Statically Checking Confidentiality of Shared-Memory Programs with Dynamic Labels

Marcus Völp
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
voelp@os.inf.tu-dresden.de

Abstract

At WITS 2005, Warnier et al. published an algorithm to statically check confidentiality of programs with dynamic labels. Unlike prior approaches, their method allows for temporary breaches of confidentiality. However, they share the commonly made assumption that programs run entirely in private memory. Thus, interaction with and observation of the checked program is restricted to program start and termination respectively.

This paper extends Warnier's approach in two fundamental aspects: shared memory and synchronisation. Through shared memory other programs may observe and interact with the checked program at memory-access granularity. Synchronisation renders parts of the shared memory inaccessible to those programs which adhere to the locking policy. We provide a mechanically-checked soundness proof and show the effectiveness of a countermeasure to the AES cache side-channel attack.

1. Introduction

Over the past years, several language-based approaches have been proposed to statically check confidentiality (for an overview see Sabelfeld et al. [11]). However, most of these assume that programs run entirely in private memory.

One such approach is dynamic labelling by Warnier et al. [5]. Dynamic labelling starts from an initial mapping $lab_0 : A \rightarrow L$, which assigns each variable (at address $a \in A$) a security level $l \in L$. The security levels are ordered in the lattice $(L, \leq)$ with greatest lower bound $\sqcap$ and least upper bound $\sqcup$. Each such level represents the secrecy of information in this variable, respectively, for output variables, the clearance of observers of this variable. By eliminating the concrete values, maintaining only these security levels, dynamic labelling abstractly interprets the checked program (in the following called p) with the help of the labelling functions `LabStat` and `LabExpr`. Confidentiality is validated by checking whether security levels in the final mapping are decreasing.

Introducing shared memory and an abstract form of concurrency control on this memory, we extend Warnier's algorithm to check:

1. Confidentiality of inputs to and secrets in the checked shared-memory program.

2. Confidentiality of data placed by other programs in shared memory. In particular, we check confidentiality of information that combines previously learned secrets.

Before detailing our approach we explain Warnier's algorithm in more detail and briefly sketch a real-life scenario: a microhypervisor-based crypto server. Crucial parts of this scenario we will use to illustrate the applicability of our approach. In particular, we verify the effectiveness of a known countermeasure [8] to the AES cache side-channel attack. A result that previous language-based approaches, every flow-insensitive type system [4] included, cannot establish because it involves a temporal breach of confidentiality in the caches.

1.1. Dynamic Labelling

$$\begin{tikzcd} s \arrow[r, "{[\![p]\!]}"] \arrow[d] & s' \arrow[d] \\ lab \arrow[r, "\texttt{LabStat}(p)"] & lab' \end{tikzcd}$$

0

0-7695-3102-4/08 $25.00 ©2008 IEEE
DOI 10.1109/ARES.2008.56

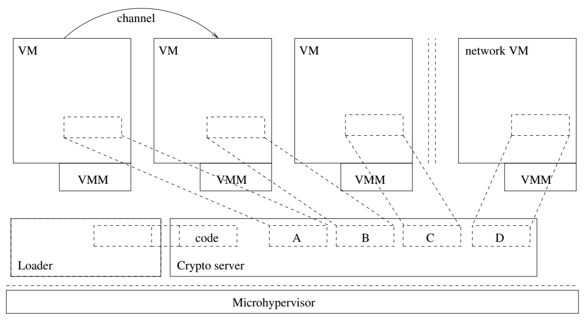

Figure 1. Set of interconnected VMs and the shared memory buffers established with the crypto server.

More precisely, checking confidentiality of a program p using abstract interpretation works as follows (the above diagram shows this procedure): First, the relevant parts from the concrete state s are extracted. In the case of checking confidentiality, we extract only the security level of information stored in the memory variables in s. Thus we obtain the mapping lab. Then, for each state transition $[\![p]\!]$ of the original program p we execute a corresponding transition on the abstract state. The latter is described by the labelling function for the respective statement: $\mathtt{LabStat(p)}$.

To illustrate this procedure, let us consider the simple example $l := 4 + h$ with respective security levels low and $high$.

The initial abstract state of these variables is: $l : low, h : high$. The abstract result state we obtain by application of the respective labelling function $\mathtt{LabStat}(\mathtt{l} := (4 + \mathtt{h}))$ for this statement. This function updates the mapping lab at position l with the security level of the result of the subexpression $(h + 4)$, which is obtained by subsequent application of labelling functions for the subexpressions: $\mathtt{LabExpr}(4)$, $\mathtt{LabExpr}(\mathtt{h})$, and $\mathtt{LabExpr}(\mathtt{h} + 4)$. These specific functions do not modify the abstract state but return as result security level: low, $high$ and $high = low \sqcup high$ respectively. A comparisson of this state $(l : high, h : high)$ with the initial state indicates an illegal information flow because the level of l is not decreasing (or equal). Thus information for which an observer of l is not cleared has been stored in l. The program is not confidential.

Before we show how to extend this algorithm for shared memory programs, we briefly sketch a real-life scenario.

1.2. Crypto Server

A crypto server securely interconnects virtual machines (VMs) over a public network with their counterparts on remote nodes. The server protects the keys used, for example, in the AES encryption [8]. It receives plain-text messages in memory buffers that are shared with the VMs and relays the encrypted messages via a shared-memory buffer to a dedicated network VM. Legitimate communication channels may connect certain VMs bypassing the crypto server.

Figure 1 illustrates this scenario, which is for example envisaged for the μ-Sina Virtual Workstation, a microkernel-based version of the Sina Virtual Workstation [12]. A similar scenario we obtain by offloading crypto-functionality in a heterogeneous multi-core environment. Application fields include secure VPNs [2] and secure banking applications [3]. Both minimise the trusted computing base through untrusted legacy-components reuse.

Compared to a monolithic operating system (OS) solution, the main difference is that here crucial OS functionality is implemented by application-level programs.

The important points illustrated are:

- All memory is, in fact, *shared* with other programs (at least with the OS). Some of these programs must be trusted to preserve p's confidentiality.
- *Private* memory (i.e., exclusively used parts of memory for which confidentiality is preserved) is a guarantee given by these trusted programs.
- Widely unknown programs (e.g., the proprietary guest-OSs inside the VMs) share memory with p.

1.3. Synopsis

The next section introduces our approach. We discuss the extensions to Warnier's algorithm in Section 3 and the AES case study in Section 4. Sections 5 and 6 relate our approach the work of others and conclude this paper.

Throughout this paper, we abstract from the PVS [9] specific syntax. The complete PVS sources for this paper are available [15].

2. Shared Memory

Although private memory is in fact shared, we will model it as exclusively owned by p and assume this guarantee is established separately for the servers providing this memory.

Programs executing in parallel to p may obtain read or read-write privileges to parts of p's memory. In addition, these other programs may have access to private memory and to a set of authorised communication channels bypassing p. Because we cannot generally trust these programs to protect the confidentiality of information, we have to assume that they will exploit all given means to leak information. In Section 3.2, we lift this restriction for programs, whose confidentiality has been established.

Assuming properly configured channels (e.g., confined subsystems with the legitimate exception of communication with p [13]), our concern is to establish:

1. that p does not leak internal information, and

2. that p does not forward information between these otherwise confined subsystems.

To capture this behaviour more formally, let T be the set of other programs. For all $t \in T$, let l_t be t's clearance. Let A be the set of addresses that are accessible by p.

Define for each $t \in T$ the set of addresses that are read-only shared ($RO_t \subseteq A$) respectively read-write shared ($RW_t \subseteq A$) with p. Furthermore, define the relation $canSend \subseteq T \times T$ denoting the set of unidirectional channels.

These sets are properly configured (i.e., there are no unauthorised channels bypassing p through which other programs may communicate) if the following property holds:

Definition 1 *Proper configuration*

$$\begin{array}{l} \forall t, t' \in T, a \in A. \\ \quad a \in RW_t \vee a \in RO_{t'} \cup RW_{t'} \vee t\, canSend\, t' \;\Rightarrow\; l_t \leq l_{t'} \end{array}$$

The restrictions on transitive channels follow from the transitivity of $\leq$.

Thus, in a properly configured system any unauthorised information flow must involve p.

To ease the further discussion, we combine these results into t-independent forms:

$$\begin{array}{l} a \in readable? \Leftrightarrow \exists t \in T.\, a \in RO_t \;\wedge\; \forall t' \in T.\, a \notin RW_{t'} \\ a \in writable? \Leftrightarrow \exists t \in T.\, a \in RW_t \end{array}$$

We combine both shared-memory channels and other communication channels into the relation on addresses: $effects?$. Two addresses are related (i.e., a $effects?$ a') if a program $t' \in T$ may write data to a' that another program $t \in T$ may eventually read from a. This is precisely the case if $a \in RO_t$ and $a' \in RW_{t'}$ and if there is a chain of programs $t = t_0, t_1, ..., t_n = t'$ connected through communication channels (i.e., t_i $canSend$ t_{i+1}) or through shared-memory channels (i.e., $\exists a_c \in RW_{t_i} \cap RO_{t_{i+1}}$).

Using this information we can now determine the security levels of shared-memory variables a in the initial mapping lab_0. We set $lab_0(a) := \underset{t \in \omega}{\sqcap} l_t$. This is the smallest (greatest lower bound) security level of all other programs $t \in \omega \subseteq T$ that can read information from a directly or indirectly through a chain of other programs. Thus, any information with security level smaller than $lab_0(a)$ can safely be written to a.

2.1. Learned Secrets

So far, we have investigated the channels (shared memory and other communication channels) through which information may be leaked immediately. Deferred leakage arises form the fact that other programs may remember secrets in their private memory.

Consider the following simple program:

$$p ::= sh_1 := h; sh_1 := l; sh_2 := sh_1$$

and two other programs q_1 and q_2 with $RW_{q_i} = sh_i$ and $RO_{q_i} = \emptyset$. Assume h contains information which must not leak to q_2.

The only possible way for q_1 to leak data to q_2 is by modifying the shared variable sh_1 immediately before p executes $sh_2 := sh_1$. Lacking internal memory, only values derived from the data in variable l could be leaked since $sh_1 = l$ prior to $sh_2 := sh_1$. p would be confidential with respect to the information in h. Using internal memory, however, q_1 may store the information in h after $sh_1 := h$ and leak a derived value via sh_1 immediately before $sh_2 := sh_1$.

As illustrated in the introductory scenario we do not know in general which code is executed by the programs $t \in T$. More important, we do not know how these programs derive the values from the secrets they learned through shared variables.

Therefore, we have to consider all possible values that may affect p's control and data flows. Let it be an input trace of the type $\mathbb{N} \rightarrow writable? \rightarrow Byte$. We introduce IT as the shortcut for this type. Every input trace contains for each program step ($\in \mathbb{N}$) the values of the writable shared-memory variables prior to p resuming its computation.

These values may possibly have been modified by other programs executing concurrently to p.

Similarly, we record the security levels of these values in a corresponding trace $lt : \mathbb{N} \rightarrow writable? \rightarrow L$ (which we abbreviate as LT).

While it is possible to quantify over all these traces when verifying some property of p in a theorem prover (e.g., soundness of our algorithm), it is far from practical to consider security-level traces when statically checking p [1]. To overcome this problem, we maintain the additional mapping $sec : writable? \rightarrow L$ in the labelling functions. $\texttt{LabStat}(p)(n, ...).sec(a')$ denotes the highest security level of secrets learned during the first n steps of p from addresses a with a $effects?$ a'. Consequently, in all realistic input traces, $lt(n)(a') \leq \texttt{LabStat}(p)(n, ...).sec(a')$ holds for the security level of an input to variable a' in step n. Statically checking confidentiality remains feasible because sec can be evolved stepwise.

[1]Note that abstract interpretation abstracts from the concrete value traces anyway.

2.2. Concurrency Control

Synchronisation primitives restrict when variables can be accessed. Consequently, while a program holds a lock, others cannot read or modify intermediate values in the lock-protected variables. This is provided that all programs adhere to the locking strategy.

In general, not all programs are affected by all concurrency-control primitives. Erroneous or malicious programs may access variables without first acquiring the protecting lock, programs on other CPUs remain unaffected from locally disabled interrupts.

To uniformly handle synchronisation schemes, we abstract from the concrete concurrency-control mechanism. Let $local? \subseteq A$ be the set of addresses that can only be accessed by those other programs that adhere to the concurrency-control mechanism. Thus $local?$ variables become inaccessible while p holds the lock. However, before p disables the concurrency-control mechanism, we have to ensure that secrets temporarily stored in locked variables have been removed. This is done by maintaining the dynamic security levels of lock-protected variables and checking them for decreasingness whenever the lock is released.

We model the program steps during which concurrency control is active as a function $locked? : \mathbb{N} \to bool$. For reasons of simplicity, we consider only equally-locked programs in which this function only depends on the number $n \in \mathbb{N}$ of previously executed instructions [2]. Checking confidentiality for delay-inequivalent programs is kept for future work.

Because most programs do not rely on being preemptible, we can transform most programs into equally-locked programs by adding additional `skip` statements.

Example 1 *For example, the program*

$$p ::= \texttt{if\_then\_else}(e_0)(\uparrow\ v := e_1\ \downarrow)(\texttt{skip})$$

can be transformed into the equally-locked program

$$p' ::= \texttt{if\_then\_else}(e_0)(\uparrow\ v := e_1\ \downarrow)(\uparrow\ \texttt{skip}_{|v:=e_1|}\ \downarrow)$$

where $|v := e_1|$ is the number of steps needed to execute $v := e_1$ and $\uparrow$ (respectively $\downarrow$) denote enabling (and disabling) of the concurrency-control mechanism.

This transformation is however, not generally applicable as some programs rely on certain preemption points.

A preemption-aware spin lock [6], for example, disables preemptions (e.g., the processor interrupts) when the lock is granted to a program and while the latter executes the critical section. During the spinning phase of the lock, preemptions are enabled to minimise interrupt-handling latencies. Because the amount of spinning that is required to get the lock is state dependent, an equally-locked implementation has to disable preemptions during the complete spinning phase, which in turn has to be elongated to a worst-case bound. This, however, jeopardises the interrupt latency and the reason why preemption-aware locks have been used in the first place.

[2]We adjust the n of instructions following a conditional to the maximum n of the respective branches.

3. Dynamic Labelling for Shared Memory Programs

We define the dynamic-labelling rules for a simple imperative programming language P with side effects.

Definition 2 *The syntax of P is given by*

$$\begin{array}{lll} \textit{Expressions} & e ::= & e_1 \circ e_2 \mid v \mid c \mid \texttt{s2e}(s) \\ \textit{Statements} & s ::= & v := e \mid s_1; s_2 \mid \texttt{skip} \mid \texttt{e2s}(e) \mid \\ & & \texttt{if\_then\_else}(e)(s_1)(s_2) \end{array}$$

$v \in A$ ranges over variables, c over constants. $\circ$ stands for the common total binary operators $+, -, ==, \ldots$ that are available, e.g., in C++. The side effects in this language origin from `s2e`, which allows a statement to appear in expressions.

`while` has been deliberately excluded from this language because prior results (e.g., [14]) indicate that in a concurrent setting, `while` may not operate on secret data. A priori bounded `while` loops we unroll to `if_then_else` sequences.

Similarly, we translate accesses into an array of length n into sequences of `if_then_else`. For example the access $a[i] := e$ we write as the sequence:

$$\begin{array}{l} \texttt{if\_then\_else}(i == 0)(v_0 := e)(\\ \ \texttt{if\_then\_else}(i == 1)(v_1 := e)(... \\ \ \ \texttt{if\_then\_else}(i == n-1)(v_{n-1} := e)(\texttt{skip})...)) \end{array}$$

Here v_i is the variable corresponding to the array element $a[i]$.

Statements in P are programs. The semantics of a statement has the type $M \to M$, expressions produce an additional result value. $M : A \to Byte$ is the byte-wise accessed memory. Memory accesses have to be byte wise and larger types have to be combined from their byte-wise representation in memory because other programs may observe updates and modify stored data in shared memory with byte granularity. We do not give a formal semantics of P, since any standard operational semantics with byte-wise memory accesses will suffice for our purpose. For a memory $m \in M$ and an instruction count n let $[\![p]\!]_n(m, it)$ be the result of executing the first n steps of the program p stopping immediately before the $n+1$‘st step. The input trace it is passed to the `skip` statement, which is executed in between any two steps of p to apply the effects of other programs. More precisely, `skip` updates the writable variables with the values obtained from the input trace for the current instruction count.

In our formal definition of confidentiality we deviate from Warnier and use instead the common l-similarity based definition.

Definition 3 *Two memories $m, m' \in M$ are observationally indistinguishable by an l-classified observer if they differ only in higher classified variables (i.e., if $m \sim_{l,lab} m'$ holds):*

$$\begin{aligned}&\sim_{L\times[A\to L]} \subseteq M \times M := \\ &\quad \{(x, y) \in M \times M \mid \forall a : A.\ lab(a) \leq l \Rightarrow x(a) = y(a)\}\end{aligned}$$

We extend l-similarity on memories to l-similarity on input traces in the usual way.

We can now formally define our confidentiality property. In this definition we use the notation $\sim_{l,lab|_S}$ to indicate that l-similarity must hold only for the addresses in the set S.

Definition 4 *A program p with initial variable-to-security-level mapping lab_0 and security-level trace lt is confidential if the following predicate holds:*

$$\begin{aligned}&\mathit{confidential?}(p, lab_0, lt) \Leftrightarrow \\ &\forall l : L, m, m' \in M, it, it' \in IT, n : \mathbb{N}. \\ &\quad m \sim_{l,lab_0} m' \wedge it \sim_{l,lt} it' \Rightarrow \\ &\quad\quad \begin{cases} [\![p]\!]_n(m, it) \sim_{l,lab_0|_{readable?\setminus local?}} [\![p]\!]_n(m', it') \\ \quad \textbf{if}\, \mathit{locked?}(n) \\ [\![p]\!]_n(m, it) \sim_{l,lab_0|_{readable?}} [\![p]\!]_n(m', it') \\ \quad \textbf{otherwise} \end{cases}\end{aligned}$$

Confidentiality states that executing p from two memories and with two input traces, which differ only in high variables ($\not\leq l$), produces the same l-observable outputs after any number of steps n. Hereby, lock protected outputs can only be seen if p does not hold the lock.

3.1. Labelling Functions

Our labelling functions match Warnier's with two exceptions. Each labelling function takes a maximum instruction count $n \in \mathbb{N}$ up to which the statement (or expression) is evaluated. The second exception is the rule $\mathtt{LabStat}(\mathtt{skip})$, in which we maintain the learned secrets sec and in which the effect of other programs is applied to lab according to the trace lt. The signature of the labelling functions thus change to:

$$\begin{aligned}&\mathtt{LabStat}(s) : \mathbb{N} \times \mathit{State} \times L \times \mathit{LT} \to \mathit{State} \\ &\mathtt{LabExpr}(e) : \mathbb{N} \times \mathit{State} \times L \times \mathit{LT} \to \mathit{State} \times L\end{aligned}$$

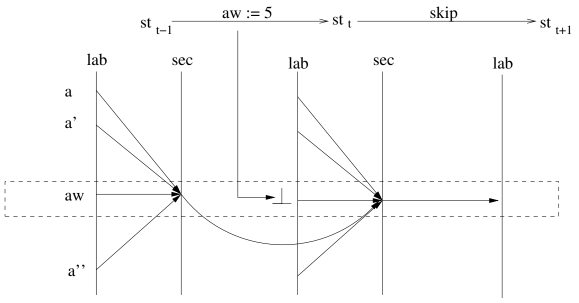

Figure 2. Update of the mappings lab and sec in the labelling function $\mathrm{LabStat}(\mathtt{skip})$.

The security level L in the domain of these functions is the security level $lenv$ of the program counter. Like in Warnier, this parameter is used to identify indirect flows in conditional statements. The record $State$ combines the mappings lab and sec with a current instruction count ic. For $st \in State$, we write $st.sec$ to access the record field sec and $st \setminus (sec) := sec'$ for the partial update of this field with sec'.

Let us here concentrate on the labelling function $\mathbf{LabStat}(\mathtt{skip})$. The remaining rules can be found in the appendix of this paper. They are straight forward extensions of those in [5].

In between any two steps $[\![p]\!]_n(m, it)$ and $[\![p]\!]_{n+1}(m, it)$ writable and not currently lock-protected shared-memory variables are updated according to the information in it. The corresponding labelling function for this update is:

$$\begin{aligned}&\mathtt{LabStat}(\mathtt{skip})(n, st, lenv, lt) := \\ &\quad st\setminus \quad (sec) := \lambda\, a \in \mathit{writable?}(n). \\ &\quad\quad\quad \bigsqcup(st.lab, \mathit{readable?}(n) \cup \{a' | a'\ \mathit{effects?}\ a\}) \sqcup \\ &\quad\quad\quad \bigsqcup(st.sec, \mathit{readable?}(n) \cup \{a' | a'\ \mathit{effects?}\ a\}) \sqcup \\ &\quad\quad\quad st.sec(a), \\ &\quad\quad (lab) := \lambda\, a \in A. \\ &\quad\quad \begin{cases} st.lab(a) \sqcup lt(st.ic)(a) & \textbf{if}\, a \in \mathit{writable?}(n) \\ st.lab(a) & \textbf{otherwise}, \end{cases} \\ &\quad\quad (ic) := st.ic + 1\end{aligned}$$

with $\mathit{readable?}(n) := \mathit{readable?} \setminus \begin{cases} \mathit{local?} & \textbf{if}\, \mathit{locked?}(n) \\ \emptyset & \textbf{otherwise} \end{cases}$ and $\mathit{writable?}(n)$ respectively. We combine the security levels of those variables from which the information origins $a'\ \mathit{effect?}...$ with $\sqcup$. The operand $\bigsqcup(lab, S)$ lifts $\sqcup$ to the set of security levels obtained by evaluating elements $a \in S$ of the set $S \subseteq A$ in lab. Figure 2 illustrates the updates performed in $\mathbf{LabStat}(\mathtt{skip})$ graphically. The security levels of variables from which the inbound arrows origin are combined using $\sqcup$.

Following in principle Warnier's algorithm, we check in each intermediate state whether the then visible (i.e., not lock protected) variables in shared memory have a decreasing security level compared to the level in lab_0. Decreasing security levels mean that during those points at which shared-memory variables are observable only information that is cleared (according to lab_0) for the observers of these variables is stored.

More formally the predicate $decreasing?$ is defined as:

Definition 5 *Decreasing*

$$
\begin{aligned}
&decreasing?(p, st_0, lenv, lt) := \\
&\quad \forall n \in \mathbb{N}. \forall a \in A. readable?(n)(a) \Rightarrow \\
&\qquad \texttt{LabStat}(p)(n, st_0, lenv, lt).lab(a) \leq st_0.lab(a)
\end{aligned}
$$

3.2. Trusted Servers

So far we assumed that no other program ($t \in T$) is trusted not to leak confidential information. Our algorithm, however, establishes precisely this form of trust for the program p. It is therefore desirable to investigate how to connect checked programs through shared memory.

For this purpose, we exploit the special role of the lattice $(\subseteq, P(Var))$ over the power set of program variables. As pointed out by Sands et al. [4], checking programs with this lattice returns precisely how information flows between program variables. Because information may be returned to p via shared-memory variables, we need to connect all $writable?$ variables to *external repeaters*. An external repeater reads from precisely one variable and returns previously read values (and their security levels) into the checked program. In the result mapping lab' of this program, each writable variable gets a security level corresponding to the set of variables from which information flows into this output variable. Thus, when building the relation $effects?$ for p we can now consider these precise channels rather than the pessimistic assumptions presented above.

3.3. Soundness

Soundness of our labelling algorithm, that is, programs our algorithm determines as being confidential are confidential, follows from the following main theorem with the help of an additional proposition.

Theorem 1 *Soundness*

$$
\begin{aligned}
&good?(p) \wedge decreasing?(p, st_0, lenv, lt) \Rightarrow \\
&\quad confidential?(p, lab_0, lt)
\end{aligned}
$$

where st_0 combines lab_0, sec_0 and an initial value for ic.

For statements p, $good?$ is defined as:

Definition 6 *Goodness*

$$
\begin{aligned}
&good?(p) \Leftrightarrow \forall m, m' \in M, it, it' \in IT, lt \in LT. \\
&\quad \forall lenv, l \in L, st \in State, n \in \mathbb{N}. \\
&(m \sim_{l,st.lab} m' \wedge it \sim_{l,lt} it' \Rightarrow \\
&\quad [\![p]\!]_n(m, it) \sim_{l,\texttt{LabStat}(p)(n,st,lenv,lt).lab} [\![p]\!]_n(m', it')) \wedge \\
&(\forall a \in A. \neg lenv \leq \texttt{LabStat}(p)(n, st, lenv, lt).lab(a) \Rightarrow \\
&\quad [\![p]\!]_n(m, it)(a) = [\![skip_{|p|}]\!]_n(m, it)(a))
\end{aligned}
$$

the definition for expressions is similar.

The first conditional of $good?$ means that execution of p maintains in general the l-similarity relation between states but with dynamically changing labels. In the second conditional $good?$ differs fundamentally from the respective predicate in Warnier's work. The latter assumed that $lenv \not\leq lab(a) \Rightarrow [\![s]\!](m)(a) = m(a)$. This is no longer the case as other programs may modify shared variables. Thus, we had to replace this conditional as shown above. A further difference is our restriction of the predicate $decreasing?$ to $readable?(n)$ variables as shown in Definition 5.

The main theorem states that, provided all labelling functions are $good?$, programs whose security levels are decreasing are confidential. The additional proposition (not shown here, see [15]) establishes $good?$ for the labelling functions of statements and expressions in P. The proof of this proposition is straight forward by structural induction over the statements and expressions of P.

In the following we sketch the proof of the main theorem. The proof that our labelling algorithm is sound (Theorem 1) follows closely the soundness proof in Warnier et al. The precise proof is included in [15]. Here we only sketch the basic steps:

Proof

Chose $l \in L$, $m, m' \in M, it, it' \in IT, lt \in LT, st_0 \in State$ such that

1. $st_0.lab = lab_0$, $st_0.sec = sec_0$,
2. both the memory states and the input traces are l-similar: $m \sim_{l,lab_0} m'$ and $it \sim_{l,lt} it'$,
3. the program p terminates in both memories before step max, and
4. in some step $n \leq max$ holds $[\![p]\!]_n(m, it)(a) \neq [\![p]\!]_n(m', it')(a)$ for some address a that is readable by another program in this step (i.e, for which $readable?(n)(a)$).

Then l should at most be $st_0.lab(a)$ to satisfy confidentiality (i.e., $l \leq lab_0(a)$). Goodness gives us $l \leq \texttt{LabStat}(p)(n, st_0, lenv, lt).lab(a)$ where we instantiate $lenv$ with the initial program

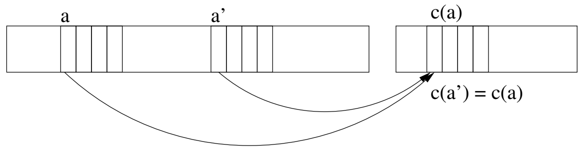

Figure 3. Processor caches are modelled as a disjoint region of shared memory.

counter level. From $decreasing?$ we know that $\texttt{LabStat}(p)(n, st_0, lenv, lt).lab(a) \leq lab_0(a)$. Transitivity of $\leq$ leads to the desired result $l \leq lab_0(a)$. **q.e.d.**

4. Case Study : AES

To illustrate the feasibility of our algorithm we prove a countermeasure to the AES cache side-channel attack to be effective.

A central part of AES involves indexing with the encryption key into pre-computed lookup tables. Osvik et al. [8] describes how to deduce the encryption key by observing these access patterns in the processor caches. A known countermeasure to this attack is to unify the cache pattern of this algorithm by accessing these tables with cachelines strides after the AES lookup.

We argue that cache memory can be modelled as shared memory and thus that our algorithm can be used to check confidentiality of this countermeasure. The process is as illustrated in Section 1.1 except that we evolve the labelling functions stepwise and that we check decreasingness after each step.

Let a be some memory address, $c(a)$ the cache set of a. Other programs may observe an access to a, though not necessarily the value of a, by checking whether conflicting addresses a' (with $c(a') = c(a)$) have been evicted from the cache. We model caches as a separate memory with addresses distinguished from the variables in the programs p and $t \in T$. A cache is shared among programs on the same CPU. Each memory access we transform into an access to the normal variables $a \in A$ followed by a write access which stores a constant to $c(a)$. The constant leaks the program-counter level $lenv$ though not the value at a. Figure 3 illustrates this transformation.

Simplified to the part necessary to illustrate how our algorithm protects confidentiality, the AES encryption program is

$$p ::= \uparrow a[key]; a[0]; ...; a[n] \downarrow$$

where $a[0], ..., a[n]$ are the cacheline-stride accesses. Preemptions are disabled during this sequence (as indicated by $\uparrow, \downarrow$). All programs on the same CPU automatically adhere to this locking scheme because the operating-system scheduler does not execute other programs when preemptions are disabled.

The array access $a[key]$ is unrolled to a sequence of $\texttt{if\_then\_else}(i == key)(a[i], c(a[i]) := 1)(...)$, thus $lenv$ and consequently $lab(c(a[key]))$ assumes the key's security level. The cache's security levels are no longer decreasing. However, because this memory is lock protected and because the stride accesses reduce $lab(c(a[i]))$ to a key independent security level, this temporal breach of confidentiality is repaired.

Note that we did not conclude that the AES encryption preserves confidentiality of the plain text. This requires an explicit downgrading of the ciphertext before its memory location becomes observable.

5. Related Work

This work directly builds on Warnier et al. [5].

Research on confidentiality goes back to the seventies to the work of the Dennings [1]. For a recent overview on the approaches and remaining issues in language-based information-flow security see Sabelfeld and Myers [11].

Others have investigated the impact of OS primitives on information flow including synchronisation primitives Sabelfeld [10], though not the possibility to use lock protected variables for temporal breaches of confidentiality. Volpano et al. [14] introduces an atomic construct with effects similar to our concurrency-control mechanism. O'Neill et al. [7] introduces IO channels which can be used to emulate shared memory. Although the above works achieve similar results, they all check confidentiality with a flow-insensitive security type system. Thus, they cannot tolerate temporary breaches of confidentiality. A feature we showed was required to verify confidentiality of the AES countermeasure.

. Acknowledgements

The author of this work was supported by the European Commission through PASR grant 104600. Further thanks go to Intel Corp. for their support and to the anonymous reviewers for their comments.

6. Conclusions

In this work, we extended Warnier's dynamic labelling algorithm to statically check confidentiality for programs that interact through shared memory with other concurrently-executing programs. Our extensions are completely formalised and proved sound in the theorem prover PVS.

The extended algorithm not only detects whether secrets initially in the checked program are leaked, but also whether this program forwards secrets from other programs. These secrets can be internal or learned during the course of execution.

We applied our algorithm to a central part of the AES encryption algorithm and proved confidentiality for a countermeasure to a cache side-channel attack on the encryption key. This prove was only possible because dynamic labelling allows confidentiality to be temporarily breached.

References

[1] D. Denning. A lattice model of secure information flow. volume 19, pages 236–243, New York, NY, USA, 1976. ACM Press.

[2] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, Mar. 2005.

[3] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[4] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM Press.

[5] B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *WITS '05: Proceedings of the 2005 workshop on Issues in the theory of security*, pages 50–56, New York, NY, USA, 2005. ACM Press.

[6] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler Conscious Synchronization. *ACM Transactions on Computer Systems*, Feb. 1997.

[7] K. O'Neill, M. Clarkson, and s. Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[8] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. 2005.

[9] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[10] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. *Lecture Notes in Computer Science*, 2001.

[11] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[12] secunet. Sina Virtual Workstation . http://www.secunet.com/index.php?id=24&L=3.

[13] J. Shapiro. Verifying the EROS Confinement Mechanism. In *IEEE Symposium on Security and Privacy*, 2000.

[14] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[15] M. Völp. Statically Checking Confidentiality of Shared-Memory Programs with Dynamic Labels - PVS Sources. (http://os.inf.tu-dresden.de/˜voelp/sources/dyn_sm.tgz).

[Labelling Functions]

```
$\texttt{LabExpr}(\texttt{s2e}(s))(n, st, lenv, lt) :=$
  $(\texttt{LabStat}(s)(n, st, lenv, lt), \bot)$

$\texttt{LabStat}(\texttt{e2s}(e))(n, st, lenv, lt) :=$
  $proj\_1(\texttt{LabExpr}(e)(n, st, lenv, lt))$

$\texttt{LabExpr}(c)(n, st, lenv, lt) := (st,\ \bot)$

$\texttt{LabExpr}(v)(n, st, lenv, lt) :=$
  Let $st_r = \texttt{LabStat}(\texttt{skip})(n, st, lenv, lt)$ In
    $(st_r,\ st_r.lab(v))$

$\texttt{LabStat}(s_1; s_2)(n, st, lenv, lt) :=$
  Let $st_r = \texttt{LabStat}(s_1)(n, st, lenv, lt)$ In
    $\begin{cases} \texttt{LabStat}(s_2)(n, st_r, lenv, lt) & \textbf{if } st_r.ic < n \\ st_r & \textbf{otherwise} \end{cases}$

$\texttt{LabStat}(v := e)(n, st, lenv, lt) :=$
  Let $(st_r, l_{res}) = \texttt{LabExpr}(\texttt{e})(n, st, lenv, lt)$ In
    If $st_r.ic < n$ Then
      Let $st'_r = \texttt{LabStat}(\texttt{e2s(e) ; skip})(n, st, lenv, lt)$ In
        $st'_r \ \backslash\ (lab)(v) := lenv \sqcup l_{res}]$
    Else
      $(st_r, l_{res})$
    Endif

$\texttt{LabExpr}(e_1 \circ e_2)(n, st, lenv, lt) :=$
  Let $(st_{r1}, l_{res1}) = \texttt{LabExpr}(e_1)(n, st, lenv, lt)$ In
    If $st_{r1}.ic < n$ Then
      Let $(st_{r2}, l_{res2}) = \texttt{LabExpr}(e_2)(n, r_1`1, lenv, lt)$,
        $ex = \texttt{LabStat}(\texttt{e2s}(\texttt{e}_1\ ;\ \texttt{e}_2)\ ;\ \texttt{skip})(n, st, lenv, lt)$
      In
        $\begin{cases} (ex,\ l_{res1} \sqcup l_{res2}) & \textbf{if } st_{r2}.ic < n \\ (st_{r2}, l_{res2}) & \textbf{otherwise} \end{cases}$
    Else
      $(st_{r1}, l_{res1})$
    Endif

$\texttt{LabStat}(\texttt{if\_then\_else}(e)(s_1)(s_2))(n, st, lenv, lt) :=$
  Let $(st_{re}, l_{res}) = \texttt{LabExpr}(\texttt{e})(n, st, lenv, lt)$,
    $ex = \texttt{LabStat}(\texttt{e2s(e); skip})(n, st, lenv, lt)$ In
    If $ex.ic < n$ Then
      Let $lenv_1 = lenv \sqcup l_{res}$,
          $st_{r1} = \texttt{LabStat}(\texttt{s}_1)(n, ex, lenv_1, lt)$,
          $st_{r2} = \texttt{LabStat}(\texttt{s}_2)(n, ex, lenv_1, lt)$
      In
        $st_{r1}\ \backslash$  $(lab) := st_{r1}.lab \sqcup_{pointwise} st_{r2}.lab$,
                $(sec) := st_{r1}.sec \sqcup_{pointwise} st_{r2}.sec$,
                $(ic := \texttt{max}(st_{r1}.ic, st_{r2}.ic)$
    Else
      $ex$
    Endif
```