input marker action. The Restriction rule interprets process $P \setminus L$ (where $L$ is a set of actions) as $P$ with the actions in $L$ forbidden. Finally, the Observation rule interprets the observation of a process through an observation criterion $O$, where the computation $P \xrightarrow{\gamma} P'$, for a sequence
Match \[ A \xrightarrow{\alpha} A' \]
\[ [a=a] \quad A \xrightarrow{\alpha} A' \]

Fail-Match \[ a \neq a' \]
\[ [a=a'] \quad A \xrightarrow{\text{fail}_A} 0 \]

Sum \[ A \xrightarrow{\alpha} A' \]
\[ A + B \xrightarrow{\alpha} A' \]

Parallel \[ A \xrightarrow{\alpha} A' \]
\[ A | B \xrightarrow{\alpha} A' | B \]

Protocol \[ P \xrightarrow{\alpha} P' \] and \[ \alpha \notin C \]
\[ P | Q \xrightarrow{\alpha} P' | Q \]

Synchronisation \[ P \xrightarrow{c_{idA}(a)} P' \] and \[ Q \xrightarrow{c_{idA'}(a)} Q' \]
\[ P | Q \xrightarrow{\delta_{idA}(a)} P' \# Q \] and
\[ P' \# Q \xrightarrow{\delta_{idA'}(a)} P' | Q' \]

Restriction \[ P \xrightarrow{\alpha} P' \] and \[ \alpha \notin L \]
\[ P \setminus L \xrightarrow{\alpha} P' \setminus L \]

Observation \[ P \xrightarrow{\gamma} P' \] and \[ \alpha \in \mathcal{O}(\gamma) \]
\[ P / \mathcal{O} \xrightarrow{\alpha} P' / \mathcal{O} \]

Fig. 5. Semantics of SPPA processes.

of actions \( \gamma = \alpha_0 \alpha_1 \ldots \alpha_n \in \text{Act}^* \), stands for the finite string of transitions \( P \xrightarrow{\alpha_0} P_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} P' \). For instance, given a subset of actions \( L \) and the observation criterion \( \mathcal{O}_L \), process \( P / \mathcal{O}_L \) behaves as \( P \) with the actions outside \( L \) ignored (set to \( \tau \)). The notion of \( \mathcal{O} \)-observation of a process is therefore aimed at defining the process resulting from its observation through the observation criterion \( \mathcal{O} \).

We say that process \( P' \) is a derivative of \( P \) if there is a computation \( P \xrightarrow{\gamma} P' \) for some \( \gamma \in \text{Act}^* \). The set of \( P \)'s derivatives is given by:

\[ \mathcal{D}(P) = \{ P' \mid \exists \gamma \in \text{Act}^*, \ P \xrightarrow{\gamma} P' \} \]
2.6 Observation-dependent Bisimulation

The concept of $\mathcal{O}$-bisimulation [3], called $\mathcal{O}$-congruence by Boudol [25], captures the notion of behavioural indistinguishability between processes through an observation criterion $\mathcal{O}$.

**Definition 1** Let $P$ and $Q$ be processes, and let $\mathcal{O}$ be an observation criterion.

1. An $\mathcal{O}$-simulation of $P$ by $Q$ is a relation $R \subseteq D(P) \times D(Q)$ such that
   - $(P, Q) \in R$;
   - For every action $\alpha \in \text{Act}$, if $(P_1, Q_1) \in R$ and $P_1 \xrightarrow{\alpha} P_2$, then there is a transition $Q_1 \xrightarrow{\gamma} Q_2$, with $\alpha \in \mathcal{O}(\gamma)$, such that $(P_2, Q_2) \in R$.

   In such a case, we write $P \preceq_\mathcal{O} Q$.

2. An $\mathcal{O}$-bisimulation between $P$ and $Q$ is a relation $R \subseteq D(P) \times D(Q)$ such that
   - $(P, Q) \in R$;
   - For every action $\alpha \in \text{Act}$, if $(P_1, Q_1) \in R$ and $P_1 \xrightarrow{\alpha} P_2$, then there is a transition $Q_1 \xrightarrow{\gamma} Q_2$, with $\alpha \in \mathcal{O}(\gamma)$, such that $(P_2, Q_2) \in R$;
   - For every action $\alpha \in \text{Act}$, if $(P_1, Q_1) \in R$ and $Q_1 \xrightarrow{\alpha} Q_2$, then there is a transition $P_1 \xrightarrow{\gamma} P_2$, with $\alpha \in \mathcal{O}(\gamma)$, such that $(P_2, Q_2) \in R$.

   In such a case, we write $P \simeq_\mathcal{O} Q$ and we say that processes $P$ and $Q$ are $\mathcal{O}$-bisimilar.

The best known examples of observation criteria are the ones defining Milner’s strong and weak bisimulations over process algebra CCS [23]. Both are special cases of criteria obtained from projections. Define $\text{Vis} = \text{Act} \setminus \{\tau\}$ the set of visible actions. Then, two sequences are equivalent through the weak criterion $\mathcal{O}_\text{Vis}$ if their visible content is the same. Moreover, by considering criterion $\mathcal{O}_\text{Act}$, one gets the strong criterion through which each sequence of actions is observable and distinguishable from any other sequence. Weak bisimulation can therefore be seen as $\mathcal{O}_\text{Vis}$-bisimulation and (strong) bisimulation can be seen as $\mathcal{O}_\text{Act}$-bisimulation.

In the following section, we define a bisimulation-based semantics for admissible interference. As we shall see, $\mathcal{O}$-bisimulation provides a suitable theoretical framework for expressing non interference and admissible interference properties. In particular, using bisimulation, instead of trace equivalence, in the context of a verification method based on equivalence-checking, provides algorithms with lower computational complexity.
3 Admissible Interference

In this section, we give a brief introduction to the information flow property called *bisimulation-based non-deterministic admissible interference* (BNAI) [3], along with an equivalence-checking-based method for the analysis of cryptographic protocols, extending, in a non-trivial way, the non interference-based approach presented by Focardi, Gorrieri & Martinelli [26]. This security formalisation of admissible interference generalises a similar trace-based results obtained by Mullins [9] into the finer notion of observation-based bisimulation. The basic idea of the method is to prove that no intruder can interfere with the protocol. Like its trace-based version, BNAI admits interference (information flow) between intruders and principals only through predetermined actions. Confidentiality and authentication for security protocols were defined in terms of BNAI in previous papers [3]. Several non interference-based methods have been designed to analyse cryptographic protocols [6,5]. Advantages of admissible interference-based methods over non interference-based methods were highlighted in a previous paper [3]. For instance, in some cases, admissible interference allows harmless interference, i.e. interference that does not correspond to a successful attack, to be discarded at the specification level, rather than screening it manually from the by-products of the verification process.

3.1 Bisimulation-based strong non-deterministic non-interference

Formally, given two disjoint subsets $K$ and $L$ of the set $Vis$ of visible actions, $K$ is said to cause interference on $L$ (within some process $P$) whenever there are actions from $K$ causing actions from $L$ which might have not occurred otherwise. For instance, in Fig. 6, we see that the action $\alpha_1$ causes interference on the action $\alpha_2$ in the process $Q$, but not in the process $P$.

\[ P \xrightarrow{\alpha_1, \alpha_2} Q \xrightarrow{\alpha_1, \alpha_2} \]

Fig. 6. SPPA processes $P$ and $Q$.

The following formulation of non-interference (with respect to $L$ and $K$) requires that a process $O_{K\cup L}$-simulates its $O_L$-observation.

**Definition 2 (BSNNI)** Process $P$ satisfies Bisimulation-based strong non-deterministic non-interference if

\[ P \simeq_{O_L} P \setminus K. \]
From Fig. 6, assuming that $\alpha_1 \in K$ and $\alpha_2 \in L$, we see that the process $P$ satisfies BSNNI, but not the process $Q$. When $K$ holds for the set $Hi$ of high-level actions and $L$ holds for the set $Lo$ of low-level actions, it is not difficult to see that this property coincides with bisimulation-based strong non-deterministic non-interference as proposed by Focardi & Gorrieri [13].

3.2 Bisimulation-based non-deterministic admissible interference

Given a set $\Gamma \subseteq V is$ of downgrading actions, admissible interference refers to the information flow property which requires that systems admit information flow from $K$ behaviours to $L$ behaviours only through specifics downgrading actions. To capture this property, Mullins [9] proposed that any process $P'$ derived from $P$ and executing no downgrading action be required to satisfy non-interference. More precisely, for $P$ to satisfy intransitive non-interference [8], process $P' \backslash \Gamma$ must satisfy non-interference for every derivative $P' \in D(P)$. Rephrasing it in the context of BSNNI as the non-interference property yields the following definition.

**Definition 3 (BNAI)** Process $P$ satisfies bisimulation-based non-deterministic admissible interference if

$$\forall_{P' \in D(P)} P' \backslash \Gamma \simeq_{OL} P' \backslash (\Gamma \cup K).$$

The previous definition is an algebraic characterisation of admissible interference in terms of $OL$-bisimulation. Therefore, it is also our main proof method for BNAI.

4 Detecting Denial of Service Vulnerabilities in Security Protocols

In this section, we investigate DoS attacks in which an intruder causes a resource exhaustion to a defender (e.g. server) through the steps of a security protocol (most commonly authentication protocols). In such an attack, at any step of the protocol (but mostly at the beginning), the intruder usually sends a fake message in order to waste the defender’s resource processing it. In this context, we mainly focus on attacks that require little effort from the intruder and cause great waste of resources to the defender. If the defender may simultaneously proceed several requests (protocol runs), the intruder can then repeat its attack up to the point of causing a resource exhaustion; the defender then has to refuse any other protocol run request, including those from honest principals. This type of DoS, which includes distributed DoS, is formalised as $N$ copies of an enemy process initiating, simultaneously, protocol
runs with a principal (defender) which may deal with up to \( N \) simultaneous requests. Since the whole attack is based on a single flaw in the protocol, it is usually enough to verify whether a single enemy process may interfere on high-cost actions of the defender by only using its low-cost actions. This single resource exhaustion flaw, once multiplied by \( N \), may lead to a lethal DoS attack.

The term *interfere* refers to the non-interference property as defined by Focardi & Gorrieri [13]. This concept is crucial in order to establish a causal dependency between the intruder’s behaviours and the defender’s behaviours: the occurrence of an high-cost actions must be a consequence of the intruder’s actions, and not of some other protocol run, in order to conclude that there is indeed a DoS attack. However, we allow any interference coming from an intruder behaving properly. Such honest behaviours include initiating a real protocol run (with its own identifier and no fake message) and replying properly to a protocol run invitation. This assumption of allowing honest intruder’s behaviours is crucial in order to view the intruder as any legitimate user. Admissible interference helps us achieve this goal by allowing an enemy process to cause harmless interference on the protocol through predetermined actions called *admissible attacks*.

The main contribution of this paper is an equivalence-checking method for the validation of security protocols against resource exhaustion attacks. It is based on an information flow properties called *impassivity* inspired by BNAI and verified through \( \mathcal{O} \)-bisimulation.

4.1 Specification of Enemy Processes

Given a security protocol, we are mostly interested in studying its behaviour while running in a hostile environment. More precisely, we want to make sure that the protocol acts “correctly” in any given critical situation. Since such hostile environments need to be related to our process algebra, they are specified as enemy processes attempting to attack the protocol through its public channels. We assume that every enemy process is related to a unique enemy identifier \( id_E \in I \) and a unique set of enemy actions \( Act_E \). We also consider a set of *admissible attacks*, denoted by \( \Gamma \), which is a subset of \( Act_E \). The exact content of the set \( \Gamma \) depends on the specification of the protocol under study.

In order to achieve a resource exhaustion attack, an intruder commonly needs to initiate several protocol runs, each exploiting the same flaw. For this reason, we only consider attacks where the intruder is the protocol’s initiator. Thus, the interaction of an enemy process \( E \) with some protocol \( P := B \parallel A_1 \parallel \ldots \parallel A_n \) is written as the SPPA process \( P_E := E \parallel B \parallel A_1 \parallel \ldots \parallel A_n \),
where $B$ is the defender (server) and the $A_i$ are others (honest) principals. Therefore, process $P_E \setminus \Gamma$ stands for the protocol run in which the enemy’s honest behaviours are removed; only potentially dangerous attacks remain.

**Example 2** The SYN flooding attack on the TCP connection protocol, specified as the SPPA process $TCP := A \parallel B^N$ (see Example 1), is parsed by the enemy process $E^N = (S_E \ldots | S_E, id_E) \ (N \text{ times})$ where

$$S_E := \text{let } x_1 = \text{newId}(-) \text{ in let } x_2 = \text{newNumber}(-) \text{ in let } x_3 = \text{makeSYN}(x_1, id_B, x_2) \text{ in } c_1(x_3).0$$

The SYN flood attack is therefore formalised as process $TCP_E := E^N \parallel B^N \parallel A$. Moreover, the set $\Gamma$ of admissible attacks contains any output and input markers actions corresponding to honest protocol runs, i.e. containing a SYN packet or a ACK packet with $id_E$ as either its source’s identifier or destination’s identifier. Formally, we have

$$\Gamma = \{ \delta_{id_E}^3(SYN_1^n), \delta_{id_E}^3(SYN_2^n) \mid id \in I \text{ and } n \in N \}$$
$$\cup \{ \delta_{id_E}^3(SYN_1^n, ACK_1^m), \delta_{id_E}^3(SYN_2^n, ACK_2^m) \mid id \in I \text{ and } m, n \in N \}$$
$$\cup \{ \delta_{id_E}^3(ACK_1^m), \delta_{id_E}^3(ACK_1^m) \mid id \in I \text{ and } m \in N \}$$

with $SYN_1^n = (0, id_E, id, n)$, $ACK_1^m = (1, id, id_E, m)$, $SYN_2^n = (0, id, id_E, n)$ and $ACK_2^m = (1, id_E, id, m)$.

### 4.2 Cost Functions

The following approach of assigning cost to actions was inspired by Meadows’ cost-based framework [18]. In order to compare resource spending, we consider two ordered sets of costs $\langle C_{cpu}, < \rangle$ (for CPU resources) and $\langle C_{mem}, < \rangle$ (for memory resources). Moreover, we consider two cost functions

$$\rho_{cpu} : Act \rightarrow C_{cpu} \quad \text{and} \quad \rho_{mem} : Act \rightarrow C_{mem}$$

where $\rho_{cpu}(\alpha)$ and $\rho_{mem}(\alpha)$ respectively stands for the quantity of CPU resources, namely the CPU cost, and the quantity of memory resources, namely the memory cost, required to execute action $\alpha$. (Note that function $\rho_{mem}$ could be extended to deal with sequences of actions, in which memory resources are allocated and released, and therefore cope with accumulative cost. However, this extension requires non trivial modifications to the verification method presented below.)
Given a principal $B$ (defender) within a protocol, we consider its CPU capacity $\text{CPU}_B \in \mathcal{C}_{\text{cpu}}$ which stands for the defender’s CPU resource capacity:

running simultaneously $N$ actions of CPU cost greater than $\text{CPU}_B$ may cause $B$ a CPU resource exhaustion DoS.

Similarly, we consider its memory capacity $\text{MEM}_B \in \mathcal{C}_{\text{mem}}$ which stands for the defender’s memory resource capacity:

running simultaneously $N$ actions of memory cost greater than $\text{MEM}_B$ may cause $B$ a memory resource exhaustion DoS.

We also consider the set $\text{Act}_{\text{costly}}$ of $B$’s costly actions define by

$$\text{Act}_{\text{costly}} = \{ \alpha \in \text{Act}_B \mid \rho_{\text{cpu}}(\alpha) > \text{CPU}_B \text{ or } \rho_{\text{mem}}(\alpha) > \text{MEM}_B \}.$$ 

Moreover, we consider intruder’s CPU capacity $\text{CPU}_E \in \mathcal{C}_{\text{cpu}}$ and memory capacity $\text{MEM}_E \in \mathcal{C}_{\text{mem}}$ defined respectively as follows:

the enemy process $E$ may only execute actions of CPU cost lest or equal to $\text{CPU}_E$

and

the enemy process $E$ may only execute actions of memory cost lest or equal to $\text{MEM}_E$.

Hence, an intruder that may only launch low-cost attacks is formalised as an enemy process $E$ in which every transition $E' \xrightarrow{\alpha} E''$, with $E' \in \mathcal{D}(E)$, is such that $\rho_{\text{cpu}}(\alpha) \leq \text{CPU}_E$ and $\rho_{\text{mem}}(\alpha) \leq \text{MEM}_E$. In that case, we say that $E$ respects its capacity. Obviously, the values of the capacities of each principal depend on many practical factors that must be taken into consideration during the specification phase of the protocol.

For the TCP protocol, we assume that actions $\text{makeACK}_id$ (along with their corresponding fail actions $\text{fail}^{\text{makeACK}}_id$) have the greatest CPU cost, although we do not assume that their CPU cost is greater than neither $B$’s capacity $\text{CPU}_B$, nor $E$’s capacity $\text{CPU}_E$. However, we assume that actions $\text{store}_id$ have the greatest memory cost and that their cost exceeds both $B$’s capacity $\text{MEM}_B$ and $E$’s capacity $\text{MEM}_E$. Therefore, memory resource exhaustion may only occur whenever $B$ stores data.
4.3 Robustness Against Denial of Service

In the following, we introduce an information flow property, called *impassivity*, formalising robustness of security protocols against DoS, more precisely against both CPU and memory resource exhaustion. Given a server $B$, impassivity requires that no enemy process respecting its capacity may cause inadmissible interference on costly actions $\alpha \in \text{Act}_{\text{costly}}$.

**Definition 4 (Impassivity)** Let $P$ be a protocol and let $E$ be an enemy process respecting its capacity. Put $O_{\text{costly}} = O_{\text{Act}_{\text{costly}}}$ and $O_{\text{costly} \cup E} = O_{\text{Act}_{\text{costly}} \cup \text{Act}_E}$.

1. $P$ is impassive with respect to $E$ if
   \[ \forall Q \in \mathcal{D}(P_E) \quad Q \setminus \Gamma \simeq_{O_{\text{costly}}} Q \setminus \text{Act}_E. \]
2. $P$ is impassive if $P$ is impassive with respect to every enemy process respecting its capacity.

From this definition, it is straightforward to see that a protocol $P$ is impassive w.r.t $E$ if and only if process $P_E$ satisfies BNAL, with $K = \text{Act}_E$, $L = \text{Act}_{\text{costly}}$ and $\Gamma$ is the set of admissible attacks (using notation introduced in Section 3).

Given an enemy process $E$ respecting its capacity, impassivity w.r.t. $E$ is satisfied whenever the protocol in which $E$ attempts to attack $B$ is bisimilar, in terms of costly actions, to the protocol in which $E$’s actions are removed. Notice that the definition of impassivity suffers from a universal quantification over enemy processes. An attractive practical alternative to this problem consists on approximating this universal quantifier using a generic enemy process and verifying impassivity w.r.t. this process. However, we are mostly interested in robustness to DoS only with particular enemy processes, and therefore verify whether a protocol is vulnerable to specifics DoS attacks (e.g. the SYN flooding). This approach to static analysis is justified by the fact that we really want to verify if some protocol is robust against “weak” intruders; we already know that sufficiently strong intruders will be able to achieve DoS.

**Remark 5** From the family of impassivity properties obtained when considering every enemy processes respecting its capacity, we may define a robustness pre-order $\preceq$ over protocols. Given two protocols $P$ and $Q$ (specified in SPPA), we say that $P$ is more robust to DoS than $Q$ if for every enemy processes $E$ respecting its capacity, we have

\[ Q \text{ is impassive w.r.t. } E \implies P \text{ is impassive w.r.t. } E. \]

In that case, we write $Q \preceq P$. 

21
Example 3 Consider the enemy process $E^N$ defined in Example 2. This enemy process that achieves the SYN flooding attack respects its capacity since there is no storing action. From Definition 4, we may conclude that the TCP connection protocol does not satisfy impassivity w.r.t. $E^N$, hence it does not satisfy impassivity. More precisely, given the set $\Gamma$ of admissible attacks defined in Example 2, we have

$$(E^N \parallel B^N) \setminus \Gamma \not\subseteq\mathcal{O}_{\text{costly}} (E^N \parallel B^N) \setminus \text{Act}_E.$$ 

Indeed, we see that process $(E^N \parallel B^N) \setminus \Gamma$ can compute the sequence of actions

$$\gamma = \text{newId}_{id_E} \text{newNumber}_{id_E} \text{makeSYN}_{id_E} \delta_{id_E}^{-1}(SYN_n) \delta_{id_B}^{-1}(SYN_n) \text{split}_{id_B} \text{store}_{id_B}$$

with $SYN_n = (0, id, id_B, n)$ for some $id \in I$ and $n \in N$. However, the process $(E^N \parallel B^N) \setminus \text{Act}_E$ cannot execute $\gamma$ since the first actions belongs to $\text{Act}_E$. From the fact that $\text{store}_{id_B} \in \mathcal{O}_{\text{costly}}(\gamma)$, we may conclude therefore that process $((E^N \parallel B^N) \setminus \Gamma)/\mathcal{O}_{\text{costly}}$ can execute the action $\text{store}_{id_B}$ while $((E^N \parallel B^N) \setminus \text{Act}_E)/\mathcal{O}_{\text{costly}}$ cannot.

Notice that $\gamma$ is the sequence of actions we identified in Section 1.1 as the SYN flooding attack. We just showed that this computation is indeed an attack, i.e. the costly action $\text{store}_{id_B}$ is a direct consequence of the intruder’s non admissible behaviour corresponding the sequence of actions

$$\gamma' = \text{newId}_{id_E} \text{newNumber}_{id_E} \text{makeSYN}_{id_E} \delta_{id_E}^{-1}(SYN_n).$$

5 Application to the 1KP Secure Electronic Payment Protocol

The family of protocols $iKP$ ($i = 1, 2, 3$) for secure electronic payments over the Internet was developed by IBM Research Division [27,28]. The protocols implement credit card based transactions between a buyer and a seller by using the existing financial network for clearing and authorisation. All three $iKP$ protocols follow the same steps and only differ over the content of the message exchanged at each step. They use public-key encryption, strong collision-resistant one-way hashing function and public-key signature. The public and secret keys of a principal $A$ are respectively denoted by $pk_A$ and $sk_A$. For the purpose of this paper, we shall only study 1KP, the simplest of the trio, which
proceeds as follows:

Message 1: \[ A \xrightarrow{\text{salt}_A, h(r_A, \text{ban}_A)} B \]
Message 2: \[ B \xrightarrow{\text{clear}} A \]
Message 3: \[ A \xrightarrow{\{\text{slip}\}_{\text{pk}_{ACQ}}} B \]
Message 4: \[ B \xrightarrow{\text{clear}, h(\text{salt}_A, \text{desc}), \{\text{slip}\}_{\text{pk}_{ACQ}}} ACQ \]
Message 5: \[ ACQ \xrightarrow{\text{resp}, [\text{resp}, h(\text{common})]_{\text{sk}_{ACQ}}} B \]
Message 6: \[ B \xrightarrow{\text{resp}, [\text{resp}, h(\text{common})]_{\text{sk}_{ACQ}}} A \]

where common = (price, id_B, tid_B, n_B, h(r_A, ban_A), h(salt_A, desc)), clear = (id_B, tid_B, n_B, h(common)) and slip = (price, h(common), ban_A, r_A, exp_A).

First, the buyer A sends to the seller B a random number salt_A and a second random number r_A hashed with the buyer’s account number ban_A (e.g. credit card number). Secondly, the seller B replies with his identifier, the transaction’s identifier tid_B, a fresh nonce n_B, and the hashed value of the field common, where price and desc are the amount and the description of the purchase. Thirdly, the buyer A sends to B the field slip encrypted with the acquirer’s public key pk_{ACQ}, where exp_A is the expiration date associated to ban_A. The seller B may now request an authorisation clearance from the acquirer ACQ for the payment by forwarding the message just received from A along with the field clear and the hashed message h(salt_A, desc). The acquirer ACQ then proceeds as follows:

1. ACQ makes sure that the values from clear were not used in a previous request;
2. ACQ decrypts \{\text{slip}\}_{\text{pk}_{ACQ}} and obtains \text{slip};
3. ACQ checks whether the h(common) from clear matches the one from \text{slip};
4. ACQ re-constructs common, computes h(common) and checks whether it matches the one from clear;
5. ACQ sends ban_A and exp_A to the credit card organisation’s clearing and authorisation system in order to obtain an on-line authorisation for the payment.

Once this authorisation procedure is finished, the acquirer ACQ sends to the seller B the response resp received from the authorisation system along with h(common) and resp signed together with ACQ’s secret key. Finally, B checks the acquirer’s signature and forwards the previous message to A.
Let $A = (S_A, id_A)$, $B = (S_B, id_B)$ and $ACQ = (S_{ACQ}, id_{ACQ})$ be the principals of the 1KP protocol, where initials agents $S_A$, $S_B$ and $S_{ACQ}$ are respectively depicted in Fig. 7, Fig. 8 and Fig. 9.

For this specification, we assume that $F = \{\text{newNumber, pair, hash, enc, sign, replay, clearing}\}$.

Note that 1KP requires two hashing functions, one with a single argument and a second with two arguments, but, for simplicity purposes, our specification uses a unique function name. The function $\text{replay}(id, tid, n, h(a))$ returns 1 if there were no previous request with same parameters, and fails otherwise (thus its domain depends on the current stored messages of the acquirer). In addition, we assume that function “replay” stores its input, thus requires memory resources. The function $\text{clearing}(\text{ban, exp, price}) = \text{resp}$ validates the values $\text{ban, exp, price}$ in order to get an authorisation for the payment. It returns $\text{ok}$ (or 1) if the validation is successful, and $\text{nok}$ (or 0) otherwise. The one run specification of 1KP is given by the SPPA process $P := A \parallel B \parallel ACQ$.

$S_A := \text{let } r = \text{newNumber}(−) \text{ in let } x_1 = \text{hash}(r, \text{ban}_A) \text{ in}
\text{let } salt = \text{newNumber}(−) \text{ in let } x_2 = \text{pair}(salt, x_1) \text{ in}
\text{c}_1(x_2). \text{c}_2(x_3). \text{let } (x_{31}, x_{32}, x_{33}, x_{34}) = x_3 \text{ in}
\text{let } x_4 = \text{hash}(salt, \text{desc}) \text{ in}
\text{let } \text{common} = \text{pair}(\text{price}, x_{31}, x_{32}, x_{33}, x_{34}, x_1, x_4) \text{ in}
\text{let } x_5 = \text{hash}(\text{common}) \text{ in}
\left[x_5 = x_{34}\right] \text{let } \text{slip} = \text{pair}(\text{price}, x_5, \text{ban}_A, r, \text{exp}_A) \text{ in}
\text{let } x_6 = \text{enc}(pk_{ACQ}, \text{slip}) \text{ in}
\text{c}_3(x_6). \text{c}_6(x_7). \text{let } (x_{71}, x_{72}) = x_7 \text{ in}
\text{let } x_8 = \text{pair}(x_{71}, x_5) \text{ in}
\text{case } x_{72} \text{ of } \left[x_8\right]_{sk_{ACQ}} \text{ in } 0$

Fig. 7. 1KP’s buyer specified in SPPA

For DoS purpose, we are mainly interested in the $N$ run specification of 1KP. It is given by the SPPA process $P^N := A \parallel B_1 \parallel \ldots \parallel B_N \parallel ACQ^N$ where the $B_i = (S_B, id_B)$, for $i = 1, \ldots, N$, are different sellers (with same behaviour but distinct identifiers) and $ACQ^N = (S_{ACQ} \parallel \ldots \parallel S_{ACQ}, id_{ACQ})$. This
\[ S_B ::= c_1(y_1). \text{let} (y_{11}, y_{12}) = y_1 \text{ in } \text{let} y_2 = \text{hash}(y_{11}, \text{desc}) \text{ in } \]

\[ \text{let } n = \text{newNumber}(-) \text{ in } \]

\[ \text{let } common = \text{pair}(\text{price}, id_B, tid_B, n, y_{12}, y_2) \text{ in } \]

\[ \text{let } y_3 = \text{hash}(\text{common}) \text{ in let } clear = \text{pair}(id_B, tid_B, n, y_3) \text{ in } \]

\[ \overline{c_2}(\text{clear}). \ c_3(y_4). \text{let } y_5 = \text{pair}(\text{clear}, y_2, y_4) \text{ in } \]

\[ \overline{c_4}(y_5). \ c_5(y_6). \text{let } (y_{61}, y_{62}) = y_6 \text{ in } \]

\[ \text{let } y_7 = \text{pair}(y_{61}, y_3) \text{ in case } y_{62} \text{ of } [y_7]_{sk_{ACQ}} \text{ in } \overline{c_6}(y_6). 0 \]

Fig. 8. 1KP’s seller specified in SPPA

\[ S_{\text{ACQ}} ::= c_4(z_1). \text{let} (z_{11}, z_{12}, z_{13}) = z_1 \text{ in let } z_2 = \text{replay}(z_{11}) \text{ in } \]

\[ [z_2 = 1] \text{let} (z_{111}, z_{112}, z_{113}, z_{114}) = z_{11} \text{ in } \]

\[ \text{case } z_{13} \text{ of } \{ z_3 \}_{pk_{\text{ACQ}}} \text{ in } \]

\[ \text{let } (z_{31}, z_{32}, z_{33}, z_{34}, z_{35}) = z_3 \text{ in } \]

\[ [z_{114} = z_{32}] \text{let } z_4 = \text{hash}(z_{34}, z_{33}) \text{ in } \]

\[ \text{let } common = \text{pair}(z_{31}, z_{111}, z_{112}, z_{113}, z_4, z_{12}) \text{ in } \]

\[ \text{let } z_5 = \text{hash}(\text{common}) \text{ in } \]

\[ [z_5 = z_{114}] \text{let } resp = \text{clearing}(z_{33}, z_{35}, z_{31}) \text{ in } \]

\[ \text{let } z_6 = \text{pair}(resp, z_5) \text{ in let } z_7 = \text{sign}(sk_{\text{ACQ}}, z_6) \text{ in } \]

\[ \text{let } z_8 = \text{pair}(resp, z_7) \text{ in } \overline{c_5}(z_8).\text{ACQ} \]

Fig. 9. 1KP’s acquirer specified in SPPA

specification of \( ACQ^N \) allows the acquire to run up to \( N \) simultaneous protocol runs.

5.2 1KP’s Costly Actions

Assigning costs to actions is always problematic since it depends on many local factors like the complexity of the encryption scheme, the hashing function and the signature scheme. For the purpose of this paper, we assume that signature, signature verification and clearing are the most CPU costly actions. In fact, we assume that those actions exceeds the principals’ CPU capacity.
i.e., $\rho_{cpu}(\alpha) > CPU_B, CPU_{ACQ}$ whenever

$$\alpha \in \{\text{sign}_{id_B}, \text{sign}_{id_B}, \text{sign}_{id_{ACQ}}, \text{sign}_{id_{ACQ}}, \text{clear}_{id_{ACQ}},$$

$$\text{fail}^{\text{sign}}_{id_B}, \text{fail}^{\text{sign}}_{id_B}, \text{fail}^{\text{sign}}_{id_{ACQ}}, \text{fail}^{\text{sign}}_{id_{ACQ}}, \text{fail}^{\text{clear}}_{id_{ACQ}}\}.$$ \]

Moreover, we assume that the intruder has less memory capacity, enabling an enemy process respecting its capacity from doing any hashing, encryption, decryption, signing, nor signature verification. Thus $\rho_{cpu}(\alpha) > CPU_E$ whenever

$$\alpha \in \{\text{hash}_{id_E}, \text{enc}_{id_E}, \text{dec}_{id_E}, \text{sign}_{id_E}, \text{sign}_{id_E},$$

$$\text{fail}^{\text{hash}}_{id_E}, \text{fail}^{\text{enc}}_{id_E}, \text{fail}^{\text{dec}}_{id_E}, \text{fail}^{\text{sign}}_{id_E}, \text{fail}^{\text{sign}}_{id_E}\}.$$ \]

For the memory resource spending, only the action replay$_{id_{ACQ}}$ requires memory for storing data. Thus, we assume that it is the only action which exceeds the memory capacity of principals i.e.,

$$\rho_{mem}(\text{replay}_{id_{ACQ}}) > MEM_{ACQ}.$$ \]

5.3 Denial of Service Attack on 1KP

Consider the enemy process $E^N = (S_E \parallel \ldots \parallel S_E, id_E)$, where

$$S_E ::= \text{let } x_1 = \text{newMessage}(-) \text{ in let } x_2 = \text{pair}(x_1, x_1) \text{ in}$$

$$c_1(x_2), c_2(x_3), c_3(x_1), c_4(x_4), c_5(x_2).0.$$ \]

This enemy process respects its capacity since it has none the costly actions identified above. Furthermore, we see that (N runs) 1KP protocol $P^N$ does not satisfy impassivity w.r.t. $E$ since

$$P_E \setminus \Gamma \not\equiv_{P_{costly}} P_E \setminus \text{Act}_E.$$ \]

Indeed, our enemy process may pursue a DoS attack on the 1KP protocol as follows:

1. $E$ generates a fake message $a$ and sends the pair $(a, a)$ to $B$;
2. $B$ processes the received data (without noticing the fake message) and replies to $E$;
3. $E$ sends back $a$ which is again processed by $B$;
4. $B$ starts the authentication procedure by sending a message to $ACQ$ over the public channel $c_4$;
(5) $E$ intercepts this message and replies to $B$ (impersonating $ACQ$) another pair of fake message $(a, a)$; 

(6) Upon receiving this message, $B$ executes a costly signature verification procedure to check the authenticity of this last message (which will obviously fail).

This DoS attack is detected by the fact that marker action $\delta_{id}^{\text{sign}}(a, a)$ causes interference on the fail action $\text{fail}_{id}^{\text{sign}}$ (from the failed signature verification case $a$ of $[y_7, sk_{ACQ}$ in]) which has a CPU cost exceeding $B$’s capacity $\text{CPU}_B$. Hence, 1KP fails to satisfy impassivity w.r.t. $E$. The attack discovered through impassivity is interesting since the process enemy $E$ uses different sellers $B_1, \ldots, B_N$ to launch its DoS attack. Hence, the $B_i$ unconsciously do almost all the job for $E$!

6 Future and Related Works

This paper presents a method based on admissible interference for detecting DoS vulnerabilities in security protocols. It uses process algebra SPPA which allows the specification of local function calls as visible actions. SPPA also gives, through markers actions, a clearer view of communication between principals. Using a cost-based extension of SPPA, we introduce an information flow property called impassivity which detects whenever an enemy process may cause interference, using its low-cost actions, on high-cost actions of other principals. It is based on the fact that such interference may lead to an attack on the protocol by exploiting this single flaw several times, thus causing DoS through resource exhaustion. We offer a sound and complete equivalence-checking proof method for impassivity based on observant-dependant bisimulation. Complete illustrations of this method are given on the TCP connection protocol and the 1KP Secure Electronic Payment Protocol. Our analysis of TCP detects the well-known SYN flooding attack, while our analysis of 1KP reveals a resource exhaustion vulnerability.

The cost-based framework introduced in this paper allows an attribution of cost which depends on the capabilities of the different principals. For instance, if we want to validate DoS robustness with respect to a strong intruder, we may require that $\rho_{\text{cpu}}(\text{enc}_{id}) \leq \text{CPU}_E$ while $\rho_{\text{cpu}}(\text{enc}_{id}) > \text{CPU}_B$, i.e. encryption is over the server’s CPU capacity (may lead to resource exhaustion) but not over the intruder’s CPU capacity. Another important issue about our cost framework is to determine whether fails actions should have the same cost as their corresponding (non-fail) actions. As example, we may wonder if a failed decryption should cost as much as a successful one, i.e. $\rho_{\text{cpu}}(\text{dec}_{id}) = \rho_{\text{cpu}}(\text{fail}_{id}^{\text{dec}})$. Our cost model has the liberty to cope with both situations.
The specification of security protocols and their validation against DoS vulnerabilities often requires viewing function generating symbolic values such random numbers, fresh nonces, fresh keys, fake address and fake messages. Following this approach, a symbolic extension of SPPA able to handle symbolic values was introduced in a previous paper [29]. The main idea behind this approach is to assign to each SPPA process a formula describing the symbolic values conveyed by its semantics. In such symbolic processes, called constrained processes, the formulae are drawn from a decidable logic based on SPPA’s message algebra. The symbolic operational semantics of a constrained process is then established through a symbolic operational semantics in which formulae are updated by adding restrictions over the symbolic values, as required for the process to evolve. In the same paper [29], we also proposed a bisimulation equivalence between constrained processes which amounts to a generalisation of Milner’s bisimulation between value-passing processes. A sound and complete symbolic bisimulation method for constructing the bisimulation between two constrained processes is also provided.

Since most resource exhaustion DoS attacks occur relatively early and usually rely on a victim allocating costly data structures, we plan on extending our method to cope with accumulative memory cost. We feel that this future extension of our cost-based framework could be easily achieved through a generalisation of our memory cost function \( \rho_{\text{mem}} \) able to give the memory cost of any sequence of actions, including sequence of actions in which memory resource are released. We are presently working on a model based on semirings for describing memory allocations and deallocations. This model provides an algorithm to analyse whether a security protocol is subjected to memory exhaustion DoS.

A formalisation of robustness to DoS using admissible interference was proposed by the author [30] in the context of discrete event systems (DES). This formal method to detect denial vulnerabilities in security protocols is based on a property called intransitive non-interference (INI), extending so to a new application domain for the theory of supervisory control of DES.

**Acknowledgement:** Many thanks to prof. J. Mullins who helped me to write the first version of this paper.

**References**


