Seminar: Formal Specification

Security of Multithreaded Programs by Compilation

Pascal Wittmann, Advisor: Artem Starostin

TU Darmstadt

1 Motivation

Protecting the confidentiality of information that is processed by a modern computer is challenging and important, since more and more sensitive information is fed to them. Access control and encryption are not enough to ensure this confidentiality, because the usage of the information after the access or decryption is not restricted. It is necessary to control the *information flow* to protect the sensitive information. On modern mobile devices sensitive information is processed and programs are executed multithreaded (e.g. to prevent lock ups when establishing a network connection). Through the timing difference of the scheduled threads it is likely that sensitive information is leaked and an attacker may obtain sensitive data.

The idea Barthe, Rezk, Russo and Sabelfeld phrased in [BRRS07] was to close this leak (formally called *covert channel*) by annotating the byte-code of the program[1] with security labels according to the confidentiality of the information, which causes the scheduler to hide threads that leak information from the attacker. Their formal model allows to formulate this idea as a framework, which can be instantiated with many programming languages, schedulers and types of byte-code.

2 Introduction to the research area

The challenge to protect the confidentiality of information processed by a computer lead thirty years ago (c.f. [Zda04]) to the research on the security of information-flows, because of the shortcomings of the standard methods. These standard methods are: Access control and encryption. Where access control ensures that the data is only accessed by an authorized entity, and encryption ensures that data can be securely transmitted over an insecure channel. But all these methods cover the release of information and not the further propagation (i.e. how the information is used after having access to it or decrypting it). This is the place where information-flow security comes into play.

[1] Most operating systems for mobile devices use some sort of byte-code (e.g. Android) for application software.

The idea is to *"track and regulate"*[Zda04] where information flows to prevent the leak of secret information. The ways on which the information flows through the system are called *channels*. Channels that are not intended to transport information are called *covered channels*. These covered channels can be classified into the following categories [SM03] (in the following examples h is a secret variable and l a variable that is public):

- *Explicit and implicit flows*: In an explicit flow the information is leaked directly into some public variable (e.g. $l := h$), while in implicit flows the information is leaked through the control structure of the program (e.g. if $h = 1$ then $l := 1$ else $l := 0$).
- *Probabilistic channels*: If an attacker is able to run a computation multiple times, he might be able to obtain information by looking at the probability distribution of the public outputs.
- *Power channels*: If the attacker has physical access to the computer or at least can figure out the power consumption, he might be able to obtain data through the changing power consumption.
- *Resource exhausting channels*: Information also may be leaked through the exhaustion of finite (mostly physical) resources (e.g. a buffer overflow).
- *Termination channels*: The attacker can obtain information through the termination or non-termination of the computation or program.
- *Timing channels:* Information can be obtained through the time at which this action occurs. On the one hand *external* timing channels cover the actions like the termination of the program, where the attacker obtains information from the total execution time of the program. On the other hand *internal* timing channels occur in multithreaded programs through the timing difference between threads. For example the two threaded (denoted with ||) program

  $$\text{if } h = 1 \ \{\text{sleep}(100)\} \ ; \ l := 1 \ || \ \text{sleep}(50) \ ; \ l := 0$$

  executed with most schedulers will leak the information of h into l through the timing difference between $h = 1$ and $h = 0$.

The notion of confidentiality that is used in most work in the area of information-flow security is *noninterference*. This policy is defined in various ways but the essence is, that it requires *"that secret information [do] not affect [the] publicly observable behavior of a system"*[Zda04][2]. For most real world applications this policy is far too strict, because it forbids useful programs like password checkers.[3] Approaches that allow controlled release of information are called declassification. The kinds of declassification are *what* information is released, *who* released this information, *where* in the system the information is released and *when* the information is released (c.f. [SS05]). For these approaches it is necessary that an active attacker can only know as much as a passive attacker.

[2] Even if the attacker has full access to the source code of the program.
[3] A password checker needs to reveal whether the user input was correct or not.

Mechanisms for controlling the information flow can be implemented either dynamic or static. Dynamic approaches[4] label the information with security labels and propagate these labels wherever the information is used. Static approaches[5] analyze the program code and are therefor far more promising, because with these approaches it is possible to check all evaluation paths.

The paper resumed in the following concentrates on internal timing channels with a noninterference policy.

3 Summary of the article

In this summary I will follow mainly the structure of the original paper [BRRS07] which is as follows. First the basic terms and definitions for multithreaded programs and the scheduler are laid. After that the notion of security we want to achieve is presented. Along with this a skeleton of a type system is described, which ensures that a program typeable in this type system is secure. The proof that this holds is sketched in the following section. In the last section the abstract framework is instantiated with a concrete example.

3.1 Syntax and Semantics of multithreaded programs

A program is viewed as an abstract thing, which consists of a set of program points $\mathcal{P}$ with a distinguished entry (1) and exit point (`exit`) and a function P that maps program points to instructions.

These instructions are not further specified, but contain an instruction to create a new thread (`start` *`pc`* where pc is the start instruction of the new thread). The instructions without `start` are called $SeqIns$.

Further, there is a relation $\mapsto$ that describes possible successor instructions. `exit` is the only program point with no successor and `start` *`pc`* may only have a single successor (the following program point).

The next thing introduced are the security levels. We assume the attacker "is" a level k. From this assumption we can reduce every set of security levels w.l.o.g into Level $= \{low, high\}$, where $low < high$, by mapping elements that are no more sensitive than k to low and all other elements – including incomparable ones – to $high$. It is also assumed, that access control works correctly (i.e. the attacker can not access $high$ elements directly).

To connect programs and security levels, a *security environment* (se) is defined, which is used to prevent flows over implicit channels. A security environment is a function that maps program points to security levels. A program point i is called high if $se(i) = high$, low if $se(i) = low$ and always high if all points j reachable (according to $\mapsto$) from i satisfy $se(j) = high$ and i is a high program point.

[4] The taint mode of the programming language Perl uses this mechanism.
[5] This mechanism is implemented in Jif for Java and Flow Caml for Caml.

Now we come to the semantics part. The main idea is to build the semantics for multithreaded programs by combining the semantics for sequential programs with a scheduler.

All active[6] threads are collected in a set $Thread$. The state of the concurrent running threads ($ConcState$) is defined as the product of the partial function space ($Thread \rightharpoonup LocState$) and the set of global memories $GMemory$. Where $LocState$ is the internal memory of a thread (from there no information can leak, because everything is private/internal) and the global memory $GMemory$ which is the critical part of the system, because it is a memory shared between all active threads.

At this point a first simplification can be made. Looking a state $s \in$ ConcState we can first extract the active threads (`s.act`) by taking the domain of the first component. According to a security environment we can classify these threads, with respect to their current program point (`s.pc(tid)` where $tid \in$ Thread), into *low* threads if the current program point is low. In *high* threads if the current program point is high. In *always high* threads if the current program point is always high and in *hidden* threads, if the current program point is high but not always high.

The last two are the interesting ones. If a thread is *always high* it can not leak any information into low, because it never gets in touch with low program points. So these threads can safely be interleaved between all other threads by the scheduler.

The *hidden* threads are the ones we have to care about. These contain obviously high information in the current program point, but have subseqent instructions that deal with low information. Since the attacker can watch the low part of the memory, chances are good that he can deduce high information through looking at changing low outputs. To prevent indirect flows that are introduced by these hidden threads, the scheduler will be modified to treat these threads in a special way. This will be done by "hiding" these threads, therefore comes the name of them.

To complete our multithreaded setup we need a scheduler. The scheduler will operate on histories. A history is a list of pairs (tid, l), where $tid \in$ Thread and $l \in$ Level. In this history all threads chosen by the scheduler are recorded.

At this point no concrete scheduler is defined to make the framework applicable for a wide class of schedulers. A scheduler is in this class if it can be modeled as a function $pickt$: ConcState $\times$ History $\rightharpoonup$ Thread which satisfies the following constraints:

1. It always picks an active thread
2. If there is a hidden thread, always choose high or always high threads[7]
3. Only use low information and the low part of the history to choose a new thread

[6] A thread is active from `start` *`pc`* until it reaches the `exit` point.
[7] With this constraint the interleaving of always high threads is realized.

To define what noninterference means we need a notion of execution. It is assumed that we have a sequential execution relation $\rightsquigarrow_{seq} \subseteq \text{SeqState} \times \text{SeqState}$. One step execution for the multithreaded language $\rightsquigarrow_{conc} \subseteq (\text{ConcState} \times \text{History}) \times (\text{ConcState} \times \text{History})$ is defined by the rules in figure 1.

$$\frac{pick(s,h) = ctid \qquad s.pc(ctid) = i \qquad P[i] \in \text{SeqIns} \qquad \langle s(ctid), s.gmem\rangle \rightsquigarrow_{seq} \sigma, \mu \qquad \sigma.pc \neq \texttt{exit}}{s, h \rightsquigarrow_{conc} s[lst(ctid) := \sigma, gmem := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{pick(s,h) = ctid \qquad s.pc(ctid) = i \qquad P[i] \in \text{SeqIns} \qquad \langle s(ctid), s.gmem\rangle \rightsquigarrow_{seq} \sigma, \mu \qquad \sigma.pc = \texttt{exit}}{s, h \rightsquigarrow_{conc} s[lst := lst \qquad ctid, gmem := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{pick(s,h) = ctid \qquad s.pc(ctid) = i \qquad P[i] = \texttt{start}\ pc \qquad fresht_{se(i)}(s) = ntid \qquad s(ctid).[pc := i+1] = \sigma'}{s, h \rightsquigarrow_{conc} s.[lst(ctid) := \sigma', lst(ntid) := \lambda_{init}(pc)], \langle ctid, se(i)\rangle :: h}$$

Fig. 1. Multithreaded execution.

The scheduler is allowed to pick a new thread after every transition of $\rightsquigarrow_{seq}$. The intuition of the first rule is, that the scheduler picks a new thread *coed* which is at program point i. This program point maps to an sequential instruction. The execution of this instruction leads to a local state σ and a global memory μ. If this is the case and the instruction i was not the last instruction (i.e. $\sigma.pc \neq \texttt{exit}$), then the concurrent transition can be made. In this transition the concurrent state is updated according to the results of the sequential execution and the thread identifier, furthermore security level of i is recorded in the history.

The second rule covers the case in which the thread picked by the scheduler has only the current instruction before terminating (i.e. $\sigma.pc = \texttt{exit}$). In the concurrent transition the current thread identifier is removed from the concurrent state and everything else is like in the first rule.

The third rule introduces the dynamic creation of new threads. In this rule the function $fresh_l$ [8] takes a set of thread identifiers and returns a new thread identifier at level l and $\lambda_{init} : \mathcal{P} \to \text{LocState}$ produces an initial state with the program pointer at the given program point. If the current instruction at program point i is the `start` instruction, a new thread identifier is picked w.r.t. the security level of i, then program counter of the current thread is increased by one, to step to the next instruction. The resulting concurrent state includes the updated state of the current thread and the initial state for the new thread.

[8] It is assumed that $fresh_l(tid) \neq fresh_{l'}(tid) \Leftrightarrow l \neq l'$.

The global memory is not modified and the history is extended with the current thread identifier and the security level of i.

Based on this an evaluation relation $\Downarrow_{conc} \subseteq (\text{ConcState} \times \text{History}) \times \text{GMemory})$ is defined by

$$s, h \Downarrow_{conc} \mu \Leftrightarrow \exists s', h' : \ (s, h \rightsquigarrow^*_{conc} s', h') \wedge s'.act = \emptyset \wedge s'.gmem = \mu$$

The intuition is that the state s evaluates according to a history h to a final global memory μ iff there is a sequence of concurrent executions that terminates (i.e. there are no active threads left) and has the global memory μ. $\rightsquigarrow^*_{conc}$ is the reflexive and transitive closure of $\rightsquigarrow_{conc}$.

$P, \mu \Downarrow_{conc} \mu'$ is a shorthand for $\langle\langle main, \lambda_{init}(1)\rangle, \mu\rangle, \epsilon^{hist} \Downarrow_{conc} \mu'$, where $main$ is the identity of the main thread and ϵ^{hist} the empty history.

Now we can define our goal: Noninterference. We define noninterference in accordance to an indistinguishability relation $\sim_g$ on global memories. Barthe et. al. state that it is not necessary – for the purpose of the paper – to specify the definition of this relation. But to get a feeling for this relation, one can define it as: $\mu \sim_g \mu' \Leftrightarrow \mu|_{low} = \mu'|_{low}$ where $\mu|_{low}$ projects out all high elements. Based on that a program P is non-interfering if for all global memories μ_1, μ_2, μ'_1 and μ'_2 it holds that:

$$\mu_1 \sim_g \mu_2 \wedge (P, \mu_1 \Downarrow_{conc} \mu'_1) \wedge (\mathrm{P}, \mu_2 \Downarrow_{conc} \mu'_2) \Rightarrow \mu'_1 \sim_g \mu'_2$$

3.2 Type system

The type system is the core part of the framework in the sense that it enforces the previously defined noninterference property. Thus every program which is typeable, is non-interfering.

The type system for multithreaded programs is build up from a type system for sequential programs for which the following assumptions hold:

1. We have a partial ordered set $(\text{LType}, \leq)$ of local types, with an initial type T_{init}
2. and typing judgments of the form $se, i \vdash_{seq} \mathrm{S} \Rightarrow \mathrm{T}$, where $\mathrm{S}, \mathrm{T} \in \text{LType}$, $i \in \mathcal{P}$ and se is a securirty environment.

The intuition of the typing judgment is, that if we execute the instruction at program point i w.r.t. the security environment se and the current type is S, then the type after the execution is T.

This type system is extended by the rules in figure 2 to support multithreading.

The first rule states, that sequential commands are treated as usual and the second rule ensures, that the security level of the entry point of the spawned thread, is lower bounded by the level of the `start` instruction.

A program is typeable in this type system (written $\mathcal{S}, se \vdash P$), where $\mathcal{S}$ is a function $\mathcal{S} : \mathcal{P} \rightarrow \text{LType}$ that maps a local type to every program point and se a

$$\frac{P[i] \in \text{SeqIns} \qquad se, i \vdash_{seq} \text{S} \Rightarrow \text{T}}{se, i \vdash \text{S} \Rightarrow \text{T}} \qquad \frac{P[i] = \texttt{start}\ pc \qquad se(i) \leq se(pc)}{se, i \vdash \text{S} \Rightarrow \text{S}}$$

Fig. 2. Extension of the sequential typing rules.

security environment, iff $\mathcal{S}$ maps every initial program point[9] to T_{init} and that for every program point j, which is an successor of i, there is a type $s \in \text{LType}$, such that S is lower bounded by $\mathcal{S}(j)$ and $se, i \vdash \mathcal{S}(i) \Rightarrow \text{S}$ holds.

3.3 Soundness

The framework is now complete, but the proof of the connection between the type system and the noninterference property is still outstanding. The full proof is not part of the paper and I will only sketch the most important parts.

The goal is to proof the following theorem:

Theorem 1. *If the scheduler is secure and $se, \mathcal{S} \vdash P$, then P is non-interfering.*

The scheduler is secure, if it is defined w.r.t. the conditions from section 3.1.

An important hypothesis to succeed in the proof of this theorem is the existence of a `next` function. This `next` function should compute for every high program point the first subsequent program point with a low security level. With this function one is able to detect when a hidden thread is allowed to become visible again.

This intuition is capture in the following properties for the function $next : \mathcal{P} \rightharpoonup \mathcal{P}$:

NePd $Dom(next) = \{i \in \mathcal{P} | se(i) = high \wedge \exists j \in \mathcal{P}. i \mapsto^* j \wedge se(j) \neq high\}$
NeP1 $i, j \in Dom(next) \wedge i \mapsto j \Rightarrow next(i) = next(j)$
NeP2 $i \in Dom(next) \wedge j \notin Dom(next) \wedge i \mapsto j \Rightarrow next(i) = j$
NeP3 $j, k \in Dom(next) \wedge i \notin Dom(next) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow next(j) = next(k)$
NeP4 $i, j \in Dom(next) \wedge k \notin Dom(next) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow next(j) = k$

where $\mapsto^*$ is the reflexive and transitive closure of $\mapsto$.

The domain of next (i.e. NePd) captures the high, but not always high program points (i.e. the ones that can result in a hidden thread). The property NeP1, states that two directly successive high program points have the same program point in which the thread becomes visible again. The property NeP2 is the counterpart of the NeP1: If a low program point follows a high program point, this low program point is the result of the next function for the high program point. NeP3 and NeP4 denote that the next of an outermost if respective while instruction is at least after the control dependence region.

[9] Including the ones of spawned threads.

3.4 Instantiation

To demonstrate how the framework can be used, it was instantiated with a simple assembly language, which is given by the following grammar:

$instr$ ::= binop op	binary operation with values from stack
| 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

where $op \in \{+, -, \times, /\}$, $n \in \mathbb{Z}$ and x are variables. The operational semantics are standard and not explicitly necessary for the following instantiation, therefor they are omitted.

The local states are modeled as a pair of the operand stack and the program counter. The initial state $\lambda_{init}(pc)$ has an empty operand stack ϵ and points to the given initial program point.

Besides this concrete language we need to define a type system to enforce noninterference according to section 3.2. The local types are defined by a stack of security levels LType = Stack(Level) and τ_{init} with the empty stack. The typing rules defined in figure 2 are extended with rules for the concrete instructions of the assembly language in figure 3. In this summary I only explain two of them, the rest follows in a similar manner.

$$\frac{P[i] = \text{store } x \qquad se(i) \sqcup k \leq \Gamma(x)}{se, i \vdash_{seq} k :: st \Rightarrow st} \qquad \frac{P[i] = \text{ifeq } j \qquad \forall j' \in reg(i), k \leq se(j')}{se, i \vdash_{seq} k :: st \Rightarrow lift_k(st)}$$

Fig. 3. Excerpt of typing rules for the assembly language.

The security levels in $LType$ are the security levels of the operands in the local state. To express our security policy, we declare a function $\Gamma(x)$ that assigns to every variable x a security level. If we now want to `store` the top of the stack into the variable x, one premise is that the security level of the program point $se(i)$ *and* the security level of the value on top of the operand stack is *lower or equal* to the security level that was assigned by Γ to the variable x. If this is the case the head of $LType$ is removed. The $\sqcup$ operator is like a logical "or", it returns the higher of the given security values.

The rule for the branching instruction can only be used if the security level of all program points in the control dependence region are lower bounded by the security level of the value that is used as the condition (i.e. the top of the operand

stack). The function $lift_k$: Level $\rightarrow$ Stack(Level) $\rightarrow$ Stack(Level) extends $\sqcup$ to stacks of security levels. This ensures, that every program point in the control dependence region has the same security level as the branching point.

The next step is to construct the `next` function. To make this task easy, we introduce a source language which will be complied into the given assembly language. The source language is defined as follows:

$$e ::= n \mid x \mid e \; op \; e \quad c ::= x := e \mid c;c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid \text{fork}(c)$$

Now it is easy to define the control dependence region and the junction points for the source language and deduce the ones for the assembly language from them. The first step is to label the source language at the regions at risk with natural numbers:

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

According to this labels we can define the control dependence regions for the source language:

Definition 1. *The control dependence region sregion(n) for a branching command $[c]^n$ in the source language is defined as the labels inside the branching command, except for those ones that are inside a* `fork` *command.*

Now we can define *tregion* according to a compilation function $\mathcal{C}$. I won't define the compilation function here, because it is not essential how it exactly looks like.

Definition 2. *tregion(n) is defined as the set of instruction labels obtained by compiling the commands $[c']^{n'}$ of the branching instruction $[c]^n$ of the source code with a compilation function $\mathcal{C}$.*

With this definition of *tregion* in mind, it is easy to define the junction points.

Definition 3. *The junction points are computed by a function $jun : \mathcal{P} \rightarrow \mathcal{P}$. This function is defined on all junction points $[c]^n$ in the source program as $jun(n) = max\{i \mid i \in tregion(n)\} + 1$.*

Intuitively a junction point according to this definition is the point that follows on the last instruction that is affected by the branching instruction.

This looks familiar to the `next` function we want to define.[10] The part that is missing, is the restriction to outermost branching points whose guards involves secrets.

To make this distinction a new type system is introduced. Type judgments have the form $\vdash_\alpha [c]^n_{\alpha'} : E$, where E is a function that maps labels to security

[10] We could define `next` for every instruction i inside an outermost branching point $[c]^n$ as $next(i) = jun(n)$.

levels.[11] α denotes if c is *part of* a branching instruction that branches on secret ($\bullet$) or public ($\circ$) data and α' is the security level of the guard *in* the branching instruction.

We now take a look at the rules of this type system concerning the `if` instruction, which are defined in figure 4.

$$\text{L-Cond}\quad \frac{\vdash e : L \qquad \vdash_\alpha c : E \qquad \vdash_\alpha c' : E}{\vdash_\alpha [\text{if } e \text{ then } c \text{ else } c']^n_\alpha : E} \qquad \text{H-Cond}\quad \frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E}{\vdash_\bullet [\text{if } e \text{ then } c \text{ else } c']^n_\bullet : E}$$

$$\text{Top-H-Cond}\quad \frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E \qquad E = lift_H(E, sregion(n))}{\vdash_\circ [\text{if } e \text{ then } c \text{ else } c']^n_\bullet : E}$$

Fig. 4. Typing rules for `if` on source level.

The first rule L-Cond from Figure 4 says that if we branch on a low guard, then everything depends on the security level of c and c'. The rule H-Cond covers the case where we have a high guard, in this case the control dependence region has to be marked high. The rule Top-H-Cond is the interesting one. Because of the preceding rules we can not be part of branch with a low guard, therefore we are in the outermost high branch. The premise $E' = lift_H(E, sregion(n))$ reads as: For all labels in the control dependence region E is defined as $E'(n) = H \sqcup E(n)$ and $E'(n) = E(n)$ for all other labels.

This type system is powerful enough to prevent explicit and implicit flows and can therefor replace the type system defined previously.

With this type system in mind, we can now define our `next` function.

Definition 4. *For every branching point c in the source program such that $\vdash_\circ [c]^n_\bullet$, the next function is defined as $\forall k \in tregion(n).\ next(k) = jun(n)$.*

The proof that this definition fulfills the properties from section 3.3 and all other proofs can be found in [BRRS09].

Now the instantiation of the framework is complete, except for the scheduler which was left unspecified in the paper.

[11] Given E it is easy to define a security environment se. For a definition of E see [BNR06].

4 Comparison with other approaches/further work

In this section I will first give a short overview of two other approaches and then compare them with the approach presented before and draw a conclusion.

4.1 Observational Determinism for Concurrent Program Security[ZM03]

Zdancewic and Myers approach has the goal to provide a secure concurrent language that has a general and realistic support for concurrency and whose security can be checked statically.

They introduced the language λ_{SEC}^{PAR} which supports higher-order functions, an unbounded number of threads, synchronization (via join patterns), message passing and shared memory. Regardless of those realistic features λ_{SEC}^{PAR} is not intended to server as a user-level programming language because its syntax and type system is too awkward. Instead it is only used a model for studying information flow.

Low-security observational determinism is used to enforce noninterference. The idea is to make the naturally nondeterministic system, deterministic from the point of view of the attacker.

This deterministic noninterference is defined as

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

where T and T' are traces of the execution of a program. m and m' are initial configurations. Let $T(L) = [M_0(L), M_1(L), \dots]$ be the list of values at location L in the trace T. $T \approx_\zeta T'$ holds if the values of $T(L)$ and $T'(L)$ are pairwise indistinguishable. Since it is allowed that one sequence is a prefix of the other, there is no need to proof termination, but external timing attacks are possible.

This notion of observational determinism is captured in a type system for λ_{SEC}^{PAR}, thus the security of programs can be checked statically.

4.2 Flexible Scheduler-Independent Security[MS10]

This approach developed by Mantel and Sudbrock has the goal to be on the one hand scheduler independent and on the other not too restrictive.

To reach this goal they model the scheduling explicit but general enough to apply to a wide class of common schedulers. So rather than talking about program states, they introduce system configurations which contains a list of threads (thread pool) the global memory and the scheduler state. The scheduler is able to store information in its state and can retrieve input (e.g. the number of active threads). Based on this information the scheduler makes decisions. To cover nondeterministic schedulers (e.g. the uniform scheduler) the decision making of the scheduler and the execution of the program is probabilistic.

Based on this scheduler model, a schedule-specific security property $\mathcal{S}$ is defined on thread pools. Intuitively a thread pool is $\mathcal{S}$-secure if the probability of running a program with system configuration $conf$ under the scheduler $\mathcal{S}$ resulting in a global memory m is the same as running it with configuration $conf'$ resulting in m', and satisfying that $m =_L m'$.

The novel security property introduced in this paper is defined without mentioning the scheduler and equal to $\mathcal{S}$-security for a *robust* scheduler $\mathcal{S}$. Therefor this property is called *flexible scheduler-independent security* (FSI-security). A thread pool is FSI-secure if (starting from an low-equal memory configuration) the resulting memories are always low-equal when executing commands that potentially modify low-variables and if spawned threads are also FSI-secure.

Those robust schedulers are those, for which *"the scheduling of low threads during a run of a FSI-secure thread pool remains unchanged when one removes all high threads from the thread pool"*[MS10].

If a program is FSI-secure or not is checked statically with a security type system, which is introduced for a sample while-language.

4.3 Conclusion

The approach of Barthe et. al. shows how to build a framework to create type-annotated programs that can be check statically and can be run independent of the scheduling algorithm. Thus there is no need to trust the compiler.

In contrast to Mantel and Sudbrocks work the whole framework depends heavily on the security of the scheduler. It is possible to choose most scheduling algorithms, but every algorithm needs to be modified satisfy the properties for a secure scheduler. Additionally its likely that the interface of the scheduler needs to be changed/extended in nearly ever implementation. The FSI-security instead is applicable to every robust scheduler without changing the algorithm or the interface.

The approach also (currently) lacks support for real world languages. For example support of synchronization would be necessary to implement the approach for Java. The approach of Zdancewic and Myers supports those features, but is much more restrictive. Due to the use of observational determinism obviously secure programs like `l := 1 || l := 0` are rejected. The approach of Barthe et. al. does not reject those programs and is in that sense much more permissive.

In the end Barthe et. al. presented a framework with reaches many goals (like permissiveness, statically check-able and requires no intervention from the programmer) but needs to modify the scheduler interface, which makes it hard to use for real programs. In addition they did not showed how to implement the type-annotations in a real language.

References

BNR06. Gilles Barthe, David Naumann, and Tamara Rezk. Deriving an information flow checker and certifying compiler for java. In *In 27th IEEE Symposium on Security and Privacy*, pages 230–242. IEEE Press, 2006.

BRRS07. 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.

BRRS09. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. `http://www.cse.chalmers.se/~russo/tissecfull.pdf`, 2009.

MS10. Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133. Springer, 2010.

SM03. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

SS05. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. In *In Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFW05)*, pages 255–269, 2005.

Zda04. Steve Zdancewic. Challenges for information-flow security. In *In Proc. Programming Language Interference and Dependence (PLID)*, 2004.

ZM03. 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.