
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 CSP\textsubscript{4e} 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 CSP\textsubscript{4e}

This section presents a dialect of communicating sequential processes (CSP) [2] language CSP\textsubscript{4e} 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, \(I = \{I_1, I_2, \ldots, I_n\}\). Processes \((P, P_1, \ldots)\) can be constructed from a set of atomic actions, or be composed using operators to create
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: \( \text{CLines} = \{ l \mid 0 \leq i \leq n \} \).

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:

\[ τ_\text{L} : \text{Vars} \mapsto τ_\text{L}, \text{L}, τ_\text{C} : \text{CLines} \mapsto τ_\text{L}, \text{L}, \tau_\text{C} : \text{CLines} \mapsto \alpha_\text{C}, I \]

where \( \text{L} \) denotes a security lattice. We write \( τ \) in stead \( τ_\text{L} \) and \( τ_\text{C} \) in general for cases without introducing any confusion. Furthermore, we consider VM hosts are assigned to different categories \( \Omega_H \), with an ordering of subset relations:

\[ β_h : \text{Hosts} \mapsto β_\text{h}, \Omega_H. \]

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

\[ β_i : \text{VMs} \mapsto β_\text{i}, \Omega_I. \]

For instance, \( P \) running upon VM \( i_1 \) belonging to sub-category: \( ω_1 = \{ \text{UG-1} \} \subseteq \text{student} \), \( Q \) running upon VM \( i_2 \) belonging to sub-category: \( ω_2 = \{ \text{UG-1, UG-2} \} \subseteq \text{student} \), and \( ω_1 \subseteq ω_2 \), so \( i_1 \subseteq i_2 \).

### B. Syntax

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

### 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:

\[ G = h_1 : i_{11} : [P_{11}]M_{113} \mid \ldots \mid h_1 : i_{1j} : [P_{11}]M_{1j3} \mid h_2 : i_{21} : [P_{21}]M_{213} \mid \ldots \mid h_2 : i_{2j} : [P_{21}]M_{2j3} \mid \ldots \mid \ldots \mid h_m : i_{m1} : [P_{m1}]M_{m13} \mid \ldots \mid h_m : i_{mj} : [P_{mj}]M_{mj3} \]
where \( h_k \) denotes the identifier of a host, \( i_{kj} \) denotes the VM instance located at \( h_k \). Each component \( h_k : i_{kj} : \{P_k\} \), \( i_{kj} \) is considered as a decomposition of \( G \). We argue such decomposition is well-defined by applying the following rules of structural equivalence:

\[
G \parallel G' \equiv G' \parallel G \parallel G'' \equiv G \parallel G''
\]

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

\( i : [P \parallel P'].M = i : [P' \parallel P].M \equiv i : [P].M_i \parallel i : [P\parallel P'].M_i \)

We now define the operational semantics of \( \text{CSP}_{4\nu} \) in terms of multiset labelled transition system \( (\Gamma, \Sigma, \Rightarrow) \):

- \( \Gamma \) is a vector of configurations of a VPN \( G \) regarding the vector of decomposition of \( G \). A configuration \( \Gamma \), regarding a single component (say process \( P \)) of the decompositions of \( G \), is defined as a tuple \( (\sigma, \delta, I, H) \):
  - \( \sigma : \text{Vars}_P \mapsto \text{INTS} \) denotes the store;
  - \( \delta : \text{CAddr}_p \mapsto (\text{INTS} \cup \{\emptyset\}) \) 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 \( P \).

- \( \Sigma \) is a set of operating events which the processes can perform;
- \( \Rightarrow \subseteq \Gamma \times \Sigma \times \Gamma \) is the multiset transition relations: \( \Sigma \) denotes a vector of operating events for the vector of components.

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

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

Action rule of assignment \( (x := \exp, \Gamma) \) updates the configuration such that the state of \( x \) is the value of \( \exp \) after the execution. Action rule of process moving operator \( \text{MOVE}_p(i') \) updates the configuration such that the identifier of instance accommodating \( P \) turns to \( i' \) 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 \( \text{CAddr}_a \) 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 \( \text{CAddr}_a \). 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 \( \text{CAddr}_a \) 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 last 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:

\[
\lambda = \{ \Gamma_0, (e_0, \Delta t_0) \} \rightarrow \cdots \rightarrow \{ \Gamma_n, (e_n, \Delta t_n) \} \rightarrow \Gamma
\]

where, \( \Gamma_0 \) and \( \Gamma \) denote the initial and final configuration respectively. \( \forall i \in \{0, n\}, e_i \) is an event will take place with passing time \( \Delta t_i \) under configuration \( \Gamma_i \); 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 \( \Delta t_i \).
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 VM1 (victim VM, labelled i1) and VM2 (attacker VM, labelled i2) be two instances running upon Host1 (labelled h1); victim processes P and Q, running over VM1, 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 VM2, 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 cread(CAddr) is a function probes cache address CAddr; returns 1 if it is available and returns 0 otherwise. We present the model in our language as follows.

VPC
\[ VPC_1 \equiv h_1 : [Host_1] \parallel h_2 : [Host_2] \]
Host1
\[ i_1 : [VM_1].M_{VM_1} \parallel i_2 : [VM_2].M_{VM_2} \]
VM1
\[ P \equiv x := \text{keyGen}(); \ \text{key}x \rightarrow Q' \]
\[ Q \equiv \text{key}?y \rightarrow Q' \]
\[ Q' \equiv m_1 := \text{encrypt}(y, \text{"message"}); \ \text{msg}!m_1 \rightarrow \text{STOP} \]
\[ P' \equiv \text{msg}?m_2 \rightarrow P'' \]
\[ P'' \equiv m := \text{decrypt}(x, m_2) \rightarrow \text{STOP} \]
\[ R \equiv \text{true} \triangleright (z := \text{cread(CAddr)})^* \]

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:

VM
\[ VM \equiv [P]\parallel R \]
\[ P \equiv \text{pwd}\!(x); \ \text{y} := 1 \triangleright (x \mod 2 = 1) \triangleright y := 0 \rightarrow \text{STOP} \]
\[ R \equiv \text{res}!(y) \rightarrow \text{STOP} \]
when accessing caches during the process communications in our model.

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

$$\Xi : (\tau_\nu, \tau_c, \alpha_c, \beta_h, \beta_i),$$

where $\tau_\nu : \text{Vars} \rightarrow \tau_\nu$, $\tau_c : \text{CLines} \rightarrow \tau_c$, $\alpha_c : \text{CLines} \rightarrow \alpha_c$, $\beta_h : \text{Hosts} \rightarrow \beta_h$, $\beta_i : \text{VMS} \rightarrow \beta_i$. Furthermore, we say $\Xi \sqsupseteq \Xi'$ if $\forall \tau \in \text{Vars}, \ l \in \text{CLines}, \ i \in \text{VMS} \ and \ h \in \text{Hosts}: \Xi(\tau_\nu(x)) \sqsupseteq \Xi'(\tau_\nu(x)) \wedge \Xi(\tau_c(l)) \sqsupseteq \Xi'(\tau_c(l)) \wedge \Xi'(\beta_h(h)) \sqsupseteq \Xi'(\beta_h(h)) \wedge \Xi'(\beta_i(i)) \sqsupseteq \Xi'(\beta_i(i)),$

and for $t \in L$, $\omega_I \subseteq \Omega_I$, and $\omega_H \subseteq \Omega_H$, we say $\Xi \sqsubseteq (t, \omega_I, \omega_H)$ iff $\Xi(\tau_\nu(x)) \sqsubseteq t \wedge \Xi(\tau_c(l)) \sqsubseteq t \wedge \Xi(\beta_h(h)) \subseteq \omega_H \wedge \Xi(\beta_i(i)) \subseteq \omega_I$, where we abuse notation $\Xi(x, l)$, $\Xi(\beta_h(h))$ and $\Xi(\beta_i(i))$ to denote $\tau_\nu(x)$, $\tau_c(l)$, $\beta_h(h)$ and $\beta_i(i)$ in environment $\Xi$ respectively.

**Definition 3 ((t, \omega_I, \omega_H)-equivalent configuration):** Consider processes running upon hosts of VPN $G$, let $\Xi$ be a security environment, $t \in L$ be a security level, $\omega_I \subseteq \Omega_I$ be a category and $\omega_H \subseteq \Omega_H$ be a sub-category. For any $x \in \text{Vars}, \ l \in \text{CLines}$, assume $i$ and $h$ are the instance and host which $x, l$ belong to, we define store $(t, \omega_I, \omega_H)$-equivalence under $\Xi$ as follows:

$$\sigma_1(x) = \Xi((t, \omega_I, \omega_H), \sigma_2(x)) \iff \Xi(\tau_\nu(x)) \sqsubseteq t \wedge \Xi(\beta_i(i)) \subseteq \omega_I \wedge \Xi(\beta_h(h)) \subseteq \omega_H \Rightarrow \sigma_1(x) = \sigma_2(x),$$

and cache line $(t, \omega_I, \omega_H)$-equivalence under $\Xi$ as follows:

$$\delta_1(l) = \Xi((t, \omega_I, \omega_H), \delta_2(l)) \iff \delta_1(l) \subseteq \delta_2(l).$$

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

$$(\forall x \in \text{Vars}. \sigma_1(x) = \Xi((t, \omega_I, \omega_H), \sigma_2(x)) \wedge (\forall l \in \text{CLines}. \delta_1(l) = \Xi((t, \omega_I, \omega_H), \delta_2(l)) \wedge (\beta_i(i_1) \subseteq \beta_i(i_2)) \wedge (\beta_h(h_1) \subseteq \beta_h(h_2))).$$

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

$$\lambda = (\Gamma_0, (e_0, \Delta t_0)) \rightarrow \cdots \rightarrow (\Gamma_n, (e_n, \Delta t_n)) \rightarrow \Gamma'$$

$$\lambda' = (\Gamma'_0, (e'_0, \Delta t'_0)) \rightarrow \cdots \rightarrow (\Gamma'_n, (e'_n, \Delta t'_n)) \rightarrow \Gamma''$$

$\forall \Gamma_0, \Gamma'_0$ such that $\Gamma_0 = \Xi((t, \omega_I, \omega_H), \Gamma_0)$, we say $\lambda$ and $\lambda'$ are strong $(t, \omega_I, \omega_H)$-bisimilar to each other, i.e., $\lambda \sim_{\Xi, (t, \omega_I, \omega_H)} \lambda'$, 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 $\lambda$ and $\lambda'$ are weak $(t, \omega_I, \omega_H)$-bisimilar to each other, i.e., $\lambda \sim_{\Xi, (t, \omega_I, \omega_H)} \lambda'$, 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 $L \in L$, $\omega_H \subseteq \Omega_H$, and $\omega_I \subseteq \Omega_I$ a VPN $G$ under security environment $\Xi$ is considered strong cache flow secure iff:

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

where $\Gamma_0$ and $\Gamma_0'$ denote the initial configuration of $\lambda$ and $\lambda'$ respectively, $\Lambda$ 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 $\tau_\nu(x) = \tau_\nu(y) = H$, $\tau_\nu(m_1) = \tau_\nu(m_2) = M$, $\tau_\nu(z) = L$, and $L \subseteq M \subseteq H \subseteq \Omega_I$. Let us assume $\beta_i(VM_1) = \beta_i(VM_2) = \omega_I$, and $\beta_h(Host_{t_1}) = \omega_H$. Communicating cache channels are thus assigned with security labels regarding the data they transmit: $\tau_c(CAddr\_key) = H, \tau_c(CAddr\_msg) = M$. Note that the state of variable $x$ 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 $\sim_{\Xi, (L, \omega_I, \omega_H)}$ 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:

| $\Gamma \vdash w \downarrow \nu \ w \leftarrow CAddr\_a \ a \Rightarrow CAddr \ t < T$ |
|---|---|---|
| (SLEEP($T - t$) $\rightarrow P \leftarrow (t := \Delta(t' \wedge t'') \wedge \sigma(x) < T) \Rightarrow \text{STOP}, \Gamma)$ |
| $a \leftarrow w \downarrow \nu \ \sigma(x) \leftarrow \delta(CAddr) \Rightarrow \text{STOP}$ |

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 $\tau_\nu(x)$. 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:

$x := \text{keyGen}();$

SLEEP($5 - t$) $\rightarrow P' \leftarrow (t := \Delta(t' \wedge (t' < 5)) \Rightarrow \text{STOP}, m_1 := \text{encrypt}(y, \text{"message"});$

SLEEP($5 - t$) $\rightarrow Q' \leftarrow (t := \Delta(t(m_1 \wedge m_2 < 5)) \Rightarrow \text{STOP}$

...
The timing condition of weak \((t, \omega_I, \omega_H)\)-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 CSP<sub>4e</sub>**

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. \(\Xi\) and \(\Xi'\) describe the type environment which hold before and after the execution of \(P\). In general, notation:

\[\Xi \vdash \text{exp} : \tau_{e}, l : \tau_{l}, \ell : \ell_{\omega}, i_{i} : \omega_{I}, h : \omega_{H}\]

describes that under type environment \(\Xi\), expression \(\text{exp}\) and (the address of) cache line \(l\) has type \(\tau_{e}\) and \(\tau_{l}\) respectively, \(l\) is allocated to VM instance \(i_{i}\), instance \(i\) is assigned to a category \(\omega_{I}\), and host \(h\) is assigned to a category \(\omega_{H}\). 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:

\[\Xi \vdash \text{exp} : \tau_{e}\text{ iff } t = \bigcup_{x \in \text{fv}(\text{exp})} \Xi(\tau_{e}(x))\]

All memory, caches and channels written by a t-level expression becomes tagged as t-level. Let \(\text{Vars}_{P}\) and \(\text{CLines}_{P}\) 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 \(\Xi'\) are less than \(\tau\) 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 \(L\) and \(P\), for all \(\tau\), \(\omega_I\), \(\omega_H\) and \(\Xi\), the type environment transition function:

\[T_{P, L}(\Xi, (\tau, \omega_I, \omega_H)) \mapsto \Xi'\] 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: \((\Xi \{P\} \Xi') \vdash \phi_{\tau, \omega_I, \omega_H}\), iff:

i) for all \(\Gamma\), \(\Gamma'\), \(x\) and \(l\):

\[(P, \Gamma) \not\vdash \Gamma' \land \Xi' \in (\tau, \omega_I, \omega_H)\]

\[\Rightarrow (\Gamma(\sigma(x)) = \Gamma'(\sigma(x)) \land \Gamma(\delta(l)) = \Gamma'(\delta(l)))\]

ii) and for all \(t \in L\), \(o \in \Omega_{I}\), \(o' \in \Omega_{H}\), \(\Gamma_{1}\) and \(\Gamma_{2}\):

\[\Gamma_{1} =_{\Xi, (t, o, o')} \Gamma_{2} \Rightarrow \Gamma_{1}' =_{\Xi, (t, o, o')} \Gamma_{2}'\]

We say the flow security system type 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.,

\[(\tau, \omega_I, \omega_H) \vdash \Xi \{P\} \Xi' \Rightarrow (\Xi \{P\} \Xi') \vdash \phi_{\tau, \omega_I, \omega_H}\]

**Theorem 3 (Flow secure of communications):** Given a CSP<sub>4e</sub> model \(A\), for all \(P\) running upon any VM \(i\) of any host \(h\) in \(A\), if \((\Xi \{P\} \Xi') \vdash \phi_{\tau, \omega_I, \omega_H}\), then \(A\) 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 \(\Xi \{R\} \Xi' \not\vdash \phi_{\tau, \omega_I, \omega_H}\) holds if the communication between \(P\) and \(Q\) fails and \(\Xi \{R\} \Xi' \not\vdash \phi_{\tau, \omega_I, \omega_H}\) 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 CAFalyst to defend cache-based side channel attacks for the cloud.
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 [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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule</th>
<th>Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 \vdash \Xi_1(P) \Xi'_1 )</td>
<td>( \tau_2 \vdash \Xi_2(P) \Xi'_2 )</td>
<td>( \tau_2 \subseteq \tau_1 ), ( \Xi_2 \subseteq \Xi_1 ), ( \Xi'_2 \subseteq \Xi'_1 )</td>
<td>( \Xi \vdash \exp \tau_n \ t )</td>
</tr>
<tr>
<td>( \Xi \vdash \exp \tau_n \ t )</td>
<td>( \Xi \vdash \exp \tau_n \ t )</td>
<td>( \Xi \vdash \exp \tau_n \ t )</td>
<td>( \Xi \vdash \exp \tau_n \ t )</td>
</tr>
</tbody>
</table>

Table III: Typing rules for processes with security configuration.
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


