

# SEMANTICS OF THE PASSWORD-CAPABILITY SYSTEM

Dan Mossop and Ronald Pose

*C.S.S.E., Monash University  
Clayton, Victoria, Australia 3800.  
{dgm,rdp}@csse.monash.edu.au*

## ABSTRACT

The increasing problems of hacking and computer viruses have demonstrated the need for more secure computer systems. Conventional operating systems such as Unix, Linux, and Windows have not proved very satisfactory in dealing with such security problems. The capability paradigm seems to offer scope for more flexible computer system security but suffered from various implementation disadvantages. Password-capabilities allow for efficient implementation and offer scope for many security benefits. The Password-Capability System provides a simple, compact probabilistic security model. The operational semantics of the system are presented here.

## KEYWORDS

operating system, password-capabilities, operational semantics

## 1. INTRODUCTION

This paper characterizes the Password-Capability System (Anderson, 1986; Anderson, Pose and Wallace, 1986; Anderson and Wallace, 1988; Wallace and Pose, 1990), a capability-based operating system kernel that provides a novel paradigm for providing flexible and secure access to computational resources. The increasing problems of hacking and computer viruses have demonstrated the need for more secure computer systems. Conventional operating systems such as Unix, Linux, and Windows have not proved very satisfactory in dealing with such security problems. The capability paradigm (Dennis and Van Horn, 1966) seems to offer scope for more flexible computer system security but suffered from various implementation disadvantages. Password-Capabilities allow for efficient implementation and offer scope for many security benefits. The Password-Capability System provides a simple, compact probabilistic security model. The operational semantics of the system are presented here. The paper sets the scene future security analyses. Conventional operating systems involve the notion of users, who are authenticated via password or other mechanisms, and who are allocated access rights to various system resources such as files. Typically, once logged-in, all of a user's resources are accessible, and potentially prey to viruses and other undesirable phenomena. In practice this is achieved through access control lists that, for each resource, list which users may gain access. One may consider this like a guard at the door checking who is allowed to enter. The capability paradigm on the other hand has, for each user, a list of which resources are accessible. An analogy is that capabilities are like keys to the locked doors. There is no guard. Anyone possessing the appropriate key can gain access. This allows for the possibility of anonymity.

The implementation of capability-based operating systems has traditionally taken the form of hardware tagging of the capabilities, as for SWARD (Myers and Buckingham, 1980), or else the segregating of the capabilities in especially secure areas, as for Hydra (Cohen and Jefferson, 1975). This is required because one must guard against the forgery of capabilities, since the security of the system is dependent on them. These two approaches involve significant hardware or software overheads and disadvantages.

The Password-Capability System overcomes this using the equivalent of combination locks rather than key operated locks, rather like a bank vault. The security is based on the infeasibility of guessing a valid password-capability. The advantage is the flexibility available in managing password-capabilities without the restrictions and disadvantages of tagged or segregated capability implementations. It may be thought that

because the security is probabilistic that such a system is less secure than a conventional approach. Actually the contrary is true. It is more secure.

## 2. THE PASSWORD-CAPABILITY SYSTEM

The Password-Capability System is an operating system with a global virtual address space, access to which is controlled by a mechanism utilizing password-capabilities. All entities such as data, files, processes, programs, in all such systems throughout the world, are considered to be *objects*. The virtual memory is divided into *volumes*, each of which is typically a storage device. Each object resides on a volume, and is uniquely identified in the system by an object name  $(v, s)$ , where  $v$  is a unique volume number and  $s$  a serial number unique among the objects on a given volume. The lengths of the binary representation of  $v$  and  $s$  will be denoted by  $VLEN$  and  $SLEN$  respectively.

Operations on objects are only permitted when a suitable *password-capability* is presented to the system. A capability comprises an object name and a randomly chosen password. The password may be conveniently considered as being in two parts,  $p1$ , and  $p2$ , each of length  $PLEN$  bits. Thus a capability may be represented as  $(v, s, p1, p2)$ . Each capability allows access to some aspects of an object. Aspects of objects that capabilities may give access to include the abilities to read or write a subset of the data in the object, the abilities to start, stop, and send messages to objects that are processes, the ability to derive new capabilities for the same object with a subset of the access rights.

Each object when created has a single *master* capability, from which derivative capabilities can be created. The derivatives can never have more authority than their parents, and are destroyed if the parent is. Thus each object has associated with it, a singly-rooted tree of capabilities.

Capabilities are simply data values, comprising the object name  $(v, s)$  and a password  $(p1, p2)$ , with length  $VLEN+SLEN+PLEN+PLEN$ . The security of an object depends on the infeasibility of guessing  $(p1, p2)$ . This is ensured by choosing the password randomly and making  $PLEN$  large enough that the probability of guessing one is significantly less than that of guessing someone's password in a conventional system.

Apart from being a store of data and potentially a process, all objects also act as stores of money. This enables an integrated economic system to be created in which data and services can be traded, and resources managed in a familiar capitalist, market model. Thus all objects are in effect bank accounts. Objects that are processes also contain some cash for spending on immediate needs such as CPU time, in the same way as people have ready cash for immediate needs.

Processes have a *cashword* in which their cash is stored. They can send this money to other processes, or store it in other objects. The master capability indicates the object's total money in its *moneyword*. Derived capabilities' moneywords indicate withdrawal limits on the object's money. Thus the amount of money that can be withdrawn using a capability is the minimum of that of its moneyword, and that of all moneywords of its ancestors leading back to the master capability.

Processes can communicate with one another by passing short messages, or for larger messages, they can use an intermediate data object to which both have access. Message passing can be used to share capabilities to such an object. A process can choose to enter a *waiting* state, in which it will remain until a message is received. Processes can also be suspended and resumed. They will be terminated if they run out of cash to pay for operations, but can be revived by another process willing to provide cash to it.

The Password-Capability System has a mechanism for addressing the confinement problem (Lampson, 1973; Lipner, 1975). Each process has a *lockword* which it cannot read or change directly. To confine a process, an arbitrary value (lock) can be XORed with this lockword. Whenever the process presents a capability which could be used to convey information (an *alter* capability), the system XORs the capability password with the lockword before checking it. As the process does not know the lockword, it will not be able to supply a usable alter capability to a third party. The confining process can, however, authorize certain capabilities by XORing them with the lock before passing it to the confined process.

The system also provides a mechanism to allow for the implementation of secure abstract data type managers.

### 3. VIRTUAL MEMORY STATE

We model the Password-Capability System as a state machine. This section defines the static state of the system, effectively a snapshot of the contents of the virtual memory. The next section then defines the operations on the virtual memory. It is these operations which move the system from one state to the next.

We present the static state of the system in the following way. First we give a definition of the basic components, the virtual memory state, denoted  $S$ , the volumes and the objects. Then we present in more detail the components of an object: its data store, the set of capabilities defined on it, and the status information, used for objects acting as processes.

#### 3.1 Volumes

Each volume is uniquely identified by a volume number  $v \in \{0, 1\}^{VLEN}$ . The state of the volume with volume number  $v$  is denoted  $V_v$ . The state of the virtual memory is fully determined by the state of the volumes comprising it, so:

$$S = \{V_v : v \in \{0, 1\}^{VLEN}\}.$$

Each  $V_v$  is in turn determined by the states of the extant objects residing on volume  $v$ . Every object has a serial number  $s \in \{0, 1\}^{SLEN}$  unique among the objects on its volume. The state of an object is denoted by  $O_{v,s}$ . If no object exists with a given object name  $(v, s)$ , we write  $O_{v,s} = \text{null}$ . We can thus write:

$$V_v = \{O_{v,s} : s \in \{0, 1\}^{SLEN}, O_{v,s} \neq \text{null}\}.$$

We represent the state of an object,  $O_{v,s}$ , by a triple containing the state of the object's data space, its capability tree and its status information, which we denote as  $\text{data}_{v,s}$ ,  $\text{caps}_{v,s}$  and  $\text{status}_{v,s}$ , respectively. So,

$$O_{v,s} = (\text{data}_{v,s}, \text{caps}_{v,s}, \text{status}_{v,s}).$$

Here  $\text{data}_{v,s} \in \{0, 1\}^*$ . The state of the capability tree and the object status are described in detail below.

#### 3.2 Capability trees

The state of the tree of capabilities defined on some object,  $(v, s)$ , is denoted  $\text{caps}_{v,s}$ . Each capability for the object has a password  $(p1, p2)$ , with  $p1, p2 \in \{0, 1\}^{PLEN}$ , and is uniquely identified among the elements of the tree by  $p1$ . The state of the capability  $(v, s, p1, p2)$  is denoted by  $C_{v,s,p1}$ , where:

$$C_{v,s,p1} = (p2_{v,s,p1}, \text{money}_{v,s,p1}, \text{parent}_{v,s,p1}, \text{children}_{v,s,p1}, \\ \text{wstart}_{v,s,p1}, \text{wend}_{v,s,p1}, \text{rights}_{v,s,p1}).$$

The second half of the capability's password is denoted by  $p2_{v,s,p1}$ . The moneyword associated with this capability is given by  $\text{money}_{v,s,p1} \in \mathbb{R}$ . The first half password for the parent capability and children (derived) capabilities is given by  $\text{parent}_{v,s,p1} \in \{0, 1\}^{PLEN}$  and  $\text{children}_{v,s,p1} \in \{0, 1\}^{PLEN}$ . The window onto the object permitted by the capability is defined by its start and endpoints,  $\text{wstart}_{v,s,p1}$ ,  $\text{wend}_{v,s,p1} \in \mathbb{R}$ , which are offsets from the start of the object. The set of rights,  $\text{rights}_{v,s,p1}$ , which can be exercised by the capability is a subset of the set of all rights. Each of these rights permits a single operation to be carried out on the virtual memory. We defer the description of each right until the section describing the virtual memory operations, when it will be treated along with the operation it permits. If the capability  $(v, s, p1, p2)$  is the master capability for the object  $(v, s)$ , then  $\text{parent}_{v,s,p1} = \text{null}$ .

### 3.3 Status information

The status information for an object  $(v, s)$ , denoted  $status_{v,s}$ , is used to manage the distinction between data objects and processes. Accordingly,  $status_{v,s}$  has two forms. If  $(v, s)$  is a data object, then:  $status_{v,s} = (process_{v,s})$ , with  $process_{v,s} = false$ . If, on the other hand,  $(v, s)$  is a process:

$$status_{v,s} = (process_{v,s}, lock_{v,s}, cash_{v,s}, suspended_{v,s}, waiting_{v,s}, terminated_{v,s}, mailbox_{v,s})$$

The lockword and cashword of the process are given by  $lock_{v,s} \in \{0, 1\}^{2(PLEN)}$ ,  $cash_{v,s} \in \mathbb{N}$ . The boolean values,  $suspended_{v,s}$ ,  $waiting_{v,s}$  and  $terminated_{v,s}$ , are respectively true if the object has been suspended, is waiting on a message or has been terminated, and false otherwise.. The process's mailbox,  $mailbox_{v,s}$ , for received messages, is a list of messages,  $(m_1, m_2, \dots, m_i)$ , with  $m_i$  being the most recent.

## 4. VIRTUAL MEMORY OPERATIONS

This section describes operations that can modify the virtual memory. Our focus is on those operations invoked by processes. We identify all primitive operations available to the process. A process must form all its actions from some combination of these primitives.

Operation names are given in lower-case. Where the operation requires a capability to possess a certain access right, the access right is given by the upper-case equivalent of the operation name. When an operation incurs a cost, the amount of the cost is denoted by the word *cost* with the operation name as a subscript. Thus, the operation *read* requires the presented capability to have the access right *READ*, and costs  $cost_{read}$  to invoke. Finally, operations are presented in the format `proc.op(args)`, with the meaning that the operation *op* is invoked by process *proc* and is passed the arguments *args*.

### 4.1 Protected operations

A number of protected operations, i.e. operations not directly available to processes, are called by those operations which are. They are described here. The full semantics of the operations are listed in Table 1.

Table 1. Protected Operations

| Operation                             | Semantics  |
|---------------------------------------|--|
| <code>pays(proc, cost)</code>         | <code>(v,s) = proc; if (cash<sub>v,s</sub> &lt; cost) return false;<br/>if (cash<sub>v,s</sub> == 0) terminated<sub>v,s</sub> = true;<br/>cash<sub>v,s</sub> -= cost; return true;</code>  |
| <code>has(cap, right)</code>          | <code>(v,s,p1,p2) = cap;<br/>if (right rights<sub>v,s,p1</sub>) return true;<br/>else return false;</code>   |
| <code>can_see(cap, start, end)</code> | <code>(v,s,p1,p2) = cap;<br/>if (wstart<sub>v,s,p1</sub> start end wend<sub>v,s,p1</sub>)<br/>{ return true } else return false;</code>  |
| <code>is_process(cap)</code>          | <code>(v,s,p1,p2) = cap; return process<sub>v,s</sub>;</code>  |
| <code>check_pass(cap, proc)</code>    | <code>(v,s,p1,p2) = cap; p1' = p1; p2' = p2;<br/>if (p1[0] == 1)<br/>{ p1' = p1' ⊕ lock<sub>v,s</sub>[0..PLEN-1];<br/>p2' = p2' ⊕ lock<sub>v,s</sub>[PLEN..2(PLEN)-1]; }<br/>if (O<sub>v,s,p1'</sub> == null) return null;<br/>else return P2<sub>v,s,p1</sub> ⊕ p2';</code> |

`pays` – If the process has enough cash, the amount specified by *cost* is deducted from the cashword

has – Checks if the capability has the access right.  
 can\_see – Checks whether the capability can access the requested object window.  
 allocate – Allocate free memory.  
 rand – Return a random element of the set X, selected uniformly.  
 is\_process – Check if object is a process.  
 check\_pass – Checks validity of capability password

## 4.2 Object operations

make\_obj – Creates a new object.

Table 2. Object operations

| Operation                                 | Semantics   |
|---|---|
| proc.make_obj(vol, size, process, rights) | <pre> if (pays(proc, cost<sub>make-obj</sub>)) { s = rand({x[] {0,1}<sup>SLEN</sup> : O<sub>vol,x</sub> = null});   p1 = rand({0,1}<sup>PLEN</sup>); p2 = rand({0,1}<sup>PLEN</sup>);   C<sub>v,s,p</sub> = (p2,0,null,∅,0,size,rights);   if (process == true)   { status<sub>v,s</sub> = (true,0,0,true,false,false,()) }   else { status<sub>v,s</sub> = (false) }   O<sub>v,s</sub> = (status<sub>v,s</sub>, {C<sub>v,s,p</sub>}, allocate(size));   return (vol,s,p1,p2) } else return false; </pre> |

## 4.3 Data operations

The data field of an object can be used as a data store by processes. This is done through the following operations. Table 3 gives the semantics for these operations.

write – Write data to the object.  
 read – Read the data in the object.

Table 3. Content operations

| Operation                         | Semantics  |
|-----------------------------------|--|
| proc.read(cap, start, end)        | <pre> if (pays(proc, cost<sub>read</sub>) AND check_pass(cap, proc)==0 AND has(cap, READ) AND can_see(cap, start, end)) { (v,s,p1,p2) = cap; return data<sub>v,s,p1,p2</sub> } else return false; </pre>                               |
| proc.write(cap, start, end, data) | <pre> if (pays(proc, cost<sub>write</sub>) AND check_pass(cap, proc)==0 AND has(cap, WRITE) AND can_see(cap, start, end)) { (v,s,p1,p2) = cap; data<sub>v,s,p1,p2</sub>[start..end] = data;   return true; } else return false; </pre> |

## 4.4 Capability tree operations

The capability tree for a given object can be accessed by certain operations. Table 4 gives their semantics.

cap\_info – Returns the size of capability cap's object window, and the value of its moneyword.  
 derive\_cap – Derives a new, perhaps more restricted, capability.  
 delete\_cap – Deletes the capability and all of its derivatives.  
 rename\_obj – Delete all capabilities for an object and create a new master capability.

Table 4. Capability tree operations

| Operation                                     | Semantics  |
|---|--|
| proc.cap_info(cap)                            | <pre> if (pays(proc, cost<sub>cap-info</sub>) AND check_pass(cap,proc) ≠ null AND has(cap,CAP_INFO)) { (v,s,p1,p2)=cap; size=wend<sub>v,s,p1</sub> - wstart<sub>v,s,p1</sub>; return (size, money<sub>v,s,p1</sub>) } else return false; </pre>  |
| proc.derive_cap(cap, rights_mask, start, end) | <pre> if (pays(proc, cost<sub>derive-cap</sub>) AND check_pass(cap,proc) ≠ null AND has(cap,DERIVE_CAP)) { (v,s,p1,p2) = cap; p1' = rand({x[] {0,1}<sup>PLEN</sup> : C<sub>v,s,x</sub> = null}); p2' = rand({0,1}<sup>PLEN</sup>); C<sub>v,s,p1'</sub> = (p2', money<sub>v,s,p1</sub>, p1, ∅, max(start, wstart<sub>v,s,p1</sub>), min(end, wend<sub>v,s,p1</sub>), rights<sub>v,s,p1</sub> [] rights_mask); children<sub>v,s,p1</sub> = children<sub>v,s,p1</sub> [] p1'; return (v,s,p1',p2' ⊕ check_pass(cap,proc)) } else return false; </pre>   |
| proc.delete_cap(cap)                          | <pre> if (pays(proc, cost<sub>delete-cap</sub>) AND check_pass(cap,proc) ≠ null AND has(cap,DELETE_CAP)) { (v,s,p1,p2)=cap; tmp={p1}; while (tmp ≠ ∅) { p1'=rand{tmp}; tmp=tmp [] children<sub>v,s,p1'</sub>; C<sub>v,s,p1'</sub>=null } } else return false; </pre>   |
| proc.rename_obj(cap)                          | <pre> if (pays(proc, cost<sub>rename-obj</sub>) check_pass(cap,proc) null AND has(cap,RENAME_OBJ)) { (v,s,p1,p2) = cap; if (parent<sub>v,s,p1</sub> ≠ null) return false; p1' = rand({x[] {0,1}<sup>PLEN</sup> : C<sub>v,s,x</sub> = null}); p2' = rand({0,1}<sup>PLEN</sup>); C<sub>v,s,p1'</sub> = C<sub>v,s,p1</sub>; caps<sub>v,s,p1'</sub> = {(p2', money<sub>v,s,p1</sub>, null, ∅, wstart<sub>v,s,p1</sub>, wend<sub>v,s,p1</sub>, rights<sub>v,s,p1</sub>)}; tmp = {p1}; while (tmp ≠ ∅) { p1'' = rand{tmp}; tmp = tmp [] children<sub>v,s,p1''</sub>; C<sub>v,s,p1''</sub> = null } } return (v,s,p1',p2' ⊕ check_pass(cap,proc)) } else return false; </pre> |

## 4.5 Status operations

A number of operations are possible on processes. Table 5 gives their semantics.

withdraw – Withdraws a sum of money and places it in the cashword of the caller.

deposit – Deposits a sum of money.

suspend – Suspends a process.

resume – Resumes a suspended process.

apply\_lock – Confines a process.

send – Sends a message, and some cash to a process.

receive – Retrieves a message.

wait – Waits for a message.

revive – Revives a terminated process.

Table 5. Status operations

| Operation                  | Semantics   |
|----------------------------|---|
| proc.withdraw(cap, sum)    | <pre> if (pays(proc, cost<sub>withdraw</sub>) AND has(cap, WITHDRAW) AND check_pass(cap,proc)≠null) { (v,s,p1,p2) = cap; (v',s') = proc; tmp = p1;   do { if (money<sub>v,s,tmp</sub> &lt; sum) return false;       else tmp = parent<sub>v,s,tmp</sub> } while (tmp≠null);   cash<sub>v,s,p1</sub> += sum; tmp = p1;   do { money<sub>v,s,p1</sub> -= sum; tmp = parent<sub>v,s,tmp</sub> }   while (tmp≠null);   money<sub>v,s,p1</sub> -= sum; return true } else return false; </pre> |
| proc.deposit(cap, sum)     | <pre> if (pays(proc, cost<sub>deposit</sub>) AND has(cap, DEPOSIT) AND check_pass(cap,proc)≠null) { (v,s,p1,p2) = cap; (v',s') = proc;   if (cash<sub>v,s,p1</sub> &lt; sum) return false;   cash<sub>v,s,p1</sub> -= sum; money<sub>v,s,p1</sub> += sum;   return true } else return false; </pre>   |
| proc.suspend(cap)          | <pre> if (pays(proc, cost<sub>suspend</sub>) AND has(cap, SUSPEND) AND check_pass(cap, proc)≠null AND is_process(cap)) { (v,s,p1,p2) = cap; suspended<sub>v,s,p1</sub> = true } else return false; </pre>   |
| proc.resume(cap)           | <pre> if (pays(proc, cost<sub>resume</sub>) AND has(cap, RESUME) AND check_pass(cap, proc)≠null AND is_process(cap)) { (v,s,p1,p2) = cap; suspended<sub>v,s,p1</sub> = false } else return false; </pre>  |
| proc.apply_lock(cap, lock) | <pre> if (pays(proc, cost<sub>apply-lock</sub>) AND has(cap, APPLY_LOCK) AND check_pass(cap, proc)≠null AND is_process(cap)) { (v,s,p1,p2) = cap; lock<sub>v,s</sub> = lock<sub>v,s</sub>⊕lock } else return false; </pre>  |
| proc.send(cap, msg, sum)   | <pre> if (pays(proc, cost<sub>send</sub>) AND has(cap, SEND) AND check_pass(cap,proc)≠ null AND is_process(cap)) { (v,s,p1,p2) = cap; (v',s') = proc;   (msgs) = mailbox<sub>v,s</sub>;   if (sum &lt; 0 OR cash<sub>v,s</sub> &lt; sum) return false;   mailbox<sub>v,s</sub> = (msgs, (msg,sum));   cash<sub>v',s'</sub> -= sum; waiting<sub>v,s</sub> = false } else return false; </pre>  |
| proc.receive()             | <pre> if (pays(proc, cost<sub>receive</sub>) { (v,s) = proc; ((msg, sum), msgs) = mailbox<sub>v,s</sub>;   cash<sub>v,s</sub> += sum; mailbox<sub>v,s</sub> = (msgs);   return msg } else return false; </pre>  |
| proc.wait()                | <pre> if (pays(proc, cost<sub>wait</sub>) { (v,s) = proc; waiting<sub>v,s</sub> = true;} else return false; </pre>  |
| proc.revive(cap, sum)      | <pre> if (pays(proc, cost<sub>revive</sub>) AND has(cap, REVIVE) AND check_pass(cap, proc)≠null AND is_process(cap)) { (v,s,p1,p2) = cap; (v',s') = proc;   if (terminated<sub>v,s</sub> == false OR sum = 0   OR sum &lt; cash<sub>v',s'</sub>) { return false }   else {terminated<sub>v,s</sub> = false;         cash<sub>v',s'</sub> -= sum; cash<sub>v,s</sub> += sum } else return false; </pre>  |

## 5. CONCLUSION

An outline of the operational semantics of the Password-Capability System has been presented. This system gives us a much more flexible and secure basis for operating systems than conventional Unix or Windows based systems. While the capability paradigm is not new, this is the first time that a formal description of the important Password-Capability System has been presented. The formal semantics described herein can be applied to a formal security analysis of such a system. This work is expected to be extensible to the second generation Password-Capability System, WALNUT (Castro, 1996), and possibly also to other password-capability based systems such as Mungi (Heiser et al, 1998) and Opal (Chase et al, 1992).

## REFERENCES

- Anderson, M., 1986. *A Password Capability System*. Ph.D. thesis, Monash University, Australia.
- Anderson, M., Pose, R. D. and Wallace, C. S., 1986. A Password-Capability System. *The Computer Journal*, Vol. 29, No. 1, pp. 1-8.
- Anderson, M. and Wallace, C. S., 1988. Some Comments on the Implementation of Capabilities. *The Australian Computer Journal*, Vol. 20, No. 3, pp. 122-130.
- Castro, M. D., 1996. *The Walnut Kernel: A Password-Capability Based Operating System*. Ph.D. thesis, Monash University, Australia.
- Chase, J. S. et al, 1992, A Single Address Space System for 64-bit Architectures. *Proceedings of the Third Workshop on Workstation Operating Systems*. Key Biscayne, FL, pp. 80-85.
- Cohen, E. and Jefferson, D., 1975, Protection in the Hydra Operating System. *Proceedings of the Fifth ACM Symposium on Operating System Principles*. Colorado, USA, pp. 141-160.
- Dennis, J. B. and Van Horn, E.C., 1966. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, Vol. 9, No. 3, pp. 143-155.
- Heiser, G. et al, 1998. The Mungi Single-Address-Space Operating System. *Software Practice and Experience*, Vol. 28, No. 9, pp. 901-928.
- Lampson, B. W., 1973. A Note on the Confinement Problem. *Communications of the ACM*, Vol. 16, No. 9, pp. 613-615.
- Lipner, S. B., 1975, Protection in the Hydra Operating System. *Proceedings of the Fifth ACM Symposium on Operating System Principles*. Colorado, USA, pp. 192-196.
- Myers, G. J. and Buckingham, B. R. S, 1980. A Hardware Implementation of Capability-Based Addressing. *ACM SIGARCH Computer Architecture News*, Vol. 8, No. 6, pp. 12-24.
- Wallace, C. S. and Pose, R. D., 1990, Charging in a Secure Environment. *Security and Persistence*. Bremen, Germany, pp. 85-97.