A Language-based Approach to Analysing Flow Security Properties in Virtualised Computing Systems

Chunyan Mu
Department of Computer Science, Teesside University, UK
c.mu@tees.ac.uk

***Abstract*—This paper studies the problem of reasoning about flow security properties in virtualised computing networks with mobility from perspective of formal language. We propose a distributed process algebra $\mathtt{CSP}_{4v}$ with security labelled processes for the purpose of formal modelling of virtualised computing systems. Specifically, information leakage can come from observations on process executions, communications and from cache side channels in the virtualised environment. We describe a cache flow policy to identify such flows. A type system of the language is presented to enforce the flow policy.**

***Keywords*-language-based security, information flow control, cache non-interference, virtualised computing systems.**

I. Introduction

In cloud systems, computing resources are shared among multiple clients. This is achieved by virtualisation in which a collection of virtual machines (VMs) are running upon the same platform under the management of the hypervisor. However, such services can also bring additional channel threat, and introduce information leakage between unrelated entities during the procedure of *resource sharing* and *communications* through unintended covert channels. To address this concern, this paper proposes to develop formal approaches to specifying, modelling and analysing flow security properties in virtualised computing networks.

Specifically, information leakage can come from observations on both program executions and cache usage in the virtualised environment. On the one hand, for processes running upon a particular VM instance, consider the processes are communication channels, sensitive inputs can be partially induced by observing public outputs of the processes regarding choices of public inputs. On the other hand, shared caches enable competing VM instances to extract sensitive information from each other. More precisely, for processes running upon different VM instances, cache usage can be considered as a communication channel. Consider the cache lines accessed by the victim instance and by the malicious instance as high level input and low level input respectively, by observing the usage (such as time) of victim cache lines (low output), the malicious observer can learn some information of the victim instance. In summary, this paper considers distributed virtualised computing environment where the attacker VM steals the information from the target one by observing executions of victim processes, and by probing and measuring the usage (timing) of the shared cache.

In particular, we develop an approach from the software language-based level to enforce the applications to access shared cache and to bring interferences in a predictable way. As a result, we aim to prevent the leakage introduced by such cache timing channel and the interference between security objects caused by executions. First, we propose a CSP-like language for modelling communicating processes running upon VMs with mobility in the computing environment. Second, we formalise a cache flow policy to specify the security condition regarding the threat model we focus on. Finally, we describe a type system of the language to enforce the flow policy and control the leakage introduced by observing the system behaviours. A full version of this paper [1] provides more details and proofs of the Theorems.

II. The Modelling Language $\mathtt{CSP}_{4v}$

This section presents a dialect of communicating sequential processes (CSP) [2] language $\mathtt{CSP}_{4v}$ for formal modelling of and reasoning about virtualised computing network systems considered in this paper. Such an environment can be considered as a distribute computing system, in which a group of inter-connected and virtualised computers are dynamically allocated for serving. Processes can move from one VM to another and communicate to each other via sending/receiving messages. Applications and data are stored and processed in the network but can be accessed from any location using a client. It is natural to specify and describe the system as a set of communicating processes in a network with consideration of resource sharing in a predictable way.

A. Terminology and notation

We consider the infrastructure consists of a set of *virtual private networks* (VPNs) upon which a set of *virtual machines* (VMs) can communicate with each other. A VPN may include one or more VMs, and the location of VMs can be viewed as a node (host) of the VPN it belongs to. An *instance* is a VM upon which a number of *processes* locate and run. Let I denote a set of instances, $I = \{I, I_1, \ldots, I_n\}$. Processes $(P, P_1, \ldots)$ can be constructed from a set of atomic actions, or be composed using operators to create

978-1-4799-7492-4/15/$31.00 ©2017 IEEE

Expressions	$\exp ::= \texttt{Vars} \mid \textsc{Ch} \mid \textsc{Ints} \mid \exp \oplus \exp\ (\oplus \in \{+, -, *, /\})$
	$b ::= \texttt{true} \mid \neg b \mid b \wedge b \mid \exp \bowtie \exp\ (\bowtie \in \{>, \geq, <, \leq, ==\})$
Process	$x := \exp \mid \texttt{STOP} \mid \texttt{SKIP} \mid \texttt{SLEEP}(\Delta t) \mid$
	$\texttt{MOVE}_P(i) \mid P;Q \mid P \lhd b \rhd Q \mid b \rhd (P)^* \mid$
	$a!w \mid a?x \mid P \| Q$
VM instances	$I ::= [\![P]\!] \mid \texttt{MOVE}_I(h) \mid I \| I'$
Hosts	$H ::= i : [\![I]\!].M_I \mid H \| H'$
VPN networks	$G ::= h : [\![H]\!] \mid G \| G'$

Table I
SYNTAX OF $\texttt{CSP}_{4v}$

more complex behaviours. The full set of actions that a system may perform is called the *alphabet* (Σ). The operators are required to obey algebraic laws which can be used for formal reasoning. The interactions carrying data values between processes take place through "channels". From an information theory point of view, a storage device such as cache which can be received from (reading) and sent to (writing) is also a kind of communication channel. CPU data caches locate between the processor cores and the main memory. We assume the clients (including the attacker) know the map between memory locations and cache sets, so we omit the details of the mapping and focus on cache organisation and operation here. Cache can be viewed as a set of cache lines: $\texttt{CLines} = \{l_i | 0 \leq i \leq n\}$.

In addition, in order to encode the desired features of the language for flow secure virtualised computing systems, we assign security labels into variables and cache lines (channels), and allocate cache lines into instances via mappings:

$$\tau_v : \texttt{Vars} \mapsto_{\tau_v} \mathcal{L},\ \tau_c : \texttt{CLines} \mapsto_{\tau_c} \mathcal{L},\ \alpha_c : \texttt{CLines} \mapsto_{\alpha_c} I$$

where $\mathcal{L}$ denotes a security lattice. We write τ in stead τ_v and τ_c in general for cases without introducing any confusion. Furthermore, we consider VM hosts are assigned to different categories Ω_H, with an ordering of subset relations:

$$\beta_h : \texttt{Hosts} \mapsto_{\beta_h} \Omega_H.$$

For instance, h_1 is assigned to category: $\Omega_1 = \{\texttt{student}\}$, h_2 is assigned to category: $\Omega_2 = \{\texttt{student, staff}\}$, and thus $h_1 \sqsubseteq h_2$ since $\Omega_1 \subseteq \Omega_2$. Similarly, we also assign VM instances into different sub-categories Ω_I, with an ordering of subset relations:

$$\beta_i : \texttt{VMs} \mapsto_{\beta_i} \Omega_I.$$

For instance, P running upon VM i_1 belonging to sub-category: $\omega_1 = \{\texttt{UG-1}\} \subseteq \texttt{student}$, Q running upon VM i_2 belonging to sub-category: $\omega_2 = \{\texttt{UG-1, UG-2}\} \subseteq \texttt{student}$, and $\omega_1 \subseteq \omega_2$, so $i_1 \sqsubseteq i_2$.

B. Syntax

Table I presents the syntax of the language $\texttt{CSP}_{4v}$. Expressions can be variables, channels, integers and arithmetic operations (denoted by $\oplus$) between expressions.

Operator $x := \exp$ assigns the value of $\exp$ to process variable x. Action operator $\texttt{STOP}$ denotes the inactive processes that does nothing and indicates a failure to terminate, and delayable operator $\texttt{SLEEP}(\Delta t)$ allows the process to do nothing and wait for Δt time units. Moving operator $\texttt{MOVE}_P(i)$ allows to move a process P from current VM to another one i. We require that VM processes running upon on a VM instance with lower (category) order are not allowed to move to a VM instance with a higher (category) order. Operator $P;Q$ denotes the sequential composition of processes P and Q. Branch operator $P \lhd b \rhd Q$ defines if the boolean expression b is true then behaves like P otherwise behaves like Q. Sending operator $a!(w)$ will output a value in expression w over channel a to an agent, and receiving operator $a?x$ allows us to input a data value during an interaction over channel a and write it into variable x of an agent. $P \| Q$ denotes the synchronous parallel composition of processes P and Q. Loop operator $b \rhd (P)^*$ denotes the loop operation of process P while b is true.

An instance I is a virtual machine (VM) hosted on a network infrastructure. Operator $[\![P]\!]$ defines the VM upon which process P runs. Operator $\texttt{MOVE}_I(h)$ allows VM instances (with all processes running on it) to migrate from current host machine to another host h to keep the instance running even when an event, such as infrastructure upgrade or hardware failure, occurs. Similarly to the process movement, we require that VM instances running upon on a host with lower (category) order are not allowed to move to a host with a higher (category) order. So, in the previous example, instances running upon h_1 are not allowed to moved to h_2, and P is not allowed to move to VM i_2. Operator $i : [\![I]\!].M_I$ defines the host machine upon which I runs, M_I denotes the cache pages allocated to instance I. To ensure that no cache is shared among different VM instances, we require that for any host h upon which any I_1 and I_2 are running: $I_1 \neq I_2 \Rightarrow M_{I_1} \cap M_{I_2} = \emptyset$, and if an instance I terminates, then set M_I to be $\emptyset$ for future allocation to other instances. Virtual private network provides connectivity for VM hosts. It can be viewed as a virtual network consisting of a set of hosts where VM instances can run and communicate with each other. The location of a host $h : [\![H]\!]$ indicates a network node of G in which the VM host machine locates.

C. Operational semantics

In order to incorporate as much parallel executions of events within different nodes as possible, we transform the network G into a finite parallel compositions of the form:

$$\begin{array}{rl} G = & h_1 : i_{11} : [\![P_{11}]\!].M_{11} \ \| \ \ldots \ \| \ h_1 : i_{1j_1} : [\![P_{1j_1}]\!].M_{1j_1} \\ \| & h_2 : i_{21} : [\![P_{21}]\!].M_{21} \ \| \ \ldots \ \| \ h_2 : i_{2j_2} : [\![P_{2j_2}]\!].M_{2j_2} \\ \| & \ldots \ \ \ldots \ \ \ldots \\ \| & h_m : i_{m1} : [\![P_{m1}]\!].M_{m1} \ \| \ \ldots \ \| \ h_m : i_{mj_m} : [\![P_{mj_m}]\!].M_{mj_m} \end{array}$$

where h_k denotes the identifier of a host, i_{kj} denotes the VM instance located at h_k. Each component $h_k : i_{kj} : [\![P_{kj}]\!].M_{kj}$ is considered as a *decomposition* of G. We argue such decomposition is well-defined by applying the following rules of structural equivalence:

$G \parallel G' \equiv G' \parallel G$

$(G \parallel G') \parallel G'' \equiv G \parallel (G' \parallel G'')$

$h : [\![I \parallel I']\!] \equiv h : [\![I' \parallel I]\!] \equiv h : [\![I]\!] \parallel h : [\![I']\!]$

$h : [\![(I \parallel I') \parallel I'']\!] \equiv h : [\![I \parallel (I' \parallel I'')]\!]$

$i : [\![P \parallel P']\!].M_i \equiv i : [\![P' \parallel P]\!].M_i \equiv i : [\![P]\!].M_i \parallel i : [\![P']\!].M_i$

$i : [\![(P \parallel P') \parallel P'']\!].M_i \equiv i : [\![P \parallel (P' \parallel P'')]\!].M_i$

We now define the operational semantics of $\texttt{CSP}_{4v}$ in terms of multiset labelled transition system $\langle \vec{\Gamma}, \Sigma, \Longrightarrow \rangle$:

- $\vec{\Gamma}$ is a vector of configurations of a VPN G regarding the vector of decomposition of G. A configuration Γ, regarding a single component (say process P) of the decompositions of G, is defined as a tuple (σ, δ, I, H):
  - $\sigma : \texttt{Vars}_P \mapsto \textsc{Ints}$ denotes the store;
  - $\delta : \texttt{CAddr}_P \mapsto (\textsc{Ints} \cup \{\emptyset\})$ defines the possible world regarding cache;
  - I specifies the owner (the VM instance identifier) of process P;
  - H specifies the host (location) of the VM instance of process P.
- Σ is a set of operating events which the processes can perform;
- $\Longrightarrow \subseteq \vec{\Gamma} \times \vec{\Sigma} \times \vec{\Gamma}$ is the multiset transition relations: $\vec{\Sigma}$ denotes a vector of operating events for the vector of components.

The action rules of the operational semantics of $\texttt{CSP}_{4v}$ is presented in Table II. Notations $\Rightarrow$, $\Leftarrow$, $\Downarrow$ denote cache addressing, cache allocation, and evaluation respectively. For instance, $\Gamma \vdash \exp \Downarrow v$ means that under configuration Γ, exp evaluates to value v, $w \Leftarrow \texttt{CAddr}_a$ means the cache allocated for expression w locates at $\texttt{CAddr}_a$, and $a \Rightarrow \texttt{CAddr}_a$ means cache address of channel a is $\texttt{CAddr}_a$, notation $\texttt{CAddr}_P$ is used to denote the cache addresses allocated for P

Store is defined as a mapping from variables to values, i.e., $\sigma : \texttt{Vars} \mapsto \textsc{Ints}$. Cache is considered as a mapping from addresses (of cache lines) to integers (cached) or $\emptyset$ (flushed), i.e., $\delta : \texttt{CAddr} \mapsto (\textsc{Ints} \cup \{\emptyset\})$.

Action rule of assignment $\langle x := \exp, \Gamma \rangle$ updates the configuration such that the state of x is the value v of expression exp after the execution. Action rule of process moving operator $\texttt{MOVE}_P(i')$ updates the configuration such that the identifier of instance accommodating P turns to be i' after the execution of the process movement. Similar action rule is applied for VM instance movement from one host to another. Action rule of sending operator $a!(w)$ updates the configuration such that the value stored in cache address ($\texttt{CAddr}_a$) is v, if expression w evaluates to v under configuration before the execution and the address of cache allocated for communicating channel a (to store expression w) is $\texttt{CAddr}_a$. Rule of receiving operator $a?x$ updates the configuration such that the state of variable x is v, if the value stored in cache address $\texttt{CAddr}_a$ of the communicating channel is v under the configuration before receiving the data. In the cross-VM communications over today's common virtualised platforms, the cache transmission scheme requires the sender and receiver could only communicate by interleaving their executions for security concerns. In order to capture timing behaviour of cache-related operations, we consider the cache-related operations, such as communicating, as time-sensitive behaviours whose lasting time is recorded. We use $\Delta t(e)$ to denote the time duration of event e lasting for. The behaviour of a process component can now be viewed as a set of sequences of *timed runs*.

Definition 1 (Timed run): A timed run of a component of G is a sequence of timed configuration event pairs leading to a final configuration:

$$\lambda = \langle \Gamma_0, (e_0, \Delta t_0) \rangle \to \cdots \to \langle \Gamma_n, (e_n, \Delta t_n) \rangle \to \Gamma$$

where, Γ_0 and Γ denote the initial and final configuration respectively. $\forall i \in \{0, n\}$, e_i is an event will take place with passing time Δt_i under configuration Γ_i; if $\Delta t_i = 0$, e_i is considered as an immediate event, if $\Delta t_i > 0$, e_i is considered as a time-sensitive event with lasting time Δt_i.

Store	$S ::= \{\} \mid \{\texttt{Vars} \mapsto \textsc{Ints}\} \cup S$		
Cache	$M ::= \{\} \mid \{\texttt{CAddr} \mapsto (\textsc{Ints} \cup \{\emptyset\})\} \cup M$		
Stop	$\langle \texttt{STOP}, \Gamma \rangle \longrightarrow \Gamma[\delta(\texttt{CAddr}_P) = \emptyset]$		
Skip	$\langle \texttt{SKIP}, \Gamma \rangle \longrightarrow \Gamma$		
Sleep	$\frac{\Delta t > 0}{\langle \texttt{SLEEP}(\Delta t), \Gamma \rangle \longrightarrow \langle \texttt{SLEEP}(\Delta t - 1), \Gamma \rangle}$	$\frac{\Delta t = 0}{\langle \texttt{SLEEP}(\Delta t), \Gamma \rangle \longrightarrow \Gamma}$	
Assignment	$\frac{\Gamma \vdash \exp \Downarrow v}{\langle x := \exp, \Gamma \rangle \longrightarrow \Gamma[\sigma(x) = v]}$		
Move	$\frac{\Gamma \vdash I \Downarrow i}{\langle \texttt{MOVE}_P(i'), \Gamma \rangle \longrightarrow \Gamma[I = i']}$	$\frac{\Gamma \vdash H \Downarrow h}{\langle \texttt{MOVE}_I(h'), \Gamma \rangle \longrightarrow \Gamma[H = h']}$	
Seq	$\frac{\langle P, \Gamma \rangle \longrightarrow \langle P', \Gamma' \rangle}{\langle P; Q, \Gamma \rangle \longrightarrow \langle P'; Q, \Gamma' \rangle}$	$\frac{\langle P, \Gamma \rangle \longrightarrow \Gamma'}{\langle P; Q, \Gamma \rangle \longrightarrow \langle Q, \Gamma' \rangle}$	
Branch	$\frac{\Gamma \vdash b \Downarrow \texttt{true}}{\langle P \lhd b \rhd Q, \Gamma \rangle \longrightarrow \langle P, \Gamma \rangle}$	$\frac{\Gamma \vdash b \Downarrow \texttt{false}}{\langle P \lhd b \rhd Q, \Gamma \rangle \longrightarrow \langle Q, \Gamma \rangle}$	
Send	$\frac{\Gamma \vdash \sigma(w) \Downarrow v \quad w \Leftarrow \texttt{CAddr}_a}{\langle a!(w), \Gamma \rangle \longrightarrow \Gamma[\delta(\texttt{CAddr}_a) = v]}$		
Recv	$\frac{a \Rightarrow \texttt{CAddr}_a \quad \Gamma \vdash \delta(\texttt{CAddr}_a) \Downarrow v}{\langle a?x, \Gamma \rangle \longrightarrow \Gamma[\sigma(x) = v]}$		
Loop	$\langle b \rhd (P)^*, \Gamma \rangle \longrightarrow \langle (P; b \rhd (P)^*) \lhd b \rhd \texttt{SKIP}, \Gamma \rangle$		
Parallel	$\frac{P \longrightarrow P'}{P \Vert Q \longrightarrow P' \Vert Q}$	$\frac{I \longrightarrow I'}{I \Vert I'' \longrightarrow I' \Vert I''}$	$\frac{H \longrightarrow H'}{H \Vert H'' \longrightarrow H' \Vert H''}$

Table II
OPERATIONAL SEMANTICS OF $\texttt{CSP}_{4v}$

III. Information Flow Policy

We consider computing environment where malicious tenants can use observations on process executions and on the usage of shared cache to induce information about victim tenants. We assume the service provider and the applications running on the victim's VM are trusted. Let us consider the attacker owns a VM and runs a program on the system, and the victim is a co-resident VM that shares the host machine with the attacker VM. In particular, there are two ways in which an attacker may learn secrets from a victim process: by probing the caches set and measuring the time to access the cache line (through the cache timing channel) - say channel **C1**, and by observing how its own executions are influenced by the executions of victim processes - say channel **C2**.

Example 1 (C1): Consider $\mathtt{VM}_1$ (victim VM, labelled i_1) and $\mathtt{VM}_2$ (attacker VM, labelled i_2) be two instances running upon $\mathtt{Host}_1$ (labelled h_1); victim processes P and Q, running over $\mathtt{VM}_1$, are communicating to each other: P generates a key and sends it to Q, Q encrypts a message using the received key and sends the encrypted message to P, P receives the message and decrypts it; and attacker process R, running over $\mathtt{VM}_2$, keeps probing cache address of the key. Let `keyGen`, `encrypt` and `decrypt` denote the function of generating a key, encryption and decryption respectively, and assume $\mathtt{cread}(\mathtt{CAddr})$ is a function probes cache address $\mathtt{CAddr}$: returns 1 if it is available and returns -1 otherwise. We present the model in our language as follows.

$$\begin{aligned}
\mathtt{VPC}_1 &\overset{\text{df}}{=} h_1 : [\![\mathtt{Host}_1]\!] \parallel h_2 : [\![\mathtt{Host}_2]\!] \\
\mathtt{Host}_1 &\overset{\text{df}}{=} i_1 : [\![\mathtt{VM}_1]\!].M_{\mathtt{VM}_1} \parallel i_2 : [\![\mathtt{VM}_2]\!].M_{\mathtt{VM}_2} \\
\mathtt{VM}_1 &\overset{\text{df}}{=} [\![P \| Q]\!] \qquad \mathtt{VM}_2 \overset{\text{df}}{=} [\![R]\!] \\
P &\overset{\text{df}}{=} x := \mathtt{keyGen}();\;\; \mathtt{key}!x \to P' \\
Q &\overset{\text{df}}{=} \mathtt{key}?y \to Q' \\
Q' &\overset{\text{df}}{=} m_1 := \mathtt{encrypt}(y, \text{“message”});\;\; \mathtt{msg}!m_1 \to \mathtt{STOP} \\
P' &\overset{\text{df}}{=} \mathtt{msg}?m_2 \to P'' \\
P'' &\overset{\text{df}}{=} m := \mathtt{decrypt}(x, m_2) \to \mathtt{STOP} \\
R &\overset{\text{df}}{=} \mathtt{true} \rhd (z := \mathtt{cread}(\mathtt{CAddr}_{\mathtt{key}}))^*
\end{aligned}$$

The communication time between P and Q is affected by the value of the key generated. There is information flow from victim VM to malicious VM through cache side channel.

Example 2 (C2): Consider process P and R are running upon a VM instance VM. Process P inputs a password through channel `pwd` into H-level variable x, and updates L-level variable y to be 1 if x is odd and to be 0 if x is even. Process R output variable y through channel `res`:

$$\begin{aligned}
\mathtt{VM} &\overset{\text{df}}{=} [\![P \| R]\!] \\
P &\overset{\text{df}}{=} \mathtt{pwd}?x;\;\; y := 1 \lhd (x \mod 2 == 1) \rhd y := 0 \to \mathtt{STOP} \\
R &\overset{\text{df}}{=} \mathtt{res}!(y) \to \mathtt{STOP}
\end{aligned}$$

Assume $L \sqsubset H \in \mathcal{L}$. Clearly there are implicit flows from x to y by observing L-level output of the process.

Information flow is controlled by means of security labels and flow policy integrated in the language. Each of the identifiers, information container, is associated with a security label. Identifiers can refer to variables, communication channels, and can refer to entities such as files, devices in a concrete level. The set of the security labels forms a security lattice regarding their ordering. We study the system flow policy which prevents information flow leakage from high-level objects to lower levels and from a target instance to a malicious one via observing process executions and cache usage (by measuring the time of accessing cache lines during communications).

In general, information flow policies are proposed to ensure that secret information does not influence publicly observable information. An ideal flow policy called Non-interference (NI) [3] is a guarantee that no information about the sensitive inputs can be obtained by observing a program's public outputs, for any choice of its public inputs. Intuitively, the NI policy requires that low security users should not be aware of the activity of high security users and thus not be able to deduce any information about the behaviours of the high users. On the one hand, for processes running upon a particular VM instance (regarding **C2**), the NI policy can be applied to control information flow from high-level input to low-level output, where state of sensitive information container (s.a. high-level variables) and observations on behaviours of public information container (s.a. low-level variables) are viewed as the high input and low output respectively. On the other hand, for processes running upon different VM instances (regarding **C1**), we adapt the NI policy here in order to control the information flow from processes running upon victim instance to malicious one through cache side channel. Consider the cache side channel as a communication channel, the cache lines accessed by the victim instance and by the malicious instance are viewed as high level input and low level input respectively, and the observations on the victim cache usage (s.a. hits/misses) are considered as low level outputs. *Cache flow non-interference* demands the changing of the cache lines accessed by the victim process (high inputs) does not affect the public observations on the cache usage (low outputs). Informally, cache flow interference happens if the usage (we focus on the accessing time) of the cache lines accessed by one victim process affects the usage of the cache lines accessed by attacker VM processes.

Formally, the policy of flow non-interference can be considered in terms of the equivalence relations on the system behaviours from the observer's view, including the state evolution of information container and the timing behaviour of cache accessing. This is due to the fact that the system behaviours are modelled as timed runs with security classification of identifiers and with timing considerations when accessing caches during the process communications in our model.

Definition 2 (Flow security environment): Let $\mathcal{L}$ be a finite flow lattice, $\sqsubseteq$ denote the ordering relation of $\mathcal{L}$, I denote the set of VM instances running upon any host h, Ω_H and Ω_I denote a set of categories for hosts and sub-categories for instances respectively. The flow security environment is considered as:

$$\Xi : (\tau_v, \tau_c, \alpha_c, \beta_h, \beta_i),$$

where $\tau_v : \texttt{Vars} \mapsto_{\tau_v} \mathcal{L}$, $\tau_c : \texttt{CLines} \mapsto_{\tau_c} \mathcal{L}$, $\alpha_c : \texttt{CLines} \mapsto_{\alpha_c} I$, $\beta_h : \texttt{Hosts} \mapsto_{\beta_h} \Omega_H$, $\beta_i : \texttt{VMs} \mapsto_{\beta_i} \Omega_I$. Furthermore, we say $\Xi \sqsubseteq \Xi'$ *iff* $\forall x \in \texttt{Vars}$, $l \in \texttt{CLines}$, $i \in \texttt{VMs}$ and $h \in \texttt{Hosts}$: $\Xi(\tau_v(x)) \sqsubseteq \Xi'(\tau_v(x)) \wedge \Xi(\tau_c(l)) \sqsubseteq \Xi'(\tau_c(l)) \wedge \Xi(\beta_h(h)) \sqsubseteq \Xi'(\beta_h(h)) \wedge \Xi(\beta_i(i)) \sqsubseteq \Xi'(\beta_i(i))$, and for $t \in \mathcal{L}$, $\omega_I \subseteq \Omega_I$, and $\omega_H \subseteq \Omega_H$, we say $\Xi \sqsubseteq (t, \omega_I, \omega_H)$ *iff*: $\Xi(\tau_v(x)) \sqsubseteq t \wedge \Xi(\tau_c(l)) \sqsubseteq t \wedge \Xi(\beta_h(h)) \sqsubseteq \omega_H \wedge \Xi(\beta_i(i)) \sqsubseteq \omega_I$, where we abuse notation $\Xi(\tau_v(x))$, $\Xi(\tau_c(l))$, $\Xi(\beta_h(h))$ and $\Xi(\beta_i(i))$ to denote $\tau_v(x)$, $\tau_c(l)$, $\beta_h(h)$ and $\beta_i(i)$ in environment Ξ respectively.

Definition 3 ((t, ω_I, ω_H)-equivalent configuration): Consider processes running upon hosts of VPN G, let Ξ be a security environment, $t \in \mathcal{L}$ be a security level, $\omega_H \subseteq \Omega_H$ be a category and $\omega_I \subseteq \Omega_I$ be a sub-category. For any $x \in \texttt{Vars}$, $l \in \texttt{CLines}$, assume i and h are the instance and host which x, l belong to, we define store (t, ω_I, ω_H)-equivalence under Ξ as follows: $\sigma_1(x) =_{\Xi,(t,\omega_I,\omega_H)} \sigma_2(x)$ *iff*: $(\Xi(\tau_v(x)) \sqsubseteq t \wedge \Xi(\beta_i(i)) \sqsubseteq \omega_I \wedge \Xi(\beta_h(h)) \sqsubseteq \omega_H) \Rightarrow \sigma_1(x) = \sigma_2(x)$, and cache line (t, ω_I, ω_H)-equivalence under Ξ as follows: $\delta_1(l) =_{\Xi,(t,\omega_I,\omega_H)} \delta_2(l)$ *iff*:

$$\begin{aligned} &(\Xi(\tau_c(l)) \sqsubseteq t \ \wedge\ \Xi(\beta_i(i)) \sqsubseteq \omega_I \ \wedge\ \Xi(\beta_h(h)) \sqsubseteq \omega_H) \\ \Rightarrow\quad & \delta_1(l) = \delta_2(l). \end{aligned}$$

Furthermore, given two configurations $\Gamma_1 = (\sigma_1, \delta_1, i_1, h_1)$ and $\Gamma_2 = (\sigma_2, \delta_2, i_2, h_2)$, we say $\Gamma_1 =_{\Xi,(t,\omega_I,\omega_H)} \Gamma_2$ *iff*:

$$\begin{aligned} & (\forall x \in \texttt{Vars}.\sigma_1(x) =_{\Xi,(t,\omega_I,\omega_H)} \sigma_2(x)) \\ \wedge\quad & (\forall l \in \texttt{CLines}.\delta_1(l) =_{\Xi,(t,\omega_I,\omega_H)} \delta_2(l)) \\ \wedge\quad & (\beta_i(i_1) \sqsubseteq \beta_i(i_2)) \ \wedge\ (\beta_h(h_1) \sqsubseteq \beta_h(h_2)). \end{aligned}$$

Definition 4 (Strong and weak bisimulation): Consider two timed runs running upon host h under security environment Ξ:

$$\lambda = \langle \Gamma_0, (e_0, \Delta t_0) \rangle \to \cdots \to \langle \Gamma_n, (e_n, \Delta t_n) \rangle \to \Gamma$$

$$\lambda' = \langle \Gamma'_0, (e'_0, \Delta t'_0) \rangle \to \cdots \to \langle \Gamma'_n, (e'_n, \Delta t'_n) \rangle \to \Gamma'$$

$\forall \Gamma_0, \Gamma'_0$ such that $\Gamma_0 =_{\Xi,(t,\omega_I,\omega_H)} \Gamma'_0$, we say λ and λ' are *strong (t, ω_I, ω_H)-bisimilar* to each other, i.e., $\lambda \overset{s}{\sim}_{\Xi,(t,\omega_I,\omega_H)} \lambda'$, iff:

$$\forall j \in \{0...n\}.(\Gamma_j =_{\Xi,(t,\omega_I,\omega_H)} \Gamma'_j) \wedge (\Delta t_j = \Delta t'_j);$$

and say λ and λ' are *weak (t, ω_I, ω_H)-bisimilar* to each other, i.e., $\lambda \overset{w}{\sim}_{\Xi,(t,\omega_I,\omega_H)} \lambda'$, iff:

$$(\Gamma =_{\Xi,(t,\omega_I,\omega_H)} \Gamma') \wedge (\sum_{j=0}^{n} \Delta t_j = \sum_{j=0}^{n} \Delta t'_j).$$

Definition 5 (Cache flow security policy): Given a security level $L \in \mathcal{L}$, $\omega_H \subseteq \Omega_H$, and $\omega_I \subseteq \Omega_I$ a VPN G under security environment Ξ is considered *strong cache flow secure iff*:

$$\forall \lambda, \lambda' \in \Lambda.(\Gamma_0 =_{\Xi,(L,\omega_I,\omega_H)} \Gamma'_0 \Rightarrow \lambda \overset{s}{\sim}_{\Xi,(L,\omega_I,\omega_H)} \lambda'),$$

where Γ_0 and Γ'_0 denote the initial configuration of λ and λ' respectively, Λ denotes all runs of components of G. Similarly, the definition of *weak cache flow secure* can be given.

Example 3: Consider the model presented in Example 1. Let $\tau_v(x) = \tau_v(y) = H$, $\tau_v(m_1) = \tau_v(m_2) = M$, $\tau_v(z) = L$, and $L \sqsubset M \sqsubset H \in \mathcal{L}$. Let us assume $\beta_i(\texttt{VM}_1) = \beta_i(\texttt{VM}_2) = \omega_I$, and $\beta_h(\texttt{Host}_1) = \omega_H$. Communicating cache channels are thus assigned with security labels regarding the data they transmit: $\tau_c(\texttt{CAddr}_{\texttt{key}}) = H$, $\tau_c(\texttt{CAddr}_{\texttt{msg}}) = M$. Note that the state of variable z depends on the state of cache address of key, and is affected by the communication time for data transmission between P and Q. Therefore the model does not satisfy the cache flow security policy since for any given two runs, both the timing condition and configuration equivalent condition of $\overset{w}{\sim}_{\Xi,(L,\omega_I,\omega_H)}$ are not guaranteed to be satisfied.

In order to close the cache timing channel, we consider the communication as a scenario of sending and receiving processes running in parallel with certain time interleaving data transmission scheme:

$$\frac{\Gamma \vdash w \Downarrow v \quad w \Leftarrow \texttt{CAddr}_a \quad a \Rightarrow \texttt{CAddr}_a \quad t < T}{\begin{array}{l}\langle \texttt{SLEEP}(T - t) \to P \ \lhd\ (t := \Delta t(a!w \parallel a?x) < T) \ \rhd\ \texttt{STOP}, \Gamma \rangle \\ \xrightarrow{a(w)} \langle P, \Gamma[\sigma(x) = v, \delta(\texttt{CAddr}_a) = \emptyset] \rangle\end{array}}$$

The communicating procedure needs to complete in T time units (together with sleeping time) and then behaves as P; the value of w is sent to variable x via channel a, channel a and the relevant cache lines are then tagged as $\tau_c(w)$. The communication will be considered as failed if T time units have passed but the communication has not completed yet. Fixed completion time T prevents the timing leakage introduced by the cache channel communication.

Example 4: Consider the model presented in Example 1, we rewrite the communicating procedure as follows:

$$\begin{array}{l} x := \texttt{keyGen}(); \\ \texttt{SLEEP}(5 - t) \to P' \lhd (t := \Delta t(\texttt{key}!x \parallel \texttt{key}?y) < 5) \rhd \texttt{STOP} \\ m_1 := \texttt{encrypt}(y, \text{“message”}); \\ \texttt{SLEEP}(5 - t) \to Q' \lhd (t := \Delta t(\texttt{msg}!m_1 \parallel \texttt{msg}?m_2) < 5) \rhd \texttt{STOP} \end{array}$$

The timing condition of weak (t, ω_I, ω_H)-bisimulation specified in Definition 4 is now ensured, while the configuration condition is still violated. This will be addressed in next Section.

IV. Flow Security Type System for $\texttt{CSP}_{4v}$

In order to make the low observation and cache accessing time of the executions be high input independent, the variables and cache lines are associated with security labels, VMs and hosts are assigned to categories, rules (semantic + typing) are required to ensure that: no information flows to lower level objects, no cache is shared among different VM instances, processes (c.f. instances) are not allowed to move from a lower order instance (c.f. host) to a higher one, and cache related operations in communications between processes are forced to be completed in certain time, and hence the cache flow policy is enforced.

For a process P (w.r.t. a component of G), we consider the type judgements have the form of:

$$(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi'$$

where the type $(\tau, \omega_I, \omega_H)$ denotes the (environment) *counter security levels* of the communication channel/variables and *and counter categories* of VMs/hosts participated in the branch events being executed for the purpose of eliminating implicit flows from the *guard*. Ξ and Ξ' describe the type environment which hold before and after the execution of P. In general, notation:

$$\Xi \vdash (\exp :_{\tau_v} t_e, l :_{\tau_c} t_l, l :_{\alpha_c} i_l, i :_{\beta_i} \omega_I, h :_{\beta_h} \omega_H)$$

describes that under type environment Ξ, expression exp and (the address of) cache line l has type t_e and t_l respectively, l is allocated to VM instance i_l, instance i is assigned to a category ω_I, and host h is assigned to a category ω_H. The type of an expression including boolean expression is defined by taking the least upper bound of the types of its free variables as standard:

$$\Xi \vdash \exp :_{\tau_v} t \text{ iff } t = \sqcup_{x \in \texttt{fv}(\exp)} \Xi(\tau_v(x)).$$

All memory, caches and channels written by a t-level expression becomes tagged as t-level. Let $\texttt{Vars}_P$ and $\texttt{CLines}_P$ denote a set of variables defined in and cache lines allocated to process P. Typing rules for processes with security configuration are presented in Table III. Details and proofs of the Theorems are provided in [1] due to page limit.

In overall, the derivation rule $(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi'$ ensures that:

- variables and cache lines whose final types in Ξ' are less than τ must not be changed by P;
- the final value of a variable (or a cache line) say x whose final type is $\Xi'(x) = t$ must not depend on the initial values of those variables (or cache lines) say z whose initial type $\Xi(z)$ is greater than t.
- processes (c.f. instances) belonging to a higher order category instances (c.f. host) must not move to an instance (c.f. a host) with a lower order category.

Theorem 1 (Monotonicity): Given $\mathcal{L}$ and P, for all τ, ω_I, ω_H and Ξ, the type environment transition function: $\mathcal{T}_{P,\mathcal{L}}(\Xi, (\tau, \omega_I, \omega_H)) \mapsto \Xi'$ regarding $(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi'$ is monotone.

Definition 6 (Semantic flow security condition): We say the semantic relation of P satisfies $(\tau, \omega_I, \omega_H)$-flow security property (denoted by $\phi_{\tau,\omega_I,\omega_H}$), written as: $(\Xi\{P\}\Xi') \models \phi_{\tau,\omega_I,\omega_H}$, iff:

i) for all Γ, Γ', x and l:

$$\begin{aligned} &\langle P, \Gamma\rangle \Downarrow \Gamma' \;\wedge\; \Xi' \sqsubset (\tau, \omega_I, \omega_H) \\ \Rightarrow\;\; &\Gamma(\sigma(x)) = \Gamma'(\sigma(x)) \;\wedge\; \Gamma(\delta(l)) = \Gamma'(\delta(l)); \end{aligned}$$

ii) and for all $t \in \mathcal{L}$, $o \in \Omega_I$, $o' \in \Omega_H$, Γ_1 and Γ_2:

$$\Gamma_1 =_{\Xi,(t,o,o')} \Gamma_2 \;\Rightarrow\; \Gamma_1' =_{\Xi,(t,o,o')} \Gamma_2'.$$

We say the flow security type system is sound if the well-typed system is flow secure, more precisely, whenever $(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi'$ then the semantic relation of P is flow secure.

Theorem 2 (Soundness of the flow security type system): The type system proposed in Table III is sound, i.e.,

$$(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi' \Rightarrow (\Xi\{P\}\Xi') \models \phi_{\tau,\omega_I,\omega_H}.$$

Theorem 3 (Flow secure of communications): Given a $\texttt{CSP}_{4v}$ model $\mathbf{A}$, for all P running upon any VM i of any host h in $\mathbf{A}$, if $(\Xi\{P\}\Xi') \models \phi_{\tau,\omega_I,\omega_H}$, then $\mathbf{A}$ is weak cache flow secure with $L = \tau$.

Example 5: Consider again the model presented in Example 1 and 3. It is clear that $(\tau, \omega_I, \omega_H) \vdash \Xi\{R\}\Xi'$ does not hold since the assignment to z make $\Xi'(z) > \Xi(z)$; while $\Xi\{R\}\Xi' \models \phi_{\tau,\omega_I,\omega_H}$ holds if the communication between P and Q fails and $\Xi\{R\}\Xi' \not\models \phi_{\tau,\omega_I,\omega_H}$ holds otherwise.

V. Related Work and Conclusions

This paper relates to the topic of information flow analysis in virtualised computing systems from perspective of formal languages with a concern of cache timing attacks.

Cross-VM side-channel attacks in virtualised infrastructure allowed the attacker to extract information from a target VM and stole confidential information from the victims [4], [5], [6]. Over the last decade, there have been a sustained effort in exploring solutions to defend cache channel attacks in virtualised computing environment via the approaches of *cache partition* at either *hardware-level* [7], [8], [9], [10] or *system-level* [11], [12], [13], [14], [15], and the approaches of *cache randomisation* via introducing randomization in cache uses through either new *hardware design* [16], [7], [8], [17], [18], [19] or *compiler-assistant design* [20], [18], [21], [22]. More recently, Liu et. al. [15] developed CATalyst to defend cache-based side channel attacks for the cloud

(TSub) $$\frac{\tau_1 \vdash \Xi_1\{P\}\Xi_1'}{\tau_2 \vdash \Xi_2\{P\}\Xi_2'} \quad \tau_2 \sqsubseteq \tau_1,\ \Xi_2 \sqsubseteq \Xi_1,\ \Xi_1' \sqsubseteq \Xi_2'$$

(TAssign) $$\frac{\Xi \vdash \exp :_{\tau_v} t}{(\tau, \omega_I, \omega_H) \vdash \Xi\{x := \exp\}\Xi'(x \mapsto_{\tau_v} \tau \sqcup t)}$$

(TStop) $$(\bot_{\mathcal{L}}, \bot_{\Omega_I}, \bot_{\Omega_H}) \vdash \Xi\{\mathtt{STOP}_P\}\Xi'(\{l \mapsto_{\tau_c} \bot \mid \forall l \in \mathtt{CLines}_P\})$$

(TSkip) $$(\bot_{\mathcal{L}}, \bot_{\Omega_I}, \bot_{\Omega_H}) \vdash \Xi\{\mathtt{SKIP}\}\Xi$$

(TMove) $$\frac{\Xi \vdash (\{x :_{\tau_v} t_x, l :_{\tau_c} t_l, l :_{\alpha_c} i \mid \forall x \in \mathtt{Vars}_P, l \in \mathtt{CLines}_P\}),\ i :_{\beta_i} \omega,\ i' :_{\beta_i} \omega'}{\begin{array}{c}(\tau, \omega_I, \omega_H) \vdash \Xi\{\mathtt{MOVE}_P(i')\}\Xi'(\{x \mapsto_{\tau_v} t_x \sqcup \tau, l \mapsto_{\tau_c} t_l \sqcup \tau, l \mapsto_{\alpha_c} i' \\ \mid \forall x \in \mathtt{Vars}_P, l \in \mathtt{CLines}_P\},\ i \mapsto_{\beta_i} \sqcup\{\omega, \omega', \omega_I\})\end{array}}$$

$$\frac{\begin{array}{l} h : [\![I]\!].M_I \quad I = \langle i : [\![P_1]\!],\ i : [\![P_2]\!],\ \ldots,\ i : [\![P_n]\!]\rangle \\ h' : [\![I']\!].M_{I'} \quad I' = \langle i' : [\![P_1']\!],\ i' : [\![P_2']\!],\ \ldots,\ i' : [\![P_n']\!]\rangle \\ \Xi \vdash h :_{\beta_h} \omega \quad \Xi \vdash h' :_{\beta_h} \omega' \\ (\tau, \omega_I, \omega_H) \vdash \Xi_1\{\mathtt{MOVE}_{P_1}(i')\}\Xi_1' \quad \ldots \quad (\tau, \omega_I, \omega_H) \vdash \Xi_n\{\mathtt{MOVE}_{P_n}(i')\}\Xi_n' \end{array}}{(\tau, \omega_I, \omega_H) \vdash \langle \Xi_1, \Xi_2, \ldots, \Xi_n\rangle\{\mathtt{MOVE}_I(h')\}\langle \Xi_1', \Xi_2', \ldots, \Xi_n'\rangle(h \mapsto \sqcup\{\omega, \omega', \omega_H\})}$$

(TSeq) $$\frac{(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi' \quad (\tau, \omega_I, \omega_H) \vdash \Xi'\{Q\}\Xi''}{(\tau, \omega_I, \omega_H) \vdash \Xi\{P; Q\}\Xi''}$$

(TBranch) $$\frac{\begin{array}{l}\Xi \vdash b :_{\tau_v} t \\ (t \sqcup \tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi_P' \\ (t \sqcup \tau, \omega_I, \omega_H) \vdash \Xi\{Q\}\Xi_Q'\end{array}}{(\tau, \omega_I, \omega_H) \vdash \Xi\{P \lhd b \rhd Q\}\Xi'} \quad \Xi' = \Xi_P' \sqcup \Xi_Q'$$

(TSend) $$\frac{\Xi \vdash w :_{\tau_v} t \quad w \Leftarrow \mathtt{CAddr}_a}{(\tau, \omega_I, \omega_H) \vdash \Xi\{a!w\}\Xi'(\mathtt{CAddr}_a \mapsto_{\tau_c} \tau \sqcup t)}$$

(TRecv) $$\frac{\Xi \vdash \mathtt{CAddr}_a :_{\tau_c} t}{(\tau, \omega_I, \omega_H) \vdash \Xi\{a?x\}\Xi'(x \mapsto_{\tau_v} \tau \sqcup t)}$$

(TLoop) $$\frac{\begin{array}{l}\Xi_i \vdash b :_{\tau_v} t_i \\ (t_i \sqcup \tau, \omega_I, \omega_H) \vdash \Xi_i\{P\}\Xi_i' \quad i = 0, \ldots, n\end{array}}{(\tau, \omega_I, \omega_H) \vdash \Xi\{b \lhd (P)^*\}\Xi_n'} \quad \Xi_0 = \Xi,\ \Xi_{i+1} = \Xi_i' \sqcup \Xi,\ \Xi_{n+1} = \Xi_n'$$

(TPar) $$\frac{\begin{array}{l}(\tau, \omega_I, \omega_H) \vdash \Xi\{P\}\Xi' \quad (\tau, \omega_I, \omega_H) \vdash \Xi\{Q\}\Xi'' \\ \Xi' \vdash (\{x :_{\tau_v} t_x', l :_{\tau_c} t_l' \mid \forall x \in \mathtt{Vars}, l \in \mathtt{CLines}\}) \\ \Xi'' \vdash (\{x :_{\tau_v} t_x'', l :_{\tau_c} t_l'' i \mid \forall x \in \mathtt{Vars}, l \in \mathtt{CLines}\})\end{array}}{(\tau, \omega_I, \omega_H) \vdash \Xi\{P \| Q\}\Xi'''(\{x \mapsto_{\tau_v} t_x' \sqcup t_x'', l \mapsto_{\tau_c} t_l' \sqcup t_l'' \mid \forall x \in \mathtt{Vars}, l \in \mathtt{CLines}\})}$$

Table III
TYPING RULES FOR PROCESSES WITH SECURITY CONFIGURATION.

computing system. However, these efforts mostly require significant changes to the hardware, hypervisors, or operating systems, which make them impractical to be deployed in current cloud data centres. Formal treatment on flow security policies upon side-channel attack detection, leveraging program analysis techniques and relevant tools are still needed in order to improve the accuracy and applicability of leakage analysis and control in the virtualised infrastructure. In this paper, we address the flow security issue in virtualised computing environment from perspective of *programming language* and *program analysis* techniques.

On the other hand, there have been lasting investigations on flow property specification and enforcement via approaches of *formal language and analysis*. The conception of information flow specifies the security requirements of the system where no sensitive information should be released to the observer during its executions. Goguen and Meseguer [3] formalised the notion of absence of information flow with the concept of non-interference. Ryan and Schneider [23] took a step towards the generalisation of a CSP formulation of non-interference to handle information flows through the concept of process equivalence. Security type systems [24], [25], [26], [27], [28], [29], [30] had been substantially used to formulate the analysis of secure information flow in programs. In addition to type-based treatments of secure information flow analysis for programs, Clark *et. al* presented a flow logic approach in [31], Amtoft and Banerjee proposed a Hoare-like logic for program dependence in [32]. Hammer and Snelting [33] presented an approach for information flow control in program analysis based on program dependence graphs (PDG). However, none of the above works has addressed the issue of flow analysis in virtualised computing systems regarding observations on both executions of communicating processes and affections of cache timing channels.

We have proposed a language-based approach for information leakage analysis and control in virtualised computing infrastructure with a concern of cache timing attacks. Specifically, we have introduced a distributed process algebra with processes able to capture flow security characters in a virtualised environment; we have described a cache flow policy for leakage analysis through communication covert channels; and we have presented a type system of the language to enforce the flow policy. In our future work, we plan to derive concrete implementation of our approach and to extend the current model to define and enforce more practical security policies.

References

[1] C. Mu, "Analysing flow security properties in virtualised computing systems," *CoRR*, vol. 2004.05500, 2020. [Online]. Available: http://arxiv.org/abs/2004.05500

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[3] J. Goguen and J. Meseguer, "Security policies and security models," in *S & P*, 1982, pp. 11–20.

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *CCS*, 2009, pp. 199–212.

[5] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX*, 2012, pp. 159–173.

[6] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *CCS*, 2012, pp. 305–316.

[7] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007, pp. 494–505.

[8] ——, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008, pp. 83–93.

[9] L. Domnitser, A. Jaleel, J. Loew, N. B. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *TACO*, vol. 8, no. 4, pp. 35:1–35:21, 2012.

[10] J. Kong, O. Aciiçmez, J. Seifert, and H. Zhou, "Architecting against software cache-based side-channel attacks," *IEEE Trans. Computers*, vol. 62, no. 7, pp. 1276–1288, 2013.

[11] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *CCSW*, 2009, pp. 77–84.

[12] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *IEEE/IFIP DSN-W*, 2011, pp. 194–199.

[13] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *USENIX*, 2012, pp. 189–204.

[14] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *CCS*, 2016, pp. 871–882.

[15] F. Liu, Q. Ge, Y. Yarom, F. McKeen, C. V. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016, pp. 406–418.

[16] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *ACSAC*, 2006, pp. 473–482.

[17] F. Liu and R. B. Lee, "Random fill cache architecture," in *MICRO*, 2014, pp. 203–215.

[18] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *ASPLOS*, 2015, pp. 87–101.

[19] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Design and implementation of the ascend secure processor," *IEEE Trans. Dependable Sec. Comput.*, vol. 16, no. 2, pp. 204–216, 2019.

[20] M. M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE Trans. Cloud Computing*, vol. 2, no. 4, pp. 395–408, 2014.

[21] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *NDSS*, 2015.

[22] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX*, 2015, pp. 431–446.

[23] P. Y. A. Ryan and S. A. Schneider, "Process algebra and non-interference," in *CSFW*, 1999, pp. 214–227.

[24] D. M. Volpano and G. Smith, "A type-based approach to program security," in *TAPSOFT*, 1997, pp. 607–621.

[25] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL*, 1999, pp. 228–241.

[26] K. Honda, V. T. Vasconcelos, and N. Yoshida, "Secure information flow as typed process behaviour," in *ESOP*, 2000, pp. 180–199.

[27] F. Pottier, "A simple view of type-secure information flow in the p-calculus," in *CSFW*, 2002, pp. 320–330.

[28] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL*. ACM Press, January 2006, pp. 79–90.

[29] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk, "Session types for access and information flow control," in *CONCUR*, 2010, pp. 237–252.

[30] S. Hunt and D. Sands, "From exponential to polynomial-time security typing via principal types," in *ESOP*, 2011, pp. 297–316.

[31] D. Clark, C. Hankin, and S. Hunt, "Information flow for algol-like languages," *Comput. Lang.*, vol. 28, no. 1, pp. 3–28, 2002.

[32] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *SAS*, 2004, pp. 100–115.

[33] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009.