Security of Multithreaded Programms by Compilation

Paper written by Barthe, Rezk, Russo and Sabelfeld [1]

Pascal Wittmann

TU Darmstadt

Seminar “Formal Specification”
December 1–2, 2011
Outline

- Why formal methods?
- Security problems of multithreaded programs.
- Discussion of a solution.
- Other/related solutions.
- Conclusion / Outlook.
Why formal methods?

- Modeling precisely a part of the world
- Formulate the problem unambiguously
- Leaving unimportant things underspecified
- Improve the understanding of the problem
- Use abstraction to cover a large number of cases
Security problems of multithreaded programs

- There are private (high) and public (low) variables
- The attacker can observe low-level variables
- Sequential:
  - explicit flows: lo := hi
  - implicit flows: if hi then lo := 1 else lo := 0
- Concurrent:
  - internal timing leak:
    if hi {sleep(100)}; lo := 1 || sleep(50); lo := 0
  - other example: hi := 0; lo = hi || hi := private-data
- External timing leaks are not covered
- Advantages of formal methods
  - Applicable on a wide rage of schedulers and bytecode
  - Verification without running the program
Discussion of a solution

- Syntax & Semantic of multithreaded programs
  - Program
  - State & Security environment
  - History & Scheduler

- Type system & it’s soundness

- The next function

- Concrete instantiation
  - Transfer rules
  - Defining the next function
We have a set of sequential Instructions $SeqIns$ and a primitive start $pc$ that spawns a new thread.

**Definition (Program $P$)**

1. A set of program points $\mathcal{P}$, with a distinguished entry point 1 and exit point exit.

2. A map from $\mathcal{P}$ to $Ins$, where $Ins = SeqIns \cup \{startpc\}$ and $pc \in \mathcal{P} \setminus \{exit\}$. This map is referred to as $P[i]$.

Further, a relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ that describes possible successor instructions and it’s reflexive and transitive closure $\rightarrow^*$. 
We have a set of local states, LocState and a global memory GMemory. In addition we have a set of thread identifiers Thread.

**Definition (State)**

1. SeqState is a product LocState $\times$ GMemory
2. ConcState is a product (Thread $\rightarrow$ LocState) $\times$ GMemory

Accessors for a state $s$:

- $s.lst$ and $s.gmem$ are projections on the first and second component
- $s.act$ is the set of active threads
- $s.pc(tid)$ retrieves the current program point of the thread tid
Syntax & Semantic of multithreaded programs

Security environment

We assume a set of levels \( \text{Level} = \{\text{low}, \text{high}\} \) where \( \text{low} < \text{high} \) with an attacker on level \( \text{low} \).

**Definition (Security environment)**

1. A function \( se : \mathcal{P} \rightarrow \text{Level} \)
2. A program point \( i \in \mathcal{P} \) is:
   - low if \( se(i) = \text{low} \), written \( L(i) \)
   - high if \( se(i) = \text{high} \), written \( H(i) \)
   - always high if \( \forall j \in \mathcal{P}.(i \rightarrow^* j) \rightarrow se(j) = \text{high} \), written \( AH(i) \)

Now we classify threads in (where \( s \) is a ConcState):

\[
\begin{align*}
    s.\text{lowT} &= \{ tid \in s.\text{act} \mid L(s.\text{pc}(tid))\} \\
    s.\text{highT} &= \{ tid \in s.\text{act} \mid H(s.\text{pc}(tid))\} \\
    s.\text{ahighT} &= \{ tid \in s.\text{act} \mid AH(s.\text{pc}(tid))\} \\
    s.\text{hidT} &= \{ tid \in s.\text{act} \mid H(s.\text{pc}(tid)) \land \neg AH(s.\text{pc}(tid))\}
\end{align*}
\]
History & Scheduler

Definition (History)

A History is a list of pairs \((tid, l)\), where \(tid \in \text{Thread}\) and \(l \in \text{Level}\).

Definition (Scheduler)

A scheduler is a function \(\text{pickt} : \text{ConcState} \times \text{History} \rightarrow \text{Thread}\) that satisfies these conditions:

1. Always picks active threads
2. if \(\text{s.hidT} \neq \emptyset\) then \(\text{pick}(s, h) \in \text{s.hightT}\)
3. Only uses low names and the low part of the history to pick a low thread
LType is a poset (reflexive, antisymmetric, transitiv) of local types.

Intuition of the type judgements: $se, i \vdash s \Rightarrow t$ means if executing program point $i$ the type changes from $s$ to $t$ w.r.t a security environment $se$.

**Definition (Typable program)**

A program is typable (written $se, S \vdash P$) if

1. for all initial program points holds $S(i) = t_{init}$ and
2. $\forall i, j \in P : (i \mapsto j) \rightarrow \exists s \in \text{LType} . se, i \vdash S(i) \Rightarrow s \wedge S(j) \leq s$

where $S : P \rightarrow \text{LType}$ and a security environment $se$. 
Soundness of the type system

Definition (Noninterfering program)

\( g \) is a indistinguishability relation on global memories. A program is noninterfering iff for all global memories \( \mu_1, \mu'_1, \mu_2, \mu'_2 \) the following holds

\[
(\mu_1 \sim g \mu_2 \land P, \mu_1 \downarrow \mu'_1 \land P, \mu_2 \downarrow \mu'_2) \rightarrow \mu'_1 \sim g \mu'_2
\]

Theorem

*If the scheduler is secure and se, S ⊢ P, then P is noninterfering*

Due to this theorem it is possible to typecheck the bytecode (which was compiled type-preserving) to proof the non-existence of internal timing leaks.

The proof is not part of this presentation, but I’ll show the next function on which the proof relies.
The next function

If the execution of program point $i$ results in a high thread, the function $\text{next} : \mathcal{P} \rightarrow \mathcal{P}$ calculates the program point in which the thread becomes visible again. The $\text{next}$ function has to fulfill the following properties:

1. $\text{Dom}(\text{next}) = \{i \in \mathcal{P} \mid H(i) \land \neg AH(i)\}$
2. $i, j \in \text{Dom}(\text{next}) \land i \leftrightarrow j \Rightarrow \text{next}(i) = \text{next}(j)$
3. $i \in \text{Dom}(\text{next}) \land L(j) \land i \leftrightarrow j \Rightarrow \text{next}(i) = j$
4. $j, k \in \text{Dom}(\text{next}) \land L(i) \land i \leftrightarrow j \land i \leftrightarrow k \land j \neq k \Rightarrow \text{next}(j) = \text{next}(k)$
5. $i, j \in \text{Dom}(\text{next}) \land L(k) \land i \leftrightarrow j \land i \leftrightarrow k \land j \neq k \Rightarrow \text{next}(j) = k$
Source and target language

- Simple language with if, ;, :=, while and fork
- Assembly
  - `push n` — push value on the stack
  - `load x` — push value of variable on the stack
  - `store x` — store first element of the stack in x
  - `goto j / ifeq j` — un-/conditional jump to j
  - `start j` — create a new thread starting in j
Transfer rules

\[ \text{LType} = \text{Stack}(\text{Level}) \]

\[
P[i] = \text{store} \times \quad \text{se}(i) \sqcup k \leq \Gamma(x) \\
\text{se}, i \vdash_{seq} k :: st \Rightarrow st
\]

\[
P[i] = \text{ifeq } j \quad \forall j' \in \text{reg}(i), k \leq \text{se}(j') \\
\text{se}, i \vdash_{seq} k :: st \Rightarrow \text{lift}_k(st)
\]

where \( \text{reg} : \mathcal{P} \rightarrow \mathcal{P}(\mathcal{P}) \) computes the control dependence region. \( \text{lift}_k(st) \) is the point-wise extension of \( \lambda k' . k \sqcup k' \). \( \Gamma(x) \) expresses the chosen security policy by assigning a security level to each variable.

Similar rules have to be established for the other commands of the target language.
Concurrent extension

The transfer rules are extended by the following rules:

\[
P[i] \in \text{SeqIns} \quad se, i \vdash_{\text{seq}} s \Rightarrow t
\]

\[
\frac{}{se, i \vdash s \Rightarrow t}
\]

\[
P[i] = \text{start pc} \quad se(i) \leq se(pc)
\]

\[
\frac{}{se, i \vdash s \Rightarrow s}
\]

We label the program points where control flow can branch or side effects can occur.

\[
c ::= [x := e]^n \mid c;c \mid [\text{if } e \text{ then } c \text{ else } c]^n \mid [\text{while } e \text{ do } c]^n
\mid [\text{fork}(c)]^n
\]

With this labeling we can define control dependence regions for the source language (sregion) and derive them for the target language (tregion).
sregion & tregion

**Definition (sregion)**

\( sregion(n) \) is defined as the set of labels that are inside a branching command \([c]^n\) except those inside fork.

**Definition (tregion)**

\( tregion(n) \) is defined for \([c]^n\) as the set of instructions/labels obtained by compiling \([c']^n'\) where \(n' \in sregion(n)\). If \(c\) is while then \(n \in tregion(n)\).

Excerpt of the compilation function \(C\):

\[
C(c) = \text{let } (lc, T) = S(c, []) \text{ in goto } (#T + 2) :: T :: lc :: \text{return}
\]

\[
S(\text{fork}(c), T) = \text{let } (lc, T') = S(c, T) \text{ in (start } (#T' + 2), T' :: lc :: \text{return})
\]
junction points & next function

**Definition (junction point)**

For every branching point $[c]^n$ in the source program we define

$$\text{jun}(n) = \max \{ i | i \in \text{tregion}(n) \} + 1$$

To identify the outermost branching points that involves secrets we extend the type system. A source program is typeable ($\vdash \circ \ c : E$ where $E$ maps labels to security levels) and judgments of the form $\vdash \alpha [c]^n_{\alpha'} : E$. One example typing rule ($\circ$ public, $\bullet$ secret):\[
\begin{array}{c}
\vdash e : H \\
\vdash \bullet \ c : E
\end{array}
\quad
E = \text{lift}_H(E, \text{sregion}(n))
\quad
\vdash \circ \ [\text{while } e \text{ do } c]^n_{\bullet} : E
\]

**Definition (next)**

For all branching program points $c$ such that $\vdash \circ \ [n]^n_{\bullet}$ next is defined as $\forall k \in \text{tregion}(n) . \text{next}(k) = \text{jun}(n)$. 
Other/related solutions

- Protection/hiding based approaches
  - Volpano & Smith [4][5][3] use a protect(c) primitive
  - Russo & Sabelfeld [2] use hide and unhide primitives
- Low-determinism approaches
  - Zdancewic and Myres [6] disallow races on public data
- External-timing based approaches
  - here the attacker is more powerful: he can measure execution time
  - this causes much more restrictiveness (e.g. loops with secret guards are disallowed)
Comparison with Zdancewi and Myres[6]

- Introduces a relative complex language $\lambda^{PAR}_{SEC}$
- Also uses a type system to enforce security
- Uses the same notion of noninterference
- Observational determinism is defined as the indistinguishability of memory access traces

\[
(m \approx_{\zeta} m' \land m \Downarrow T \land m' \Downarrow T') \Rightarrow T \approx_{\zeta} T'
\]

Thus it rejects Programs like \( \text{lo} := 1 || \text{lo} := 0 \)

- In contrast to the paper discussed here, $\lambda^{PAR}_{SEC}$ provides support for synchronization using join patterns
Adaption to the JVM

- JVML’s sequential type system is compatible with bytecode verifikation, thus it’s compatible with the concurrent type system.
- The scheduler is mostly left unspecified, thus introducing a secure scheduler is possible.
- Issues
  - Method calls have a big-step semantic
  - This approach does not deal with synchronization
Conclusion

- Proof of noninterference for a concurrent low-level language
- Proof of type-preserving compilation in context of concurrency
- Scheduler is driven by the security environment
- Independent of the scheduling algorithm
- No useful secure programs are rejected
- No need to trust the compiler, checking can be done at target level (without running the program)
- Programmer does not need to know about internal timing leaks
- No restrictions on dynamic thread creation
- What needs to be done? Extension for real world languages e.g. adding support for synchronization
Bibliography I

Security of multithreaded programs by compilation.  

Securing interaction between threads and the scheduler.  

A sound type system for secure flow analysis.  
Secure information flow in a multi-threaded imperative language.  

Probabilistic noninterference in a concurrent language.  

Observational determinism for concurrent program security.  