Journal of Universal Computer Science, vol. 9, no. 11 (2003), 1350-1369
submitted: 14/6/03, accepted: 5/9/03, appeared: 28/11/03

An Information Flow Method to Detect Denial of Service Vulnerabilities

Stéphane Lafrance
(École Polytechnique de Montréal, Canada)
stephane.lafrance@polymtl.ca

John Mullins
(École Polytechnique de Montréal, Canada)
john.mullins@polymtl.ca

Abstract: Meadows recently proposed a formal cost-based framework for the analysis of denial of service, showing how to formalize some existing principles used to make cryptographic protocols more resistant to denial of service by comparing the cost to the defender against the cost to the attacker. The first contribution of this paper is to introduce a new security property called *impassivity* designed to capture the ability of a protocol to achieve these goals in the framework of a generic value-passing process algebra called *Security Process Algebra* (SPPA) extended with local function calls, cryptographic primitives and special semantic features in order to handle cryptographic protocols. *Impassivity* is defined as an information flow property founded on *bisimulation-based non-deterministic admissible interference*. A sound and complete proof method for impassivity is provided. The method extends previous results of the authors on bisimulation-based non-deterministic admissible interference and its application to the analysis of cryptographic protocols. It is illustrated by its application to the TCP/IP protocol.
Key Words: Denial of service, Protocols, Admissible interference, Bisimulation, Equivalence-checking.
Category: C.2.2, C.2.4

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 become one of the primary tasks in computer science. In recent years, equivalence-checking has proved to be useful for the verification of security protocols [Abadi and Gordon 1998, Boreale et al. 1999, Cortier 2002, Lafrance and Mullins 2002a]. The main idea behind this approach of formal verification is to verify a security property by testing whether the process (specifying the protocol) is bisimilar to its intended behavior. The success of these methods is based on two characteristics: process algebras are applicable to the specification of such protocols, including cryptographic protocols; and bisimulation offers an expressive semantics for process calculi. Many other methods from a wide range of approaches have been proposed in the literature to analyze security protocols, but most are dedicated to the validation of confidentiality and authentication policies. So far, little attention has been paid to denial of service (DoS), even though the inability to clearly establish a formal characterization for DoS has made this type of attack a growing concern for protocol designers. This paper introduces a method based on equivalence-checking for the detection of denial of service vulnerabilities in security protocols.

In recent years, several Internet sites have been subjected to DoS attacks. One of the most famous DoS is the *SYN flooding* attack [Schuba et al. 1997] on the TCP/IP protocol. Since 1996, this resource exhaustion attack has been launched at several occasions by intruders who were able to initiate with little effort a large number of protocol runs. This was possible because it is easy to forge an identity, and so it is difficult for the victim to identify an intruder. Other DoS attacks have targeted several e-commerce sites, including Yahoo, Ebay and E*trade in February 2000, and Microsoft in January 2001.

Yu & Gligor [Yu and Gligor 1988] have proposed a formal specification and verification method for the prevention of DoS. 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 a liveness problem (*some users prevent some other users from making progress within the service for an arbitrary long time*) and a safety problem (*some users make some other users receive incorrect service*). Millen [Millen 1992] extended the Yu-Gligor resource allocator model by explicitly representing time. By doing so, Millen can support probabilistic policies, e.g. a maximum-waiting-time policy. Cuppens & Saurel [Cuppens and Saurel 1999] introduced a similar approach using an availability policy formalized in temporal logic and deontic logic. The approaches developed around Yu & Gligor's frameworks rely essentially on an access control policy called *user agreement*. They do not offer protection against attacks which occur before parties are mutually authenticated, as is the case for the SYN flooding attack and distributed DoS attacks. In such DoS attacks on protocols which establish authenticated communication channels, the identity of the intruder is generally unknown because authentication has not yet been completed. One way to prevent such attacks is the use of 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 he can force a system to spend resources running the later stages of the protocol. Recently, Meadows [Meadows 2001] has proposed a cost-based method to analyse these protocols, based on Gong & Syverson's fail-stop protocol model [Gong and Syverson 1998]. Meadows interprets this fail-stop model by requirements specification based on Lamport's *causally-precedes* relation, which states which events should causally precede others in a protocol:

although an intruder is capable to break some of the (weak) authentication steps of a protocol, it will cost him a dissuading effort. This approach prevents multiple exploitations of a single protocol flaw which could lead to DoS. The *NRL protocol analyzer* [Meadows 1996] was used within this framework.

The basic idea in our own method is to prove that no intruder can use the protocol to *interfere* with costly actions of the defender to cause resource exhaustion DoS. *Non-interference* properties [Goguen and Meseguer 1982] capture any causal dependencies between private actions and public behaviours which could be used to infer private information from public channels. However, many practical secrecy problems go beyond the scope of non-interference. As an example, cryptosystems permit classified or encrypted private information to flow safely onto unprotected channels despite the obvious causal dependency between the secret data m and encryption key k, and, on the other hand, the declassified data $\{m\}_k$ (m encrypted by k). Indeed, any variation of m or k is reflected in $\{m\}_k$. In this case, the basic concern is to ensure that programs leak sensitive information *only* through the cryptosystem or more generally, through the downgrading system. *Admissible interference* [Mullins 2000, Mullins and Yeddes 2001] is such a property. In a previous paper [Lafrance and Mullins 2002a], the authors have designed an equivalence-checking method based on admissible interference to analyze cryptographic protocols. This bisimulation-based method consists in proving that no intruder can interfere with the protocol unless the interference occurs through a predetermined action. Admissible interference is expressed by simply identifying admissible attacks corresponding to harmless enemy actions which may occur in the protocol.

The main contribution of this paper is an admissible-interference-based security property and a bisimulation-based algorithm for the validation of security protocols with respect to DoS robustness, called *impassivity*. Our method verifies whether the behaviour of a principal (e.g. server) differ when we introduce an intruder in the protocol. Roughly speaking, if the principal behaves differently when it is being attacked, then we conclude that the protocol is unsafe. Our robustness against DoS, called *impassivity*, is then formalized by asking that any costly behaviour (in terms of CPU or memory), which could lead to a resource exhaustion DoS for the defender (server), must be independent of any attack. Hence, robustness against DoS should be satisfied whenever the costly behaviour of the server being attacked are bisimilar to the costly behaviours of the server within a regular protocol run (not being attacked). This approach requires formal specifications of both the protocol's principals and the intruder's attacks. Therefore, we establish our theoretical framework by introducing a generic process algebra with value-passing called *Security Protocols Process Algebra* (*SPPA*), with extensions to monitor local function calls made by a principal (process) as visible actions and using marker actions to keep track of information exchanges, which would be lost otherwise during communication. Moreover, every action is assocaited to a cost describing the quantity of resources used to execute it.

Although the cost-based framework of our method is inspired by Meadows's, we do not consider the accumulative cost of the intruder and other principals throughout a protocol run. Instead, we simply observe the maximal cost actions of behaviours. Intruder capabilities are explicitly captured in this paper by our notion of enemy process. Enemy processes are regular SPPA processes which may individually pursue a specific attack. Meadows's procedure to evaluate whether a protocol is vulnerable to DoS is primarily based on a tolerance relation which determines the amount of resources the designer is willing to spend to provide a given level of security.

The paper is organised as follows. Process algebra SPPA, which handles the architecture of cryptographic protocols, is described in [Section 2]. In [Section 3], a short introduction to the notions of non-interference and admissible interference are given together with an unwinding theorem. Our information flow method for the detection of potential resource exhaustion DoS is presented in [Section 4], along with a sound and complete proof method [see Theorem 3] and an application to the TCP protocol. In [Section 5], we discuss related and future works.

1.1 The Transmission Control Protocol

The *Transmission Control Protocol* (TCP) provides a reliable connection-oriented data stream delivery service for applications. The TCP connection protocol will serve as an 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 address, the port number, the timer, the sequence number, the flow control status, etc. (A full description of the TCP connection establishment process in terms of a state machine is given by Schuba & al. [Schuba et al. 1997].) Before starting the transmission of data between a source A and a destination B, TCP uses a connection establishment protocol called the *three-way handshake*. The three-way handshake is achieved with the following steps:

$$\begin{array}{lccc} \text{Message 1:} & A & \overset{SYN_n}{\longrightarrow} & B \\ \text{Message 2:} & B & \overset{SYN_m, ACK_{n+1}}{\longrightarrow} & A \\ \text{Message 3:} & A & \overset{ACK_{m+1}}{\longrightarrow} & B\ . \end{array}$$

First, A initiates the connection by sending to B a SYN_n packet, 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+1} and SYN_m to A. Finally, A acknowledges B's packets by replying ACK_{m+1}. Whenever B receives a SYN packet, data structures are allocated.

For instance, consider a *SYN flooding* resource exhaustion attack on the TCP/IP protocol which is initiated as follows:

1. *newid*: the intruder generates a fake identifier (address) id;
2. $newSYN(id)$: the intruder generates a random number n and creates a SYN packet SYN_n containing the sequence number (in that case n) along with the source and destination's IP address (in that case, a fake address id and the server's address);
3. $output(SYN_n)$: the intruder sends SYN_n to the server.

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

1. $store(SYN_n)$: the server allocates data structures in which packet SYN_n is stored;
2. $makeACK(SYN_n)$: the server creates an acknowledgment packet ACK_{n+1} for SYN_n;
3. $newSYN(SYN_n)$: the server generates a random number m and creates a SYN packet SYN_m;
4. $output(SYN_m, ACK_{n+1})$: the server sends SYN_m and ACK_{n+1} to E.

The intruder then repeats this attack with different fake identifiers and different SYN packets SYN_n, but without completing the protocol run. This attack is possible because generating fake identifiers (addresses) and SYN packets requires few resources. Resource exhaustion DoS occurs because the server allocates expensive data structures upon receiving a SYN packet (specified above as the action $store(SYN_n)$). In order to analyze 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 $(cost(newid)+cost(newSYN)+cost(output))$ is much less than the cost of processing it $(cost(store) + cost(newACK) + cost(newSYN) + cost(output))$, then the protocol obviously has a flaw since an intruder could launch, with very few resources, multiple attacks that can waste a lot of the server's resources. If this exceeds the server's resource capacity, it must deny any new legitimate protocol run. Given an attack that is within the intruder capacity, we verify whether the server is robust against resource exhaustion DoS by testing if the protocol in which the attack is launched is bisimilar, in terms of the server's costly behaviours, to the protocol in which no attack is launched. In the case of the SYN flooding attack, the bisimulation is not satisfied since the server's costly behaviour $(\stackrel{store}{\longrightarrow} \stackrel{newACK}{\longrightarrow} \stackrel{newSYN}{\longrightarrow} \stackrel{output}{\longrightarrow})$ is a direct consequence of the intruder's attack $(\stackrel{newid}{\longrightarrow} \stackrel{newSYN}{\longrightarrow} \stackrel{output}{\longrightarrow})$. Hence, this server's behavior cannot be simulated by the protocol with no intruder. Note that a server's costly behaviour may have been caused by some other (honest) principal, in which case the property is satisfied.

2 A Process Algebra to Specify Security Protocols

Our first step for the validation of security protocols is to find a language able to express both the protocols and the security policies we want to enforce. Process algebra has been used for several years to specify protocols as a cluster of concurrent processes (representing principals participating in the protocol) able to communicate and exchange data. CSP was one of the first process algebras successfully used for this purpose [Schneider 1996]. In this section we introduce a generic process algebra à la CCS [Milner 1989] with value-passing called *Security Protocol Process Algebra* (SPPA) with some extensions to handle security protocols. SPPA allows the specification of *local function calls* and introduces *marker actions* used to tag value exchanges between processes. Up to these extensions tailored just to fit the ideas presented here, SPPA is very similar to SPA presented in [Durante et al. 1999]. Also, 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 analyze cryptographic protocols. In the following, we give a brief description of SPPA's syntax and operational semantics.

2.1 The Syntax of SPPA

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

$$\begin{array}{rcll} t ::= & n & (number) \mid \ id \ (identifier) & \mid \ x \ (variable) \\ \mid & (t, \ldots, t) & (n\text{-}tuple) \mid \{t\}_t \ (encryption) & \mid [t]_t \ (signature) \\ \mid & h(t) & (hashing) & \end{array}$$

For any term t, we denote $fv(t)$ 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 the sake of clarity, we will discriminate a subset $\mathcal{K} \subseteq \mathcal{M}$ of messages that may be used as encryption keys. Note that the definition of the set $\mathcal{K}$ usually depends on the cryptosystems used by the protocol. Moreover, in order to deal with public-key encryption, we use an idempotent operator $[-]^{-1} : \mathcal{K} \rightarrow \mathcal{K}$ such that a^{-1} denotes the private decryption key corresponding to the public encryption key a, or vice versa. For symmetrical encryption, we set $a^{-1} = a$. One assumes perfect encryption and hashing.

We consider a finite set $\mathcal{F}$ of *private functions* which range over messages and create new messages using the grammar rules above. $\mathrm{dom}(f)$ denotes the domain of messages of the private function f. Moreover, we assume disjoint sets $\mathcal{F}_{id}$ of private functions, for every identifier id, such that $\mathcal{F} = \bigcup\limits_{id \in \mathcal{I}} \mathcal{F}_{id}$. Intuitively, the principal assigned to the identifier id has only access to functions from $\mathcal{F}_{id}$, which usually includes the following:

- $extract_{id}^{n,i}((a_1,\ldots,a_n)) = a_i$ (*extraction function* for $i = 1, 2$ with domain $\{(a_1,\ldots,a_n) \mid a_1,\ldots,a_n \in \mathcal{M}\}$);
- $enc_{id}(k,a) = \{a\}_k$ (*encryption function* with domain $\mathcal{K} \times \mathcal{M}$);
- $dec_{id}(k^{-1}, \{a\}_k) = a$ (*decryption function* with domain $\{(k^{-1}, \{a\}_k) \mid k \in \mathcal{K}$ and $a \in \mathcal{M}\}$);
- $hash_{id}(a) = h(a)$ (*hash function* with domain $\mathcal{M}$);
- $sign_{id}(k,a) = [a]_k$ (*signature function* with domain $\mathcal{K} \times \mathcal{M}$);
- $checksign_{id}(k^{-1}, a, [a]_k)$ (*signature verification function* with domain $\{(k^{-1}, a, [a]_k) \mid k \in \mathcal{K}$ and $a \in \mathcal{M}\}$).

Note that the *checksign* function does not produce any new term since its primary task is to verify whether its input term is within its domain. Such a *verification function* is treated as a function whose only output term is the Boolean "$(k^{-1}, a, [a]_k) \in \mathrm{dom}(checksign_{id})$".

We consider a finite set C of *public channels*. Public channels are commonly used to specify message exchanges between principals. Every public channel c has a predetermined domain $\mathrm{dom}(c)$ of messages which can be sent and received over c. In this paper, we shall assume that $\mathrm{dom}(c) = \mathcal{M}$ for every c. The *prefixes* of SPPA are obtained as follows:

$$\begin{array}{lll} \mu ::= & \overline{c}(t) \;\; (\textit{output prefix}) & \mid c(x) \; (\textit{input prefix}) \\ & \mid x := f(t) \; (\textit{functional prefix}) & \end{array}$$

where t is any term such that $x \notin fv(t)$. For a verification function f, we often write $f(t)$ instead of $x := f(t)$ since f has no output. Moreover, we write $\texttt{fail} := f(a)$ whenever $a \notin \mathrm{dom}(f)$.

Let μ be a prefix and let t, t' be terms. The *agents* of SPPA are constructed from the following grammar:

$$\begin{array}{llll} S ::= & \mathbf{0} \;\; (\textit{empty agent}) & \mid \; \mu.S & (\textit{prefix agent}) \\ & \mid [t = t']\; S \; (\textit{match}) & \mid S + S & (\textit{sum}) \\ & \mid \;\; S|S \;\; (\textit{parallel composition}) & \mid \; S/\mathcal{O} & (\mathcal{O}\textit{-observation}) \\ & \mid \;\; S\backslash L \;\; (\textit{restriction}) & & \end{array}$$

where L is a set and $\mathcal{O}$ is a partial mapping (both to be clarified in [Section 2.2]). In this syntax, recursion is handled using agent names (e.g. by writing $S = \mu_1.\mu_2.S$). Given an agent S, we define its set of *free variables*, denoted by $fv(S)$, as the set of variables x appearing in S which are not in the scope of an input prefix $c(x)$ or a functional prefix $x := f(t)$; otherwise the variable x is said to be *bound*. Given a free variable $x \in fv(S)$ 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 $fv(S) = \emptyset$.

Intuitively, closed agents are used to specify the principals of security protocols. More specifically, an 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 X as a reference to the principal (S_X, id_X) where S_X is the *initial agent* of X i.e., the closed agent specifying the entire behaviour of the principal X within the protocol. Moreover, given a principal A from a protocol, we commonly use the identifier id_A for a message containing its address, while we simply use A to refer to the protocol's entity. For simplicity, we often write $A_1|A_2$ instead of $(S_1|S_2, id)$, $A_1 + A_2$ instead of $(S_1 + S_2, id)$, and $[a = a']A_1$ instead of $([a = a']S_1, id)$, where $A_1 = (S_1, id)$ and $A_2 = (S_2, id)$ (they must have the same identifier).

In order to specify a security protocol in SPPA, we use the classic approach [Focardi et al. 1997, Schneider 1996] of specifying the principals as concurrent agents. Given a principal A, SPPA *processes* are constructed as follows:

$$\begin{aligned} P ::= \quad & A \quad (\mathit{principal}) \qquad | \ A \parallel P \ (\mathit{protocol}) \ | \ P \backslash L \ (\mathit{restriction}) \\ & | \ P/\mathcal{O} \ (\mathcal{O}\text{-}\mathit{observation}) \end{aligned}$$

where $\parallel$ is an associative and commutative operator that forces communication over the set C of public channels used by the protocol (commonly, there is one channel for every step of a protocol run).

Example 1. The (one run) TCP's three-way handshake connection establishment protocol is specified as follows:

$$TCP \ ::= \ A_n \parallel B_m$$

where $C = \{c_1, c_2, c_3\}$ and principals A_n and B_m (with $id_{A_n} = id_A$ and $id_{B_m} = id_B$) are defined as follows:

$$\begin{aligned} A_n ::= \ & x_1 := store_{id_A}(SYN_n).\ \overline{c_1}(x_1).\ c_2(x_2).\ x_3 := extract^{2,1}_{id_A}(x_2). \\ & x_4 := extract^{2,2}_{id_A}(x_2).\ checkAck_{id_A}(SYN_n, x_4). \\ & x_5 := makeAck_{id_A}(x_3).\ \overline{c_3}(x_5).\ A' \\ B_m ::= \ & c_1(y_1).\ y_2 := makeAck_{id_B}(y_1).\ y_3 := store_{id_B}(SYN_m). \\ & \overline{c_2}((y_3, y_2)).\ c_3(y_4).\ checkAck_{id_B}(y_3, y_4).\ B' \end{aligned}$$

where A' and B' are principals specifying the remainder of the TCP protocol and the private functions are defined as follows:

– $store_{id}(a) = a$ $\quad$ (*storing function* over $\mathcal{M}$);

– $checkAck_{id}(SYN_n, ACK_{n+1})$ (*acknowledgment verification function* over the set of pairs of SYN packet and ACK packet with corresponding sequence number);

– $makeAck_{id}(SYN_n) = ACK_{n+1}$ (*acknowledgment function* over the set of SYN packets).

Intuitively, $store_{id}(a)$ really stands for the address where the message is being stored. Note that SYN_n and ACK_n packets may be viewed as tuples $(header, id_X, id_{X'}, n)$ from our message algebra where $id_X, id_{X'} \in \mathcal{I}$ are the source and the destination identifiers, and $n \in \mathcal{N}$ is a sequence number. Since TCP does not need an encryption-related function, we only need to consider, for every $id \in \mathcal{I}$, the set of private functions

$$\mathcal{F}_{id} = \{extract_{id}^{n,i},\ store_{id},\ checkAck_{id},\ makeAck_{id}\}.$$

A destination principal B (e.g. a 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 $B_{m_1,\ldots,m_N} ::= B_{m_1} \mid \ldots \mid B_{m_N}$ where principals B_{m_i} use different packets SYN_{m_i}. Since $n, m_1, \ldots, m_N$ stands for random numbers, we simply write A instead of A_n, and B^N instead of $B_{m_1,\ldots,m_N}$. Therefore, the TCP connection establishment process is given by process $TCP ::= A \parallel B^N$.

2.2 The Semantics of SPPA

Markers 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 since they are not considered as prefixes, and their specific semantics restricts their occurrence in order to tag the exchanges between principals. In value-passing process algebra, communication is commonly expressed by replacing the matching output action and input action by the silent action τ, which causes a considerable loss of information about the values exchanged 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* $\overline{\delta_{id_X}^c}(a)$ stands for "the principal X has sent message a over the channel c", and the occurrence of an *input marker* $\delta_{id_X}^c(a)$ stands for "the principal X has received message a over the channel c". Hence, similarly to private functions, every marker action belongs to the principal stated in its parameter.

Given a message $a \in \mathcal{M}$, the *actions* of SPPA are defined as follows:

$$\begin{array}{rcl} \alpha ::= & \overline{c}(a) & (\textit{output action}) \\ | & c(a) & (\textit{input action}) \\ | & a' := f(a) & (\textit{functional action} \text{ where } a' = f(a)) \\ | & \texttt{fail} := f(a) & (\textit{fail action} \text{ where } a \notin \mathrm{dom}(f)) \\ | & \delta(a) & (\textit{marker action}) \\ | & \tau & (\textit{silent action}). \end{array}$$

For instance, action $\{a\}_k := enc_{id_X}(k, a)$ stands for principal X encrypting message a with the key k and obtaining, in return, message $\{a\}_k$. We write Act to denote the set of all actions and we consider the set Act_X of actions observable only by the principal X, defined by:

$$\begin{aligned} Act_X = & \{a' := f(a) \in Act \mid f \in \mathcal{F}_{id_X} \text{ and } a \in \mathrm{dom}(f)\} \\ & \cup \{\texttt{fail} := f(a) \in Act \mid f \in \mathcal{F}_{id_X} \text{ and } a \notin \mathrm{dom}(f)\} \\ & \cup \{\overline{\delta^c_{id_X}}(a),\ \delta^c_{id_X}(a) \in Act \mid c \in C \text{ and } a \in \mathcal{M}\}. \end{aligned}$$

We typically use C to denote both the set of public channels and the set of output and input actions.

An *observation criterion* is a partial mapping $\mathcal{O} : Act^* \mapsto Act$ designed to express the equivalence

between process behaviours. Two sequences of actions γ_1 and γ_2 are said to carry out the same observation α whenever $\gamma_1, \gamma_2 \in \mathcal{O}^{-1}(\alpha)$. Given a subset $L \subseteq Act \setminus \{\tau\}$, we consider the observation criterion $\mathcal{O}_L$ defined as follows:

$$\mathcal{O}_L^{-1}(\alpha) = \begin{cases} (Act \setminus L)^* \ \alpha \ (Act \setminus L)^* & \text{if } \alpha \in L \\ (Act \setminus L)^* & \text{if } \alpha = \tau. \end{cases}$$

Only the behaviour from the set L is observable along this criterion. In particular, we have a natural observation criterion $\mathcal{O}_{Act_X \cup C}$, often denoted by $\mathcal{O}_X$, describing the actions observable by a principal X.

The operational semantics of a process can be viewed as an extension of the usual notion of non-deterministic automaton where we generally do not consider final states. The operational semantics of SPPA processes is defined in [Fig. 1] where $a \in \mathcal{M}$ is a message, $L \subseteq Act$ is a subset of actions, and P, P', Q, Q' are processes. The `Sum`, `Parallel`, `Protocol` and `Synchronisation` rules are assumed to be both associative and commutative.

The `Output` rule allows a principal A to output messages over public channels. Conversely, the `Input` rule needs to consider every possible message that an agent may receive over a public channel. The `Function` rule restricts the execution of local function calls to their owner, while the `Fail` rule deals with the case where a function is called on terms outside its domain. The `Match` rule allows the verification of equality between two messages. The `Sum` and `Parallel` rules allow for the specification of non-deterministic sum and parallel product of principals (with a matching identifier). The `Protocol` and `Synchronisation`

$$\texttt{Output} \qquad \frac{-}{\overline{c}(a).P \xrightarrow{\overline{c}(a)} P}$$

$$\texttt{Input} \qquad \frac{a \in \mathcal{M}}{c(x).P \xrightarrow{c(a)} P[a/x]}$$

$$\texttt{Function} \qquad \frac{a'{=}f(a) \quad \text{and} \quad f \in \mathcal{F}_{id_P}}{x{:=}f(a).P \xrightarrow{a'{:=}f(a)} P[a'/x]}$$

$$\texttt{Fail} \qquad \frac{a{\notin}\mathrm{dom}(f) \quad \text{and} \quad f \in \mathcal{F}_{id_P}}{x{:=}f(a).P \xrightarrow{\texttt{fail}{:=}f(a)} \mathbf{0}}$$

$$\texttt{Match} \qquad \frac{P \xrightarrow{\alpha} P'}{[a{=}a]P \xrightarrow{\alpha} P'}$$

$$\texttt{Sum} \qquad \frac{P \xrightarrow{\alpha} P'}{P{+}Q \xrightarrow{\alpha} P'}$$

$$\texttt{Parallel} \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\texttt{Protocol} \qquad \frac{P \xrightarrow{\alpha} P' \quad \text{and } \alpha{\notin}C}{P\|Q \xrightarrow{\alpha} P'\|Q}$$

$$\texttt{Synchronisation} \qquad \frac{P \xrightarrow{\overline{c}(a)} P' \quad \text{and} \quad Q \xrightarrow{c(a)} Q'}{P\|Q \xrightarrow{\overline{\delta^c_{id_P}}(a)} P'\|Q \xrightarrow{\delta^c_{id_Q}(a)} P'\|Q'}$$

$$\texttt{Restriction} \qquad \frac{P \xrightarrow{\alpha} P' \quad \text{and} \quad \alpha \notin L}{P\backslash L \xrightarrow{\alpha} P'\backslash L}$$

$$\mathcal{O}\texttt{-Observation} \qquad \frac{P \xrightarrow{\gamma} P' \quad \text{and} \quad \gamma \in \mathcal{O}^{-1}(\alpha)}{P/\mathcal{O} \xrightarrow{\alpha} P'/\mathcal{O}}.$$

Figure 1: *The Semantics of SPPA processes.*

rules allows the specification of protocols, where the operator $\|$ is similar to a parallel product between principals in which the communication between principals is achieved (and forced) through public channels. The `Restriction` rule interprets $P\backslash L$ (where L is a set of actions) as P with the actions in L forbidden. Finally, the $\mathcal{O}$-`Observation` rule interprets the observation of a process through an observation criterion $\mathcal{O}$, where the *computation* $P \xrightarrow{\gamma} P'$, for a sequence of actions $\gamma = \alpha_0\alpha_1\ldots\alpha_n \in Act^*$, stands for the finite string of transitions

$P \xrightarrow{\alpha_0} P_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} P'$. Thus, $P/\mathcal{O}_L$ (where L is a set of actions) means P with the actions outside L ignored.

A process P' is a *derivative* of P if there is a computation $P \xrightarrow{\gamma} P'$ for some $\gamma \in Act^*$. We shall frequently use the set

$$\mathcal{D}(P) = \{P' \mid \exists_{\gamma \in Act^*}\ P \xrightarrow{\gamma} P'\}$$

the set of P's derivatives.

For the following, we need Milner's notions of (strong) bisimulation, denoted by $\simeq$ [Milner 1989]. The concept of $\mathcal{O}$-bisimulation, called $\mathcal{O}$-congruence by Boudol [Boudol 1985], captures the notion of behavioural indistinguishability through an observation criterion $\mathcal{O}$. Given an observation criterion $\mathcal{O}$, we say that the process P is $\mathcal{O}$-*simulated* by the process Q whenever $P/\mathcal{O} \sqsubseteq Q/\mathcal{O}$, and we write $P \sqsubseteq_{\mathcal{O}} Q$. Moreover, we say that the process P is $\mathcal{O}$-*bisimilar* to the process Q whenever $P/\mathcal{O} \simeq Q/\mathcal{O}$, and we write $P \simeq_{\mathcal{O}} Q$. For instance, consider the weak criterion $\mathcal{O}_{Vis}$, where $Vis = Act \setminus \{\tau\}$ is the set of visible actions. We can easily see that $\mathcal{O}_{Vis}$-bisimulation corresponds to Milner's weak bisimulation [Milner 1989].

3 Admissible Interference

Given a process P and two disjoint subsets K and L of the set Vis of visible actions, K is said to cause *interference* on L (within the process P) whenever there are actions from K (in P) causing actions from L which might have not occurred otherwise. For instance, in [Fig. 2] we see that action α_1 causes interference on action α_2 in process Q, but not in process P.

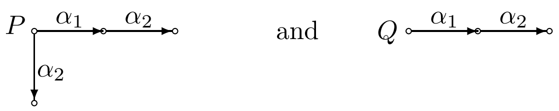

Figure 2: *SPPA processes P and Q.*

The following formulation of non-interference (with respect to L and K) requires that a process $\mathcal{O}_{K\cup L}$-simulates its $\mathcal{O}_L$-observation. Hence, roughly speaking, *bisimulation-based strong non-deterministic non-interference* (BSNNI) states that any observable behaviour from L remains a behaviour of the process in which actions from L and K are observable, in order to disallow any correlation between behaviours from K and from L. Formally, process P satisfies BSNNI if

$$P/\mathcal{O}_L \ \simeq_{\mathcal{O}_{K\cup L}} \ P \setminus K\ .$$

From [Fig. 2], assuming that $\alpha_1 \in K$ and $\alpha_2 \in L$, we see that process P satisfies BSNNI, but not process Q. When K holds for the set Hi of *high-level* observable actions and L holds for the set Lo of *low-level* observable actions, it is not difficult to see that this property coincides with bisimulation-based strong non-deterministic non-interference as proposed by Focardi & Gorrieri [Focardi and Gorrieri 1994/1995].

Given a set $\Gamma \subseteq Vis$ 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 downgrading actions. To capture this property, it was proposed [Mullins 2000] 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 [Rushby 1992], process $P' \setminus \Gamma$ must satisfy non-interference for every derivative $P' \in \mathcal{D}(P)$. Rephrasing this condition in the context of BSNNI as the non-interference property yields the definition of *bisimulation-based non-deterministic admissible interference* (BNAI) [Lafrance and Mullins 2002a]. Formally, process P satisfies BNAI if

$$\forall_{P' \in \mathcal{D}(P)} \quad (P' \setminus \Gamma)/\mathcal{O}_L \ \sqsubseteq_{\mathcal{O}_{K \cup L}} \ (P' \setminus \Gamma).$$

The next theorem is an algebraic characterisation of BNAI based on $\mathcal{O}_L$-bisimulation.

Theorem 1 (Unwinding Theorem for BNAI). *Process P satisfies BNAI if and only if*

$$\forall_{P' \in \mathcal{D}(P)} \quad P' \setminus \Gamma \ \simeq_{\mathcal{O}_L} \ P' \setminus (\Gamma \ \cup \ K).$$

A complete proof of this result was presented in [Lafrance and Mullins 2002a] in the context of a simpler process algebra. The general proof has to be done by double induction on the structures of both messages and processes. In the following section, we only give a sketch of this proof.

Proof. Given $Q ::= P' \setminus \Gamma$ for $P' \in \mathcal{D}(P)$, we see that

$$\begin{aligned} Q/\mathcal{O}_L \ \sqsubseteq_{\mathcal{O}_{L\cup K}} Q &\iff Q/\mathcal{O}_L \ \sqsubseteq \ Q/\mathcal{O}_{L\cup K} \\ &\iff Q/\mathcal{O}_L \ \sqsubseteq \ (Q/\mathcal{O}_{L\cup K}) \setminus K \\ &\iff Q/\mathcal{O}_L \ \sqsubseteq \ (Q \setminus K)/\mathcal{O}_L \\ &\iff Q \ \sqsubseteq_{\mathcal{O}_L} Q \setminus K. \end{aligned}$$

The result then follows from the fact that any $\mathcal{O}_L$-simulation of Q by $Q/\mathcal{O}_L$ is actually an $\mathcal{O}_L$-bisimulation. □

4 Finding 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 (mostly authentication protocols). In such an attack, at any step of the protocol (but mostly at the beginning), the intruder 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 a large waste of resources to the defender. If the defender can simultaneously process several requests (protocol runs), the intruder can then repeat his attack up to the point of causing a resource exhaustion; the defender then has to refuse any other request for a protocol run, even from honest principals. This type of DoS, which includes distributed DoS, is formalized as N copies of an enemy process simultaneously initiating protocol runs with a principal (defender) able to handle a maximum of N simultaneous requests. Since the whole attack is based on a single flaw in the protocol, it is 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 fatal distributed DoS attack.

However, we allow any interference coming from an intruder behaving properly. Such honest behaviours include the initiation of a real protocol run (with its own identifier and no fake message) and a proper response to an invitation to a protocol run. This assumption of allowing the behaviours of honest intruders is often omitted in the literature, but is crucial in order to view the intruder as a 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 property called *impassivity*, inspired by admissible interference and verified through $\mathcal{O}$-bisimulation.

4.1 Specification of Enemy Processes

Given a security protocol, we are particularly interested in studying its behaviour in a hostile environment. More precisely, we want to make sure that the protocol acts "correctly" in any given critical situation. In our process algebra, such hostile environments are expressed as enemy processes attempting to attack the protocol through its public channels. We consider a unique enemy identifier $id_E \in \mathcal{I}$ and a unique set $\mathcal{F}_{id_E}$ of enemy private functions. Therefore, every enemy process is related to the same enemy identifier. The set of admissible attacks, denoted by Γ, is a subset of the set Act_E.

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 the protocol P is written as the process $P_E \ ::= \ E \parallel B$, where B is the defender (server). Therefore, process $P_E \setminus \Gamma$ stands for the protocol run in which its honest behaviours are removed, leaving only potentially dangerous attacks.

For instance, the SYN flooding attack on the TCP connection protocol, specified as $A \parallel B^N$, is pursued by the enemy principal

$$E_{n_1,\ldots,n_N} ::= \ (\ \overline{c_1}(SYN_{n_1}).\mathbf{0} \mid \ldots \mid \overline{c_1}(SYN_{n_N}).\mathbf{0} \ )$$

(we write E^N for short) where each SYN_{n_i} packet contains a fake identifier instead of id_E. Therefore $TCP_E \ ::= \ E^N \parallel B^N$. Moreover, the set Γ of admissible attacks corresponds to the set of markers actions $\overline{\delta^{c_1}_{id_E}}(SYN_n)$ in which SYN_n contains id_E as its source's identifier. The set Γ also contains every input marker $\delta^{c_1}_{id_E}(SYN_n)$ in which SYN_n is intended for E.

4.2 Cost Function

The following approach for assigning cost to actions is inspired by Meadows' cost-based framework [Meadows 2001]. In order to compare resource spending, we consider two ordered *sets of costs* $\langle \mathcal{C}_{cpu}, < \rangle$ (for CPU resources) and $\langle \mathcal{C}_M, < \rangle$ (for memory resources). Moreover, we consider two *cost functions* $\rho_{cpu} : Act \mapsto \mathcal{C}_{cpu}$ and $\rho_M : Act \mapsto \mathcal{C}_M$, where $\rho_{cpu}(\alpha)$ stands for the amount of CPU resources (CPU cost) required to execute action α, and $\rho_M(\alpha)$ stands for the quantity of memory resources, namely the memory cost to execute action α. (Note that ρ_M's definition could be extended to deal with sequences of actions in which memory resource are released.)

Given a principal B (defender) within a protocol, we consider its *CPU capacity* $CPU_B \in \mathcal{C}_{cpu}$ which stands for the defender's CPU resource capacity: *running simultaneously N actions of CPU cost greater than CPU_B may cause B a CPU resource exhaustion DoS.* Similarly, we consider its *memory capacity* $M_B \in \mathcal{C}_M$, which stands for the defender's memory resource capacity: *running simultaneously N actions with a memory cost higher than M_B may cause B a memory resource exhaustion DoS.* Moreover, we consider the intruder's CPU capacity $CPU_E \in \mathcal{C}_{cpu}$ and memory capacity $M_E \in \mathcal{C}_M$, which respectively stand for the intruder's CPU resource capacity and memory resource capacity: *the enemy process E may only execute actions of CPU cost lesser or equal to CPU_E* and *the enemy process E may only execute actions with a maximum memory* *usage of* M_E*.* Therefore an intruder which may only launch low-cost attacks is specified as an enemy process E in which every transition $E' \xrightarrow{\alpha} E''$, with $E' \in \mathcal{D}(E)$, is such that $\rho_{cpu}(\alpha) \leq CPU_E$ and $\rho_M(\alpha) \leq M_E$. In that case, we say that the enemy process E *respects its capacities*. Obviously, the values of the capacities of each principal depend on many factors.

For the TCP protocol, we can assume that actions $checkAck_{id}(SYN, ACK)$ (along with their corresponding fail actions $\texttt{fail} := checkAck_{id}(a, a'))$ have the highest CPU cost, although we do not assume that their CPU cost is higher than B's capacity CPU_B, or E's capacity CPU_E. However, we assume that actions $store_{id}(a)$ have the largest memory cost and that their cost exceeds both B's capacity M_B, and E's capacity M_E. Therefore, memory resource exhaustion may occur whenever B stores some data.

4.3 Verifying Robustness Against DoS Through Equivalence-Checking

The following information flow property, called *impassivity*, used to specify robustness against DoS vulnerabilities, is inspired by information flow property BNAI. More precisely, impassivity verifies robustness against both CPU and memory resource exhaustion DoS. Given a server B, it states that no enemy process respecting its capacity may cause inadmissible interference on actions $\alpha \in Act^{>CPU_B} \cup Act^{>M_B}$, where $Act^{>CPU_B} = \{\alpha \in Act_B \mid \rho(\alpha) > CPU_B\}$ is the set of B's CPU-costly actions, and $Act^{>M_B} = \{\alpha \in Act_B \mid \rho(\alpha) > M_B\}$ is the set of B's memory-costly actions. Put $Act^{costly} = Act^{>CPU_B} \cup Act^{>M_B}$.

Definition 2 (Impassivity). Protocol P is *impassive* if, for every enemy process E respecting its capacity, the process P_E satisfies BNAI, with $K = Act_E$ and $L = Act^{costly}$.

The following theorem provides a sound and complete proof for impassivity. Its proof follows from [Theorem 1], where $P_E \setminus (\Gamma \cup Act_E) \simeq P_E \setminus Act_E$ since $\Gamma \subseteq Act_E$.

Theorem 3 (Unwinding Theorem for Impassivity). *Protocol P is impassive if, for every enemy process E respecting its capacity,*

$$\forall_{Q \in \mathcal{D}(P_E)} \quad Q \setminus \Gamma \ \simeq_{\mathcal{O}_{costly}} \ Q \setminus Act_E$$

where $\mathcal{O}_{costly} = \mathcal{O}_{Act^{costly}}$.

Therefore, given an enemy process, impassivity 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. Thus, Impassivity is satisfied by security protocols which use 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 he can force a system to expend resources running the later stages of the protocol.

One must also note that the definition of impassivity suffers from a universal quantification over enemy processes. This problem can be circumvented by defining a generic enemy process (respecting the enemy capacity) and verifying our property only with this process. This type of process, capable of $\mathcal{O}_{Vis}$-simulating any other enemy process, has been presented in [Lafrance and Mullins 2002b]. Although this generic process is infinite, it still is an attractive practical alternative for approximating the universal quantifier "*for every enemy process* E". ¿From [Theorem 3] and the enemy process E_N defined in [Section 4.1] (and which respects its capacity since there are no storing action), we can conclude that the TCP connection protocol does not satisfy impassivity. More precisely, we can see that

$$(E^N \parallel B^N) \setminus \Gamma \;\not\approx_{\mathcal{O}_{costly}}\; (E^N \parallel B^N) \setminus Act_E$$

where $\Gamma \;=\; \{\overline{\delta^c_{id_E}}(SYN_n),\ \delta^c_{id_E}(SYN_n) \mid SYN_n \text{ contains } id_E\}$.

4.4 Example of a Protocol not Satisfying Impassivity

The following is an example of an authentication protocol which does not satisfy impassivity.

Example 2. Consider the following protocol:

$$\begin{array}{llcl} \text{Message 1:} & A & \xrightarrow{id_A, n, [id_A, n]_{k_A}} & B \\ \text{Message 2:} & B & \xrightarrow{n} & A\ . \end{array}$$

In this simple protocol, principal A starts an authentication procedure with B by sending its identifier id_A and a fresh nonce n, both signed and unsigned, where k_A is A's private key. Upon receiving this message, B authenticates A's signature and replies with the nonce as an acknowledgment. We consider principals A and B, specified as follows:

$$A ::= \ x_1 := sign_{id_A}(k_A, (id_A, n)).\ \overline{c_1}(((id_A, n), x_1)).c_2(x_2).[x_2 = n]\ \mathbf{0}$$

$$\begin{aligned} B ::= \ & c_1(y_1).y_2 := extract^{2,1}_{id_B}(y_1).\ y_3 := extract^{2,2}_{id_B}(y_1). \\ & checksign_{id_B}(k_A^{-1}, y_2, y_3).\ y_4 := extract^{2,1}_{id_B}(y_2).\ \overline{c_2}(y_4).\mathbf{0}\ . \end{aligned}$$

The protocol is then specified as the process $P ::= A \parallel B$. For this protocol, we consider the set of functions $\mathcal{F}_{id} = \{extract_{id},\ sign_{id},\ checksign_{id}\}$ and we assume that no action exceeds the CPU capacity of server (B) nor the intruder's memory capacity. However, we assume that the signature verification actions exceeds the server's CPU capacity i.e.

$$\rho_{CPU}(checksign_{id_B}) \ > \ CPU_B$$

and that both the signature verification actions and signing actions exceed the intruder's CPU capacity, that is:

$$\rho_{CPU}(checksign_{id_E}), \ \rho_{CPU}(sign_{id_E}) \ > \ CPU_E \ .$$

Moreover, we consider the enemy principal (which respects its capacity) $E ::= \overline{c_1}(a).\mathbf{0}$ where $a = (a_1, a_2)$ is any pair of messages. This enemy process attacks the protocol by sending a fake message which causes B to execute a costly signature verification action (which fails). ¿From process $P_E ::= E \parallel B$, we see that protocol P does not satisfy impassivity since the marker action $\overline{\delta^{c_1}_{id_E}}(a)$ causes interference on the action $checksign_{id_B}$ which exceeds B's CPU capacity.

5 Related Work and Future Work

This paper presents a method based on admissible interference for the detection of DoS vulnerabilities in security protocols. It uses the SPPA process algebra, 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 SPPA and a cost-based framework, we introduce an information flow property called *impassivity* which detects any case when 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. Moreover, our cost-based framework allows an attribution of cost which depends on the capabilities of the various principals. For instance, if we suspect an attack by a strong intruder, we may impose that $\rho_{CPU}(\{a\}_k := enc_{id_E}(k,a)) < \rho_{CPU}(\{a\}_k := enc_{id_B}(k,a))$ i.e. impose that encryption require more CPU resource for the server B than for the intruder E.

The specification of security protocols and their validation against DoS vulnerabilities often requires to view function generating symbolic values such as random numbers, fresh nonces, fresh keys, fake addresses and fake messages. Following this approach, the authors [Lafrance and Mullins 2003] have introduced a symbolic extension of SPPA able to handle symbolic values. 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 formulas 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 formulas are updated by adding restrictions over the symbolic values, as required for the process to evolve. The authors also proposed a bisimulation equivalence between constrained processes which amounts to a generalisation of Milner's bisimulation between value-passing processes. The authors also provides a sound and complete symbolic bisimulation method to construct the bisimulation between constrained processes.

The most fatal distributed denial of service (DDoS) attacks have also caused their share of mayhem. Some tools devoted to DDoS presented by Criscuolo [Criscuolo 2000], Dietrich, Long & Dittrich [Dietrich et al. 2000] and Paxson [Paxson 2001] were developed to analyze such attacks based on specific malicious applications like *Trin00*, *TFN2K* and *Stacheldraht*. We also plan to improve our cost-based framework in order to clearly grasp DDoS. Since most resource exhaustion DoS attacks occur relatively early and usually involve a victim allocating costly data structures, we plan to extend 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 ρ_M able to compute the memory cost of any sequence of actions, including sequences of actions in which memory resource are released.

A tool to check whether a process satisfies admissible interference or not has been designed and implemented at the *École Polytechnique de Montréal*. We are currently extending this tool into a security protocol compiler. Protocols will be specified using a notation *à la Alice and Bob*, compiled into SPPA processes and analyzed along the lines described in this paper.

References

[Abadi and Gordon 1998] Abadi, M., Gordon, A. D.: "A bisimulation method for cryptographic protocols"; Nordic Journal of Computing, 5, 4 (1998), 267-303.

[Boreale et al. 1999] Boreale, M., De Nicola, R., Pugliese, R.: "Proof techniques for cryptographic processes"; Proc. Logic in Computer Science (1999), 157-166.

[Boudol 1985] Boudol, G.: "Notes on algebraic calculi of processes": Proc. Logic and Models of Concurrent Systems, NATO ASI Series F-13, Springer (1985), 261-303.

[Cortier 2002] Cortier, V.: "Observational equivalence and trace equivalence in an extension of spi-calculus. application to cryptographic protocols analysis"; Technical Report LSV-02-3, Lab. Specification and Verification, ENS de Cachan, Cachan, France (2002).

[Criscuolo 2000] Criscuolo, P. J.: "Distributed denial of service trin00, tribe flood network, tribe flood network 2000, and stacheldraht"; Technical Report CIAC-2319, Lawrence Livermore National Laboratory (Feb 2000).

[Cuppens and Saurel 1999] Cuppens, F., and Saurel, C.: "Towards a formalization of availability and denial of ssrvice"; Proc. Information Systems Technology Panel Symposium on Protecting Nato Information Systems in the 21st century, Washington (1999).

[Dietrich et al. 2000] Dietrich, S., Long, N., Dittrich, D.: "Analyzing distributed denial of service tools: The shaft case"; Proc. USENIX LISA (2000).

[Durante et al. 1999] Durante, A., Focardi, R., Gorrieri, R.: "CVS: A compiler for the analysis of cryptographic protocols"; Proc. of 12th IEEE Computer Security Foundations Workshop, IEEE Computer Society (June 1999).

[Focardi et al. 1997] Focardi, R., Ghelli, A., Gorrieri, R.: "Using non interference for the analysis of security protocols": Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols, Orman, H., Meadows, C. (editors), Rutgers University (Sep 1997).

[Focardi and Gorrieri 1994/1995] Focardi, R., Gorrieri, R.: "A classification of security properties for process algebras"; Journal of Computer Security, 3, 1 (1994/1995), 5-33.

[Goguen and Meseguer 1982] Goguen, J.A., Meseguer, J.: "Security policies and security models"; Proc. 1982 IEEE Symposium on Research in Security and Privacy (1982), 11-20.

[Gong and Syverson 1998] Gong, L., Syverson, P.: "Fail-stop protocols: An approach to designing secure protocols"; Proc. Dependable Computing for Critical Applications, 5, IEEE Computer Society (1998), 79-100.

[Lafrance and Mullins 2002a] Lafrance, S., Mullins, J.: "Bisimulation-based nondeterministic admissible interference and its application to the analysis of cryptographic protocols"; Proc. Electronic Notes in Theoretical Computer Science, 61, Harland, J. (editor), Elsevier Science Publishers (2002).

[Lafrance and Mullins 2002b] Lafrance, S., Mullins, J.: "A generic enemy process for the analysis of cryptoprotocols"; Proc. FSCBS'2002 (2002). Available at `www.crac.polymtl.ca/mullins`.

[Lafrance and Mullins 2003] Lafrance, S., Mullins, J.: "A symbolic approach to the analysis of security protocols"; Proc. of Foundations of Computer Security *affiliated with LICS'03*, Ottawa (2003). Available at `http://theory.stanford.edu/ iliano/fcs03/www/`.

[Meadows 1996] Meadows, C.: "The NRL protocol analyzer: An overview"; Journal of Logic Programming, 26, 2 (1996), 113-131.

[Meadows 2001] Meadows, C.: "A cost-based framework for analysis of denial of service networks"; Journal of Computer Security, 9, 1/2 (2001), 143-164.

[Millen 1992] Millen, J.: "A resource allocation model for denial of service"; Proc. of the 1992 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (1992), 137- 147.

[Milner 1989] Milner, R.: "Communication and concurrency"; Prentice-Hall (1989).

[Mullins 2000] Mullins, J.: "Nondeterministic admissible interference"; Journal of Universal Computer Science, 6, 11 (2000), 1054-1070.

[Mullins and Yeddes 2001] Mullins, J., Yeddes, M.: "Two proof methods for bisimulation-based non-deterministic admissible interference"; Submitted for publication (2001). Available at `www.crac.polymtl.ca/mullins`.

[Paxson 2001] Paxson, V.: "An analysis of using reflectors in distributed denial-of-service attacks"; (2001).

[Rushby 1992] Rushby, J.: "Noninterference, transitivity and channel-control security policies"; Technical Report CSL-92-02, SRI International, Menlo Park CA, USA (Dec 1992).

[Schneider 1996] Schneider, S.: "Security properties and CSP"; Proc. IEEE Symposium on Security and Privacy (1996), 174-187.

[Schuba et al. 1997] Schuba, C. L., Krsul, I. V., Kuhn, M. G., Spafford, E. H., Sundaram, A., Zamboni, D.: "Analysis of a denial of service attack on TCP"; Proc. of the 1997 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (May 1997), 208-223.

[Yu and Gligor 1988] Yu, C., Gligor, V. D.: "A formal specification and verification method for the prevention of denial of service"; Proc. 1988 IEEE Symposium on Security and Privacy, 117, IEEE Computer Society Press (Apr 1988), 187-202.