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 unambiguous
- 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
  - Tansfer rules
  - Defining the `next` function

Program

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 distinguised entry point 1 and exit point `exit`
2. A map from $\mathcal{P}$ to *Ins*, where $Ins = SeqIns \cup \{startpc\}$ and $pc \in \mathcal{P} \setminus \{\texttt{exit}\}$. This map is refered to as P[i].

Further, a relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$ that describes possible successor instructions and it's reflexive and transitive closure $\mapsto^*$.

State

We have a set of local states, $\texttt{LocState}$ and a global memory $\texttt{GMemory}$. In Addition we have a set of thread identifiers $\texttt{Thread}$.

Definition (State)

1. $\texttt{SeqState}$ is a product $\texttt{LocState} \times \texttt{GMemory}$
2. $\texttt{ConcState}$ is a product $(\texttt{Thread} \rightharpoonup \texttt{LocState}) \times \texttt{GMemory}$

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

Security environment

We assume a set of levels `Level` $= \{low, high\}$ where $low < high$ with an attacker on level *low*.

Definition (Security environment)

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

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

$$
\begin{aligned}
s.lowT &= \{tid \in s.act \mid L(s.pc(tid))\} \\
s.highT &= \{tid \in s.act \mid H(s.pc(tid))\} \\
s.ahighT &= \{tid \in s.act \mid AH(s.pc(tid))\} \\
s.hidT &= \{tid \in s.act \mid H(s.pc(tid)) \land \neg AH(s.pc(tid))\}
\end{aligned}
$$

History & Scheduler

Definition (History)

A History `History` is a list of pairs (tid, l), where tid $\in$ `Thread` and $l \in$ `Level`.

Definition (Scheduler)

A scheduler is a function $pickt : ConcState \times History \rightharpoonup Thread$ that statisfies these conditions:

1. Always picks active threads
2. if s.hidT $\neq \emptyset$ then pick(s, h) $\in$ s.hightT
3. Only uses low names and the low part of the history to pick a low thread

Type system

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 $se, \mathcal{S} \vdash P$) if

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

where $\mathcal{S} : \mathcal{P} \rightarrow \mathtt{LType}$ and a security environment se.

Soundness of the type system

Definition (Noninterfering program)

$\sim_g$ 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 $se, \mathcal{S} \vdash P$, 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 next : $\mathcal{P} \rightharpoonup \mathcal{P}$ calculates the program point in which the thread becomes visible again.
The next function has to fulfill the following properties:

$$Dom(next) = \{i \in \mathcal{P} \mid H(i) \wedge \neg AH(i)\} \quad (1)$$

$$i, j \in Dom(next) \wedge i \mapsto j \Rightarrow next(i) = next(j) \quad (2)$$

$$i \in Dom(next) \wedge L(j) \wedge i \mapsto j \Rightarrow next(i) = j \quad (3)$$

$$j, k \in Dom(next) \wedge L(i) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow next(j) = next(k) \quad (4)$$

$$i, j \in Dom(next) \wedge L(k) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow next(j) = k \quad (5)$$

Source and target language

- Simple langugage 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

$\texttt{LType} = \mathit{Stack}(\texttt{Level})$

$$\frac{P[i] = \mathit{store}\ x \qquad se(i) \sqcup k \leq \Gamma(x)}{se, i \vdash_{seq} k :: st \Rightarrow st}$$

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

where $reg : \mathcal{P} \rightharpoonup \mathfrak{P}(\mathcal{P})$ computes the control dependence region. $\mathit{lift}_k(st)$ 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:

$$\frac{\mathrm{P[i]} \in \mathrm{SeqIns} \qquad se, i \vdash_{seq} s \Rightarrow t}{se, i \vdash s \Rightarrow t}$$

$$\frac{\mathrm{P[i]} = \texttt{start pc} \qquad se(i) \leq se(pc)}{se, i \vdash s \Rightarrow s}$$

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

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

With this labeling we can define control dependence regions for the source langugage (`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 $n' \in \mathit{sregion}(n)$. If c is `while` then $n \in \mathit{tregion}(n)$.

Excerpt of the compilation function C:

```
C(c) = let (lc, T) = S(c, []);
         in goto (#T + 2) :: T :: lc :: return
S(fork(c), T) = let (lc, T') = S(c, T);
         in (start (#T' + 2), T' :: lc :: return)
```

junction points & next function

Definition (junction point)

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

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

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):

$$\frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad E = lift_H(E, sregion(n))}{\vdash_\circ [while\ e\ do\ c]^n_\bullet : E}$$

Definition (next)

For alle branching program points c such that $\vdash_\circ [n]^n_\bullet$ *next* is defined as $\forall k \in tregion(n)\ .\ next(k) = jun(n)$.

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 λ_{SEC}^{PAR}
- 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

$$(m \approx_\zeta m' \wedge m \Downarrow T \wedge m' \Downarrow T') \Rightarrow T \approx_\zeta T'$$

  Thus it rejects Programs like `lo := 1` || `lo := 0`
- In contrast to the paper discussed here, λ_{SEC}^{PAR} 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

[1] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld.
Security of multithreaded programs by compilation.
In *In Proc. 12th European Symposium on Research in Computer Security*, pages 2–18. Springer-Verlag, 2007.

[2] Alejandro Russo and Andrei Sabelfeld.
Securing interaction between threads and the scheduler.
In *IEEE Computer Security Foundations Symposium*, pages 177–189, 2006.

[3] G. Smith and D. Volpano.
A sound type system for secure flow analysis.
In *J. Computer Security 4*, pages 167–187, 1996.

Bibliography II

[4] G. Smith and D. Volpano.
Secure information flow in a multi-threaded imperative language.
In *ACM Symp. on Principles of Programming Languages*, pages 355–364, 1998.

[5] G. Smith and D. Volpano.
Probalistic noninterference in a concurrent language.
In *J. Computer Security 7*, pages 231–253, 1999.

[6] Steve Zdancewic and Andrew C. Myers.
Observational determinism for concurrent program security.
In *In Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.