Information Flow Security For a Concurrent Language With Lock-based Synchronization

David Miranda[1], Ana Almeida Matos[1,2], and Jan Cederquist[1,2]

[1] Instituto Superior Técnico, Universidade de Lisboa
[2] Instituto de Telecomunicações
{david.domingues.miranda, ana.matos, jan.cederquist}@tecnico.ulisboa.pt

1 Introduction

In today's world, a central problem in computer security is how to ensure data confidentiality or integrity. While encryption [1] and access control [2] are key mechanisms in information security, these mechanisms do not provide the necessary scope and refinement to control how information that programs can access is disseminated during their execution. Static Information Flow Analysis is a standard approach to find dependencies in the code that can leak sensitive information. Denning and Denning [3] extended Bell and LaPadula's work [4] on a certification mechanism for verifying secure information flows in a program, using the properties of lattices [5]. *Type Systems* can ensure that a program satisfies non-interference properties [6, 7] by means of rules that reject program structures that contain insecure dependencies. The idea is that if a program is typable according to a type system that is sound concerning non-interference, then the program is secure, i.e., does not exhibit illegal information flows.

According to Raynal [8] one can implement multiple synchronization primitives based on *semaphores*, a generalization of *locks*. Terauchi [9] is the main work that focuses on information flow security with semaphores as the synchronization primitive. It proposes a type system where the updates to low memory locations cannot be dependent on the schedule decisions and initial values of high memory locations. Terauchi requires the updates to occur in the same order despite scheduling or initial high-security variables' values.

Even though locks are an essential synchronization primitive [8], controlling synchronization of write/read operations on shared resources by defining a mutual exclusion zone, we are not aware of an information flow analysis directed towards the use of lock primitives. As we will see in Section 2.3, locks can be used to create illegal flows in multiple ways.

2 A Concurrent WHILE Language with Locks

We assume, as usual, a security policy in the form of a lattice of security levels. However, in most of our examples, we shall use only two security levels, low (public, l) and high (secret, h), where l represents a lower confidentiality level than h.

2.1 Runtime Syntax

The runtime syntax of the concurrent WHILE language is as follows:

$e ::= x$	*(variable)*	$s ::= \text{ref}\,(x,\ell) = e \text{ in } s$	*(new pointer)*
$\mid p$	*(pointer)*	$\mid s[p/x]$	*(p is bound to x)*
$\mid n$	*(integer constant)*	$\mid s_1; s_2$	*(sequence)*
$\mid (e_1 \; op \; e_2)$	*(integer operations)*	$\mid e_1 := e_2$	*(write)*
$\mid !e$	*(pointer read)*	$\mid (\text{if } e \text{ then } s_1 \text{ else } s_2)$	*(branch)*
		$\mid (\text{while } e \text{ do } s)$	*(loop)*
		$\mid (\text{spawn } s)$	*(new thread)*
		$\mid (\text{lock } e \text{ in } s)$	*(lock pointer)*
		$\mid (s \backslash p)$	*(pointer locked on s)*
		$\mid \text{skip}$	*(skip)*

All the commands presented in the syntax are used in the source language, except $s[p/x]$ and $(s\backslash p)$ which are only generated on runtime. The construct ref $(x, \ell) = e$ in s, creates a new pointer with a security level of ℓ and initializes it to the result of evaluating e, binding it to the variable x, originating $s[p/x]$. The construct (spawn s) creates a new thread to evaluate s.

Threads may synchronize with each other using locks, with the construct (lock e in s), further explained in Section 2.2.

2.2 Operational Semantics

The essence of information flow is to check whether the information contained in high security pointers could leak to an attacker who can only observe the contents of low security pointers.

The semantics of the language is defined by small-step reductions from state to state, except the semantics of expressions that are big-step since what occurs in each step is not relevant to the analysis being made. A state is a triple (S, L, q) where S is a store mapping pointers to values (pointers or integers), L is a finite set of the locked pointers, and q is a program state, a parallel composition of statements; and can be written as s or $q_1 \parallel q_2$.

The evaluation contexts are defined as follows:

$$\mathbf{E} ::= [\,] \quad \mid \quad E\backslash p \quad \mid \quad E; s \quad \mid \quad E \| q \quad \mid \quad q \| E$$

To define the semantics of nested locks, let us use the set $\lceil \mathbf{E} \rceil$ of pointers held in the context $\mathbf{E}$, analyzed by a "stack inspection" mechanism, like the one shown below:

$$\lceil [\,] \rceil = \varnothing \qquad \lceil \mathbf{E} \rceil = \begin{cases} \{\text{p}\}, \text{ if } \mathbf{E} = E\backslash p \\ \varnothing, \text{ otherwise} \end{cases} \qquad \lceil E[E'] \rceil = \lceil E \rceil \cup \lceil E' \rceil$$

The rules for the crucial case of the operational semantics of locks are presented below. For any of the rules to be applied e must be evaluated as a pointer p, $(S, e) \rightsquigarrow p$. If $p \notin \lceil \mathbf{E} \rceil$, it means that p is already locked by this thread, by analyzing the evaluation context; and the rule [LOCK-S1] applies, ignoring the

lock instruction, proceeding to the evaluation of s. Otherwise, if p is not locked in any other thread, $L \cap \{p\} = \varnothing$, we apply the rule [LOCK-S2]. But if it is locked, the instruction blocks waiting for p to be released by the holder thread. Note that on termination of s, the pointer p is released.

$$\text{[LOCK-S1]} \quad \frac{(S, e) \rightsquigarrow p \quad p \in \lceil \mathbf{E} \rceil}{(S, L, E[(\text{lock } e \text{ in } s)]) \to (S, L, E[s])}$$

$$\text{[LOCK-S2]} \quad \frac{(S, e) \rightsquigarrow p \quad p \notin \lceil \mathbf{E} \rceil \quad L \cap \{p\} = \varnothing}{(S, L, E[(\text{lock } e \text{ in } s)]) \to (S, L \cup \{p\}, E[(s \backslash p)])}$$

2.3 Insecure Program Example

The program in Listing 1.1 uses three variables: y with a low security level, and x and z with a high security level. For simplicity we assume that these variables have binary values, 0 or 1. Notice that after the spawn, the main thread can only pass the while loop when z is different from 0; and for that to happen, the spawned thread needs to acquire the lock on variable x. This program does not satisfy non-interference.

```
1 y_l := 0;
2 z_h := 0;
3 (spawn
4     (lock x_h in z_h := 1; (while !x_h == 0 do skip))
5 );
6 (while !z_h == 0 do skip);
7 (lock x_h in y_l := 1)
```

Listing 1.1. Unsafe program using locks exploiting non-termination

3 Non-interference

Low-equality. Two states are said to be low-equal if they have the same low domain, if they give the same values to all references that are labeled with low security levels, and contain the same set of low locks.

The low part of a memory is defined with respect to a security level "low", and all levels lower.

Bisimulation. Bisimulations provide a natural way of formulating Non-interference in non-deterministic settings. The idea is to express the requirement that two programs are related if they have the same behavior on the low part of two states. Then, in programs that are bisimilar to themselves, the high part of the state does not interfere with the low part, i.e. no security leak occurs.

Definition 1 (Non-interference). *A program q is secure with respect to Non-interference if it satisfies $q \approx_{\Gamma,l} q$ (bisimulation) for all security levels l.*

4 Type System

In this section, we present a type and effect system for ensuring non-interference in our language.

As usual, security levels are assumed to form a lattice $(\mathcal{L}, \top, \bot, \sqcap, \sqcup \leq)$, where $\leq$ means "less confidential than", and $\sqcup$ denotes lowest upper bound.

Expressions will be typed with effect types which provide a security level and distinguish between integers and pointers. *Effect types* have the following syntax:

$$\ell, \kappa ::= h \mid l$$
$$\tau, \theta ::= ref(\tau)^{\ell} \mid int^{\ell}$$

The judgment for expressions is of the form $\Gamma \vdash e : \tau$, where Γ is a type environment mapping variables and pointers to their types, and τ is the type of e. The rules for statements are of the form $\Gamma \vdash s : (\ell, \kappa)cmd$, where cmd shows that the type is command/instruction, where ℓ represents a lower bound on the level of writes, and κ is used to prevent exploits based on termination.

$$\text{[LOCK-T]}\ \frac{\Gamma \vdash e : ref^{\ell_1}, \quad \Gamma \vdash s : (\ell, \kappa)cmd, \quad \ell_1 \leq \ell}{\Gamma \vdash \text{lock } e \text{ in } s : (\ell_1, \kappa \sqcup \ell_1)cmd}$$

5 Soundness

Towards the process of proving soundness of the type system, we have established Subject Reduction.

Subject Reduction states that the type of a program is preserved by reduction. When a program performs a computation step, the reading and writing effects of a program may "weaken" during the execution.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash q : (\ell, \kappa)$ *cmd and* $(S, L, q) \rightarrow (S', L', q')$*, then* $\Gamma \vdash q' : (\ell', \kappa')$ *cmd, with* $\ell \leq \ell'$ *and* $\kappa' \leq \kappa$*.*

Proof. By induction on the inference of $\Gamma \vdash$ q : (ℓ, κ) cmd, and analyzing the resulting ℓ' and κ'. We analyze the multiple cases that a statement may have in the operational semantics, and by their type system rules, we analyze the resulting ℓ' and κ' and verify that $\ell \leq \ell'$ and $\kappa' \leq \kappa$.

Soundness states that the type system only accepts expressions that are secure in the sense of Bisimulation.

Conjecture 1 (Soundness for Non-interference). If for a program q there exists ℓ, κ such that $\Gamma \vdash q : (\ell, \kappa)cmd$, then P satisfies the Non-interference.

6 Conclusion

We have seen that information flow exploits can be expressed via lock based synchronization. This work is still in progress. While some intermediate results that support the correction of the type system have been proven, the soundness proof is still underway. As future work, we plan to develop a tool that implements the type system and helps reject programs that are not secure due to information leaks that are created by means of synchronization primitives.

References

1. Popek, G. J., & Kline, C. S. (1979). Encryption and secure computer networks. ACM Computing Surveys (CSUR), 11(4), 331-356.
2. Downs, D. D., Rub, J. R., Kung, K. C., & Jordan, C. S. (1985, April). Issues in discretionary access control. In 1985 IEEE Symposium on Security and Privacy (pp. 208-208). IEEE.
3. Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. Communications of the ACM, 20(7), 504-513.
4. Bell, D. E., & LaPadula, L. J. (1973). Secure computer systems: Mathematical foundations. MITRE CORP BEDFORD MA.
5. Denning, D. E. (1976). A lattice model of secure information flow. Communications of the ACM, 19(5), 236-243.
6. Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. IEEE Journal on selected areas in communications, 21(1), 5-19.
7. Volpano, D., Irvine, C., & Smith, G. (1996). A sound type system for secure flow analysis. Journal of computer security, 4(2-3), 167-187.
8. Raynal, M. (2012). Concurrent programming: algorithms, principles, and foundations. Springer Science & Business Media.
9. Terauchi, T. (2008, June). A type system for observational determinism. In 2008 21st IEEE Computer Security Foundations Symposium (pp. 287-300). IEEE.