Combining Different Proof Techniques for Verifying Information Flow Security

Heiko Mantel, Henning Sudbrock, and Tina Kraußer

Security Engineering Group, RWTH Aachen University, Germany
{mantel,sudbrock,krausser}@cs.rwth-aachen.de

Abstract. When giving a program access to secret information, one must ensure that the program does not leak the secrets to untrusted sinks. For reducing the complexity of such an information flow analysis, one can employ compositional proof techniques. In this article, we present a new approach to analyzing information flow security in a compositional manner. Instead of committing to a proof technique at the beginning of a verification, this choice is made during verification with the option of flexibly migrating to another proof technique. Our approach also increases the precision of compositional reasoning in comparison to the traditional approach. We illustrate the advantages in two exemplary security analyses, on the semantic level and on the syntactic level.

1 Introduction

Information flow security aims at answering the question: Is a given system sufficiently trustworthy to access secret information? The two main research problems are, firstly, finding adequate, formal characterizations of trustworthiness and, secondly, developing sound and efficient verification techniques based on these characterizations. Information flow security has been a focal research topic in computer security for more than 30 years. Nevertheless, the problem to secure the flow of information in systems is far from being solved. In [28], the state of the art was surveyed for approaches to capturing and analyzing information flow security of concrete programs. For information flow security at the level of more abstract specifications, a broad spectrum of approaches has been developed (see, e.g., [12, 19, 20, 11, 26, 17, 5]). The most popular verification techniques are the unwinding technique on the level of specifications (see, e.g., [13, 24, 18, 4]), and security type systems and program logics on the level of programs (see [28] for a good overview). In this article, we focus on a multi-threaded programming language.

We use the standard scenario for investigating information flow security of imperative programs. That is, the initial values of some variables, the so called *high variables*, constitute the secrets that must be protected while the remaining variables, the *low variables*, initially store public data. We assume an attacker ζ

Appeared in G.Puebla (Ed.): LOPSTR 2006, LNCS 4407, pp. 94–110, 2007
© Springer-Verlag Berlin Heidelberg 2007

who can observe the values of low variables before and at the end of a program run. The security requirement is that no information flows from the high variables into low variables during program execution. We use l to denote low variables and h to denote high variables, i.e. variables that may store secrets.

There are various possibilities for how a program could accidentally or maliciously leak secrets. It could copy a secret into a low variable as, e.g., in $P_1 = l{:=}h$. Such leaks are referred to as *intra-command leaks* or *explicit leaks* [9]. More subtly, a secret could influence the flow of control, leading to different assignments to low variables as, e.g., in $P_2 = \mathsf{if}\ h = 0\ \mathsf{then}\ l{:=}0\ \mathsf{else}\ l{:=}1\ \mathsf{fi}$. If the value of l is 0 at the end of the run, $h = 0$ must have held initially, and if l is 1 then $h \neq 0$ held. Such information leaks are referred to as *inter-command leaks* or *implicit leaks*. Even more subtle leaks originate in a multi-threaded setting.

Verification techniques are often based on characterizations of information flow security that are compositional with respect to the primitives of the programming language. Two well known observations motivated our work:

- Compositionality is indeed helpful, both for making verification techniques efficient and for simplifying the derivation of results at the meta level, e.g., for proving a soundness theorem for a syntactic, type-based analysis.
- Compositionality leads to overly restrictive characterizations of security. Simple programs that are typically rejected include, e.g., $\mathsf{while}\ h \leq 10\ \mathsf{do}\ h := h + 1\ \mathsf{od}$, $l{:=}h; l{:=}0$, $h{:=}0; l{:=}h$, and $\mathsf{if}\ h = 0\ \mathsf{then}\ l := 0\ \mathsf{else}\ l := 0\ \mathsf{fi}$ (for instance, the security type systems in [32, 29] reject all these programs).

More recent work aimed at relaxing security definitions and type systems such that intuitively secure programs like the above examples are not rejected anymore by a security analysis. For instance, [30] and [25] provide solutions for, e.g, $\mathsf{while}\ h \leq 10\ \mathsf{do}\ h := h + 1\ \mathsf{od}$ (possibly requiring the addition of auxiliary commands to the program), and [15] offers a solution for, e.g., $\mathsf{if}\ h = 0\ \mathsf{then}\ l := 0\ \mathsf{else}\ l := 0\ \mathsf{fi}$. While this progress is promising, the approach taken requires the incremental improvement of each individual analysis technique. In this article, we present an alternative approach. We show that and how different analysis techniques can be combined, effectively developing a higher-level security calculus that can be extended with existing verification techniques as plugins. This approach applies to the semantic level, where one applies (semantic) characterizations of security that enjoy desirable meta properties (such as, e.g., compositionality) and uses a calculus for some general-purpose logic for verification. The approach also applies to the syntactic level, where one uses specific security calculi (such as, e.g., security type systems) for verification. Instead of eliminating weaknesses of each individual verification technique, our approach aims at combining the strengths of available techniques.

In summary, the contributions of this article are, firstly, a novel approach to verifying information flow security and, secondly, the illustration of how different verification techniques can be beneficially combined in the information flow analysis of a fairly realistic example program. The article constitutes an initial step in the proposed direction, and some issues such as finding a fully satisfactory baseline characterization will need further investigation.

2 Information Flow Security in an Imperative Language

To make our approach concrete, we introduce a simple, multi-threaded programming language that includes assignments, conditionals, loops, a command for dynamic thread creation, and a sync command. Without sync command and arrays, this language is also used, e.g., in [29]. The set Com of commands is defined by (where V is a command vector in $\boldsymbol{Com} = \bigcup_{n\in\mathbb{N}} Com^n$)

$$C ::= \mathsf{skip} \mid Id{:=}Exp \mid Arr[Exp_1]{:=}Exp_2 \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \mid \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od} \mid \mathsf{fork}(CV) \mid \mathsf{sync}.$$

We restrict program variables to Booleans, integers, and arrays. The length of an array Arr is denoted by $Arr.length$ and is treated like a constant. The ith element of Arr is denoted by $Arr[i]$ and treated like a variable. *Expressions* are program variables, constants, and terms resulting from applying operators to expressions: $Exp ::= Const \mid Var \mid Arr[Exp] \mid Arr.length \mid op(Exp_1, \ldots, Exp_n)$.

A *state* is a mapping from variables in a given set Var to values in a given set Val. The set of states is denoted by S. We use $[v = n]s$ to denote the state that maps v to n and all other variables to the same values like the state s. We treat arrays like in [8]: If an array access $a[i]$ is out of bounds (i.e. $i < 0$ or $i \geq a.length$) then a dummy value is returned (0 for integers and False for Booleans), no exception is raised and no buffer overflow occurs. We use the judgment $\langle Exp, s\rangle \downarrow n$ for specifying that expression Exp evaluates to value n in state s. Expression evaluation is assumed to be total and to occur atomically.

A *configuration* is a pair $\langle V, s\rangle$ where the vector V specifies the threads that are currently active and s defines the current state of the memory.

$$\frac{}{\langle \mathsf{skip}, s\rangle \rightarrowtriangle \langle\langle\rangle, s\rangle} \qquad \frac{\langle Exp, s\rangle \downarrow n}{\langle Id{:=}Exp, s\rangle \rightarrowtriangle \langle\langle\rangle, [Id = n]s\rangle}$$

$$\frac{\langle Exp', s\rangle \downarrow i \quad 0 \leq i < Arr.length \quad \langle Exp, s\rangle \downarrow n}{\langle Arr[Exp']{:=}Exp, s\rangle \rightarrowtriangle \langle\langle\rangle, [Arr[i] = n]s\rangle} \qquad \frac{\langle Exp', s\rangle \downarrow i \quad (i < 0 \vee i \geq Arr.length)}{\langle Arr[Exp']{:=}Exp, s\rangle \rightarrowtriangle \langle\langle\rangle, s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrowtriangle \langle\langle\rangle, t\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle \langle C_2, t\rangle} \qquad \frac{\langle C_1, s\rangle \rightarrowtriangle \langle\langle C_1'\rangle V, t\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle \langle\langle C_1'; C_2\rangle V, t\rangle} \qquad \frac{}{\langle \mathsf{fork}(CV), s\rangle \rightarrowtriangle \langle\langle C\rangle V, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \rightarrowtriangle \langle C_1, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \rightarrowtriangle \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle \rightarrowtriangle \langle C; \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle \rightarrowtriangle \langle\langle\rangle, s\rangle}$$

Fig. 1. Small-step deterministic semantics

$$\frac{\langle C_i, s\rangle \rightarrowtriangle \langle W, t\rangle}{\langle\langle C_0 \ldots C_{n-1}\rangle, s\rangle \rightarrow \langle\langle C_0 \ldots C_{i-1}\rangle W \langle C_{i+1} \ldots C_{n-1}\rangle, t\rangle}$$

$$\frac{\forall i \in \{0, \ldots, n-1\} : (C_i = \mathsf{sync} \wedge V_i' = \langle\rangle) \vee (C_i = \mathsf{sync}; D_i \wedge V_i' = \langle D_i\rangle)}{\langle\langle C_0, \ldots, C_{n-1}\rangle, s\rangle \rightarrow \langle V_0' \ldots V_{n-1}', s\rangle}$$

Fig. 2. Small-step non-deterministic semantics

The operational semantics is formalized in Figures 1 and 2. *Deterministic judgments* have the form $\langle\!\langle C, s\rangle\!\rangle \rightarrowtriangle \langle\!\langle W, t\rangle\!\rangle$ expressing that command C performs a computation step in state s, yielding a state t and a vector of commands W, which has length zero if C terminated, length one if it has neither terminated nor spawned any threads, and length greater than one if new threads were spawned. That is, a command vector of length n can be viewed as a *pool of n threads* that run concurrently. *Non-deterministic judgments* have the form $\langle\!\langle V, s\rangle\!\rangle \rightarrow \langle\!\langle V', t\rangle\!\rangle$ (note the new arrow), where V and V' are thread pools, expressing that some thread C_i in V performs a step in state s resulting in the state t and some thread pool W'. The global thread pool V' results then by replacing C_i with W'.

Our sync command blocks a given thread until each other thread has terminated or is blocked. Executing sync unblocks all threads (see the rule in Figure 2).

The following example illustrates the subtle possibilities for leaking information in a multi-threaded setting. It also demonstrates that the parallel composition of two secure programs can result in an insecure program.

Example 1. If $P_3 = h{:=}0; P_2$ (where $P_2 =$ if $h = 0$ then $l{:=}0$ else $l{:=}1$ fi) runs concurrently with $P_4 = h{:=}h'$ under a shared memory and a round robin scheduler then the final value of l is 0 (respectively, 1) given that the initial value of h' is 0 (respectively, not 0). This is illustrated below where $(v_l, v_h, v_{h'})$ denotes the state s with $s(l) = v_l$, $s(h) = v_h$, and $s(h') = v_{h'}$:

$$\begin{array}{ll}
\langle\!\langle \langle P_3, P_4\rangle, (0,0,0)\rangle\!\rangle & \langle\!\langle \langle P_3, P_4\rangle, (0,0,1)\rangle\!\rangle \\
\rightarrow \langle\!\langle \langle P_2, P_4\rangle, (0,0,0)\rangle\!\rangle & \rightarrow \langle\!\langle \langle P_2, P_4\rangle, (0,0,1)\rangle\!\rangle \\
\rightarrow \langle\!\langle \langle P_2\rangle, (0,0,0)\rangle\!\rangle & \rightarrow \langle\!\langle \langle P_2\rangle, (0,1,1)\rangle\!\rangle \\
\rightarrow \langle\!\langle \langle l{:=}0\rangle, (0,0,0)\rangle\!\rangle \rightarrow \langle\!\langle \langle\rangle, (0,0,0)\rangle\!\rangle & \rightarrow \langle\!\langle \langle l{:=}1\rangle, (0,1,1)\rangle\!\rangle \rightarrow \langle\!\langle \langle\rangle, (1,1,1)\rangle\!\rangle
\end{array}$$

That is, the final value of l equals the initial value of h' and, hence, the attacker is able to reconstruct the secret, initial value of h' from his observation of l. $\Diamond$

In the following, we adopt the naming conventions used so far: s and t denote states, *Exp* denotes an expression, B denotes a Boolean expression, *Arr* denotes an array, C and D denote commands, and V and W denote command vectors.

2.1 Security Policy, Labelings, and Security Condition

We assume a security lattice that comprises two security domains, a *high level* and a *low level* where the requirement is that no information flows from *high* to *low*. This is the simplest policy for which the problem of information flow security can be investigated. Each program variable is associated with a security domain by means of a *labeling* $lab : Var \rightarrow \{low, high\}$. The intuition is that values of *low variables* can be observed by the attacker and, hence, should only be used to store public data. *High variables* are used for storing secret data and their content is not observable for the attacker. For a given array *Arr*, the content has a security domain (denoted $lab(Arr)$) and the length has a security domain (denoted $lab(Arr.length)$) that must be at or below the one for the content. All elements of the array are associated with the same security domain. If $Arr : high$

then $Arr[i] : high$ and if $Arr : low$ and $i : low$ then $Arr[i] : low$. If $Arr : low$ and $i : high$ then $Arr[i]$ has no security domain and cannot be typed (see [10]).

As before, h and l denote high and low variables, respectively. An expression Exp has the security domain low (denoted by $Exp : low$) if all variables in Exp have domain low and, otherwise, has security domain $high$ (denoted by $Exp : high$). The intuition is that values of expressions with domain $high$ possibly depend on secrets while values of low expressions can only depend on public data.

Definition 1. *Two states $s, t \in S$ are* low equal *(denoted by $s =_L t$) iff*

$$\forall var \in Var : lab(var) = low \implies s(var) = t(var) \ .$$

Two expressions Exp, Exp' are low equivalent *(denoted by $Exp \equiv_L Exp'$) iff*

$$\forall s, s' \in S : (s =_L s' \wedge \langle\!| Exp, s |\!\rangle \downarrow n \wedge \langle\!| Exp', s' |\!\rangle \downarrow n') \implies n = n' \ .$$

We decided to use a possibilistic security condition (like in [31]) despite the fact that this condition is not entirely satisfactory from a practical perspective as it does not take scheduling into account (unlike the conditions in, e.g., [32, 30]) and, in particular, is not scheduler independent (unlike the condition in [29]). However, possibilistic security is conceptually simple and suitable for illustrating our verification technique, and this is our focus in this article.

Definition 2. *A symmetric relation R on command vectors is a* possibilistic low indistinguishability *iff for all $V, W \in \boldsymbol{Com}$ with $V \ R \ W$ the following holds:*

$$\begin{aligned} \forall s, s', t \in S : ((s =_L t \wedge \langle\!| V, s |\!\rangle \rightarrow^* \langle\!| \langle\rangle, s' |\!\rangle) \\ \Rightarrow \exists t' \in S : (\langle\!| W, t |\!\rangle \rightarrow^* \langle\!| \langle\rangle, t' |\!\rangle \wedge s' =_L t')). \end{aligned}$$

The union of all possibilistic low indistinguishabilities, $\sim_L$, is again a possibilistic low indistinguishability. Note that $\sim_L$ is transitive and symmetric, but not reflexive. For instance, $l{:=}h \sim_L l{:=}h$ does not hold. Intuitively, only programs with secure information flow are related to themselves.

Definition 3. *A program V is possibilistic low secure iff $V \sim_L V$.*

The idea of possibilistic security is that an observer without knowledge of the scheduler cannot infer from the values of low-level variables that some high variable did not have a particular value. That is, any low output that is possible after the system starts in a state s is also possible when the system starts in any other state that is low equal to s.

Example 2. It is easy to see that $P_1 = l{:=}h$ and $P_2 = \mathsf{if}\ h = 0\ \mathsf{then}\ l{:=}0\ \mathsf{else}\ l{:=}1\ \mathsf{fi}$, both are not possibilistic low secure. Moreover, P_3 and P_4 from Example 1, each is possibilistic low secure, but $\langle P_3, P_4 \rangle$ is not (take s and t as in Example 1). $\Diamond$

3 Combining Calculus

In general, compositional reasoning about information flow security is not sound. This applies, in particular, to our baseline condition, possibilistic low security,

which is neither preserved under parallel composition nor under sequential composition, in general (see Example 2 and below). For making compositional reasoning sound, one must strengthen the definition of secure information flow until one arrives at a compositional property. This approach is taken, e.g., in the derivation of the strong security condition [29]. However, the resulting composable security definitions are over-restrictive in the sense that they are violated by many programs that are intuitively secure.

In this section, we present an approach for deducing the security of a composed program from the fact that each sub-program satisfies some notion of security that is stronger than the baseline property. We derive sufficient conditions for sequential composition, for parallel composition, for conditional branching, and for while loops. This leads us to four compositionality results. These constitute the theoretical basis of our combining calculus, which allows one to flexibly apply available verification techniques during an information flow analysis. We then revisit some available verification techniques and provide plugin-rules that enable the use of these techniques in a derivation with our combining calculus.

3.1 Compositionality Results and Basic Calculus Rules

Auxiliary concepts. If $C \sim_L C'$ and $D \sim_L D'$ hold then $C; D \sim_L C'; D'$ does not necessarily hold because threads spawned during execution of C might still be running when D begins execution, influencing computations in D through shared variables. For instance, the program $\mathsf{fork}(\mathsf{skip},\ P_2; l{:=}2); l'{:=}l$ where $P_2 = \mathsf{if}\ h = 0\ \mathsf{then}\ l{:=}0\ \mathsf{else}\ l{:=}1\ \mathsf{fi}$ does not satisfy the baseline property (due to the race between the second assignment to l and the assignment to l') although it is the sequential composition of two programs that both satisfy the baseline property. If the main thread is the last thread to terminate before D (respectively D') can begin execution then such problems cannot occur.

Definition 4. *A thread pool V is* main-surviving *(denoted by $\mathsf{MS}(V)$), if for arbitrary states s and t as well as for each thread pool $\langle C_0, \ldots, C_{n-1}\rangle$ with $\langle\!|V, s|\!\rangle \to^* \langle\!|\langle C_0, \ldots, C_{n-1}\rangle, t|\!\rangle$ one of the following two conditions holds:*

- *There is no state t' such that $\langle\!|C_0, t|\!\rangle \twoheadrightarrow \langle\!|\langle\rangle, t'|\!\rangle$.*
- $n = 1$.

One can make a program main-surviving by adding sync statements. Consider as an example the program $\mathsf{fork}(h := 0,\ \ h := h')$, which is not main-surviving as both conditions in Definition 4 are violated. Main-surviving programs are, e.g., $\mathsf{fork}(h := 0,\ \ h := h'); \mathsf{sync}$ and $\mathsf{fork}(\mathsf{sync}; h := 0,\ \ h := h')$.

Parallel composition shares the problems of sequential composition: given $V \sim_L V'$ and $W \sim_L W'$ one does not necessarily obtain $VW \sim_L V'W'$. This is caused by shared variables, which allow one thread to influence the behavior of another thread. Even if the composed thread pools have no low variables in common, we do not obtain a general compositionality theorem (see Example 1). A sufficient condition for preserving low indistinguishability is the disjointness of all variables.

Definition 5. *We say that two thread pools V and W are* variable independent *($V \gtrless W$) if the sets of variables occurring in V respectively W are disjoint.*

Compositionality. We are now ready to present our compositionality results:

Theorem 1. *Let C, C', D, and D' be commands and V, V',W, and W' be thread pools such that $C \sim_L C', D \sim_L D', V \sim_L V'$ and $W \sim_L W'$. Then*

1. *if C and C' are main-surviving then $C; D \sim_L C'; D'$;*
2. *if $V \gtrless W$ and $V' \gtrless W'$ then $VW \sim_L V'W'$;*
3. *if $B \equiv_L B'$ then if B then C else D fi $\sim_L$ if B' then C' else D' fi; and*
4. *if $B \equiv_L B'$ and C and C' are main-surviving, then while B do C od $\sim_L$ while B' do C' od.*

A note with the proof of Theorem 1 is available on the authors' homepage.

Basic calculus rules. We raise the possibility for compositional reasoning about low indistinguishability with Theorem 1 to compositional reasoning about information flow security. This results in the calculus rules depicted below. The judgment $\vdash \mathsf{bls}(V)$ intuitively means that the program V is possibilistic low secure. A soundness result is provided in Section 3.4.

$$[\text{SEQ}]\frac{\vdash \mathsf{bls}(C) \quad \vdash \mathsf{bls}(D) \quad \mathsf{MS}(C)}{\vdash \mathsf{bls}(C; D)} \quad [\text{PAR}]\frac{\vdash \mathsf{bls}(V) \quad \vdash \mathsf{bls}(W) \quad V \gtrless W}{\vdash \mathsf{bls}(VW)}$$

$$[\text{ITE}]\frac{\vdash \mathsf{bls}(C) \quad \vdash \mathsf{bls}(D) \quad B \equiv_L B}{\vdash \mathsf{bls}(\mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ D\ \mathsf{fi})} \quad [\text{FRK}]\frac{\vdash \mathsf{bls}(\langle C \rangle V)}{\vdash \mathsf{bls}(\mathsf{fork}(CV))}$$

$$[\text{WHL}]\frac{\vdash \mathsf{bls}(C) \quad \mathsf{MS}(C) \quad B \equiv_L B}{\vdash \mathsf{bls}(\mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od})} \quad [\text{SNC}]\frac{\vdash \mathsf{bls}(C)}{\vdash \mathsf{bls}(C; \mathsf{sync})}$$

It should be noted that it is not intended that one proves the security of a complex program solely with the above rules. There are many secure programs for which the side conditions main surviving and variable independence are too restrictive. For analyzing such programs with the combining calculus, one employs plugin rules. The combining calculus is not intended as an alternative to existing security-analysis techniques, but rather as a vehicle for using different analysis techniques in combination. The plugins presented in the following, in particular, allow one to analyze programs that contain races.

3.2 Plugin: Strong Security

Definition 6 ([29]). *The* strong low-bisimulation $\approx_L$ *is the union of all symmetric relations R on command vectors $V, V' \in \boldsymbol{Com}$ of equal size, i.e. $V = \langle C_0, \ldots, C_{n-1} \rangle$ and $V' = \langle C'_0, \ldots, C'_{n-1} \rangle$, such that*

$$\begin{aligned}&\forall s, s', t \in S : \forall i \in \{0, \ldots, n-1\} : \forall W \in \boldsymbol{Com}:\\ &\ [(V\ R\ V' \wedge s =_L s' \wedge \langle\!\langle C_i, s \rangle\!\rangle \twoheadrightarrow \langle\!\langle W, t \rangle\!\rangle)\\ &\quad \Rightarrow \exists W' \in \boldsymbol{Com}: \exists t' \in S : (\langle\!\langle C'_i, s' \rangle\!\rangle \twoheadrightarrow \langle\!\langle W', t' \rangle\!\rangle \wedge W\ R\ W' \wedge t =_L t')]\ .\end{aligned}$$

Note that $\cong_L$ is only a partial equivalence relation, i.e. it is transitive and symmetric, but not reflexive. In fact, $\cong_L$ only relates secure programs to themselves (note the structural similarity to the relationship between Definitions 2 and 3).

Definition 7 ([29]). *A program V is* strongly secure *iff $V \cong_L V$ holds.*

The strong security condition is scheduler independent and enjoys compositionality results that make it a suitable basis for a compositional security analysis.

Theorem 2 ([29, 22]). *Let C, D and V be strongly secure programs that do not contain* sync *statements. If $B \equiv_L B$ then $C; D$, fork(CV), if B then C else D fi, and* while B do C od *are strongly secure. If $C \cong_L D$ holds then* if B then C else D fi *is also strongly secure (even for B : high).*

Proof. [22] extends the proof in [29] to the language with arrays. □

The strong security condition constitutes a conservative approximation of our security definition as the following theorem demonstrates.

Theorem 3. *If V is strongly secure and does not contain any* sync *statements, then V is possibilistic low secure.*

Proof. Let $s =_L t$. If $\langle V, s\rangle \rightarrow^* \langle\langle\rangle, s'\rangle$) then one can, by applying Definition 6, inductively construct (over the length of the computation sequence) a computation $\langle W, t\rangle \rightarrow^* \langle\langle\rangle, t'\rangle$ of the same length such that $s' =_L t'$. □

While the strong security condition can be suitable for reasoning about secure information flow, there are also situations where it is too restrictive.

Example 3. The programs $l := h; l := 1$, if h then skip else skip; skip fi, and while $h > 0$ do $h := h - 1$ od all have secure information flow (according to Definition 3). However, none of these programs is strongly secure. ♢

The problems in Example 3 can be overcome by applying our combining calculus, in which strong security constitutes only one of several plugins. Its plugin rule is depicted to the right. When this rule is applied, the premise could be proved, e.g., with a security type system (see Section 5), or with some general-purpose theorem prover.

$$[\mathrm{P}_{SLS}]\frac{V \cong_L V \quad V \text{ is sync-free}}{\vdash \mathsf{bls}(V)}$$

3.3 Plugin: Low-Deterministic Security

Roscoe pioneered a characterization of information flow security based on the notion of low determinism. The resulting security definitions for the process algebra CSP [23] are intuitively convincing as they ensure that the low-level behavior of a process is deterministic, no matter what the high-level behavior is. A disadvantage, however, is that it is unnecessarily restrictive with respect to nondeterministic system behavior on the low level. Zdancewic and Myers [33] argue that this disadvantage is acceptable when the approach is applied to concrete programs. We adopt this approach to our setting.

Definition 8. *A program V is* low-deterministic secure *iff*

$$\forall s,t,s',t' \in S : [(s =_L t \wedge \langle\!\langle V,s\rangle\!\rangle \rightarrow^* \langle\!\langle\langle\rangle,s'\rangle\!\rangle \wedge \langle\!\langle V,t\rangle\!\rangle \rightarrow^* \langle\!\langle\langle\rangle,t'\rangle\!\rangle) \implies s' =_L t'].$$

That is, if one runs a program that is low-deterministic secure in two arbitrary starting states that are low equal then all final states are also low equal.

Theorem 4. *Let V be a program that is low-deterministic secure. Assume further, that if the program can terminate in some state it can terminate in each low equal state (written $\mathsf{PLT}(V)$). Then V is possibilistic low secure.*

Proof. Let s,s',t,t' be states such that $s =_L t$. Assume that $\langle\!\langle V,s\rangle\!\rangle \rightarrow^* \langle\!\langle\langle\rangle,s'\rangle\!\rangle$ for some state s'. By assumption, V can terminate in t. Hence, there exists $t' \in S$ such that $\langle\!\langle V,t\rangle\!\rangle \rightarrow^* \langle\!\langle\langle\rangle,t'\rangle\!\rangle$. From Definition 8, we obtain $s' =_L t'$. □

In the plugin-rule depicted to the right, we use the judgment $\models \mathit{lds}(V)$. This judgment captures the intuition that V is low-deterministic secure. Again, first-order logic could be used to express and prove the semantic preconditions.

$$[\mathrm{P}_{LDS}]\frac{\models \mathit{lds}(V)\ ,\ \mathsf{PLT}(V)}{\vdash \mathsf{bls}(V)}$$

3.4 Soundness and Examples

The combining calculus is sound in the following sense:

Theorem 5. *Let V be a program such that $\vdash \mathit{bls}(V)$ is derivable in the combining calculus. Then V is possibilistic low secure.*

Proof. The soundness of the rules [SEQ], [PAR], [ITE], and [WHL] follows directly from Theorem 1, while the soundness of rule [FRK] follows from the soundness of [PAR], the definition of possibilistic low security, and the operational semantics. Rule [SNC] is sound since, firstly, a sync statement does not change the state, and, secondly, the sync statement is appended at the end of the command C and therefore does not retard the execution of subsequent commands. The plugin-rules are sound by Theorems 3 and 4. □

We illustrate the usage of the combining calculus with a simple example. Consider the program $\mathsf{fork}(l := 0,\ \ l := 1); \mathsf{sync}; \mathsf{while}\ h \leq 5\ \mathsf{do}\ h := h+1\ \mathsf{od}$. By applying [SEQ] we obtain three new proof obligations, firstly $\vdash \mathsf{bls}(\mathsf{fork}(l := 0,\ \ l := 1); \mathsf{sync})$, secondly $\vdash \mathsf{bls}(\mathsf{while}\ h \leq 5\ \mathsf{do}\ h := h+1\ \mathsf{od})$, and thirdly $\mathsf{MS}(\mathsf{fork}(l := 0,\ \ l := 1); \mathsf{sync})$. The first one can be proved by the application of [SNC] and subsequently [P_{SLS}], followed by an analysis of strong security, while the second one can be proved by the application of [P_{LDS}], followed by an analysis of low-deterministic security. The third obligation is obviously true. Strong security does not suffice to prove the program secure, since while loops with high guards are rejected; an analysis of the whole program with low-deterministic security would also fail due to the race between $l := 0$ and $l := 1$.

4 Information Flow Security of a PDA Application

In this section, we illustrate how the possibility of combining proof techniques can be exploited in a concrete security analysis. The security of the example program can be successfully verified by combining strong security and low-deterministic security, while none of these security definitions alone provides a suitable basis for the analysis. The example application is a multi-threaded program for managing finances on mobile devices. The program gives an overview of the current stock portfolio, possibly illustrating profits, losses, and other trends with statistics. When the user starts the application he obtains a listing of his portfolio, revealing name and quantity for each stock. In parallel to printing, the current rates of EuroStoxx50 entries are retrieved. When all data is available, informative statistics can be computed. For minimizing idle time during this computation, a background thread already incrementally prepares the printout of the statistics. Finally the statistics is displayed, together with a pay-per-click commercial.

```
fork
(
 //getPortfolio:
esOP_l:= getES50old;
i_h:=0; pfName_h:=getPFNames;
pfNum_h:=getPFNum;
while (i_h<pfName_h.length) do
  pfTabPrint_h:= pfName_h[i_h] + "|"
    + pfNum_h[i_h];
  i_h:= i_h+ 1 od
,
//getEuroStoxx50:
j_l:=0; nwOutBuf_l:= getES50;
while (nwInBuf_l= "") do skip od;
strArr_l:= split(nwInBuf_l, ":");
while (j_l<50) do
  esName_l[j_l] := strArr_l[2*j_l];
  esP_l[j_l] := strArr_l[2*j_l+1];
  j_l:= j_l+1 od;
coShort_l:= strArr_l[100];
coFull_l:= strArr_l[101]; coId_l:= strArr_l[102]
)
;sync;
fork
(
//computeStatistics:
k_l:=0;
while (k_l<50) do
  IPF_h:= locPF(esName_l[k_l], pfName_h);
  //calculate profit for stock at position k_l
  st_h[k_l]:=(esOP_l[k_l] -esP_l[k_l])* pfNum_h[IPF_h]
  k_l:= k_l+1 od
,
//generateOutput:
m_l:=0;
while (m_l<50) do
  while (k_l≤m_l) do skip od;
  outL_h[m_l] := m_l+ "|"
    + esName_l[m_l]+ "|"
    + esP_l[m_l] + "|" +     st_h[m_l];
  m_l:= m_l+1 od
)
//displayOutputAndCommercial:
;n_l:=0; stTabPrint_h("No. | Name | Price | Profit");
while (n_l<50) do   stTabPrint_h:= outL_h[n_l];   n_l:= n_l+ 1 od;
stTabPrint_h:= coShort_l+ "Press # to get more information.";
while (key_l= ' ') do skip od;
if (key_l≠ '#') then   coDispPrint_h:= coFull_l;   nwOutBuf_l:= "shownComm:"+ coId_l
else skip fi
```

Fig. 3. Implementation

The implementation of the application (Figure 3) is divided into five blocks: reading the portfolio from non-volatile storage (getPortfolio), retrieving current stock rates (getEuroStoxx50), computing statistics (computeStatistics), preparing a printout of the statistics (generateOutput), displaying the printout, advertising the commercial by a preview, and waiting for the user's input (displayOutputAndCommercial). If the user decides to view the commercial, it is displayed in full and a confirmation message is sent to the server.

As an example, we give a detailed description of getEuroStoxx50: After the initialization of the loop variable j_l (where the subscript l indicates that j is a low variable), a request is sent to the network interface represented by the variable $\mathsf{nwOutBuf}_l$. Due to the lack of interrupts we have to do busy waiting until the variable $\mathsf{nwInBuf}_l$ representing the incoming network stream contains an answer. The answer is a string (sequence of ASCII numbers) containing name and current rate of each stock listed in the EuroStoxx50, separated by colons. To avoid a second network request, the commercial, including the preview, the full version, and a reference ID are already included, again separated by colons. The operation split in the third line of getEuroStoxx50 is similar to the method split of the Java String class. It splits a single string in an atomic step into an array of strings, which then is processed further in the subsequent loop. After extracting the commercial data from the array its memory could be deallocated (but this is outside our language).

We assume that the application is running in a sandbox that protects the memory from programs outside the sandbox. The only exception is the underlying operating system with whom the application communicates via predefined interface variables. Besides the two interface variables for network communication ($\mathsf{nwInBuf}_l$, $\mathsf{nwOutBuf}_l$), the program uses display variables ($\mathsf{pfTabPrint}_h$, $\mathsf{stTabPrint}_h$, $\mathsf{coDispPrint}_h$), variables that represent parts of the non-volatile storage (getES50old, getPFNames, getPFNum), and the keyboard variable (key_l). Assignments to these variables in the program correspond to the output of the information on the associated interface. Reading these variables corresponds to retrieving input through the operating system.

The parallel execution of getEuroStoxx50 and getPortfolio prevents blocking during time-consuming network activity. Concurrent programming increases efficiency and also complies with programming recommendations for mobile devices like, e.g., [14, 16]. For simplicity, computeStatistics calculates only the user's profit for each stock. One could easily imagine more complex statistics. The atomic operation locPF in computeStatistics locates the index of the kth stock value within the portfolio and returns -1 if the value is not present.

The secret to be protected in the given scenario is the content of the portfolio. The sink where this information could be leaked is the network interface (assuming that the display is only accessible for users who are permitted to read the printouts). Both assignments to the $\mathsf{nwOutBuf}_l$ are intuitively secure. Hence, there is no direct leakage of secrets and starting a more detailed information flow analysis is appropriate. For the security analysis, we use a combination of low-deterministic security and strong security. The strong security of a program

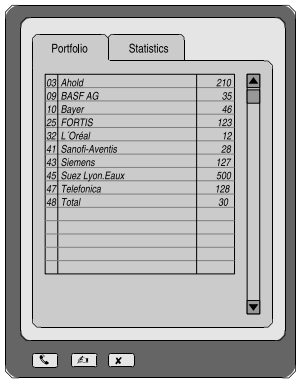

Fig. 4. Portfolio Tab

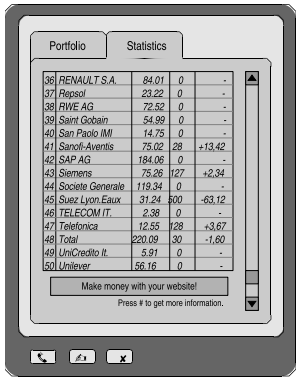

Fig. 5. Statistics Tab

Fig. 6. Commercial Screen

implies that the run-time of this program is independent of the initial value of high variables. This is obviously not the case for the loop in getPortfolio, where the run-time is directly influenced by the value of the high variable $\mathsf{pfName}_h.\mathit{length}$. However, each of the five program blocks can be successfully analyzed. The result of this investigation is expressed by the following two theorems. Due to space restrictions we only sketch the proof of the first one.

Theorem 6. *The program* getPortfolio *is low-deterministic secure.*

Proof. Since i_h is incremented in the body of the loop, the loop will eventually terminate. Moreover, the only assignment to a low variable, esOP_l := getES50old, does not depend on the initial high values. Hence the final value of low variables depends deterministically on their initial values. □

Theorem 7. *The programs* getEuroStoxx50, computeStatistics, generateOutput, *and* displayOutputAndCommercial *are strongly secure.*

From these two theorems and the compositionality of strong security, we conclude that the program fork(computeStatistics, generateOutput); displayOutputAndCommercial is strongly secure. From the plugin-rules [P_{SLS}] and [P_{LDS}], we obtain that getPortfolio and getEuroStoxx50 both satisfy the baseline policy. The parallel execution of these programs also satisfies the baseline policy according to rule [PAR], since variable independence holds. After an application of [FRK], an application of [SNC], and an application of [SEQ], we conclude that the entire program satisfies the baseline property. Hence the program is possibilistic low secure.

The application shows that the combining calculus is applicable for fairly realistic programs. The advantages will become even clearer in Section 5 where we integrate security type systems. Using a type system for the strong security condition, one can efficiently verify four parts of the program and only the remaining part would require a semantic check of low-deterministic security (for which no suitable calculus is available yet).

5 Plugins for Type-based Analysis Techniques

While Sections 3 and 4 presented plugin rules for semantic security definitions, this section illustrates how syntactic, type-based analysis techniques can be integrated and beneficially exploited. We provide two additional plugins for the combining calculus: one to integrate the security type system proposed in [6] and one to integrate the security type system from [29]. When introducing the second type system, we also illustrate the possibility to integrate transforming type systems. Such type systems may generate a secure program from a given, possibly insecure program. Additionally, we show how to combine transforming and non-transforming analysis techniques.

5.1 Plugin : Boudol and Castellani's Security Type System

In [6] Boudol and Castellani propose a type system that does not generally reject programs containing loops with high guards, unlike the type systems in, e.g., [29] or [31]. The type judgments are of the form $\Gamma \vdash C : (\tau, \sigma)\ cmd$, where C is a command, τ and σ are security labels, and the context Γ is a mapping from variables to security labels. In the type judgment, τ is a lower bound for the level of the variables to which assignments are made in C, and σ is an upper bound for the security levels occurring in the guards of loops and conditionals in C. After adapting the typing rules to our language, fixing a variable labeling and the induced context Γ, we obtain the following result:

Theorem 8. *Let C be a command that always terminates. If $\Gamma \vdash C : (\tau, \sigma)\ cmd$ can be derived for some security labels τ and σ, then C is possibilistic low secure.*[1]

For programs that always terminate we obtain the plugin rule depicted to the right. The combining calculus extended by this rule is sound due to Theorem 8.

$$[\mathrm{T}_{BC}]\frac{\Gamma \vdash C : (\tau, \sigma)\ cmd}{\vdash \mathsf{bls}(C)}$$

5.2 Plugin : Sabelfeld and Sand's Security Type System

In [29] Sabelfeld and Sands propose a transforming type system approximating the strong security condition. Its judgments are of the form $V \hookrightarrow V' : Sl$, where V is the program to be checked, V' a transformation of the program, and Sl is the type of V'. The type contains auxiliary information that is used for the transformation of the program. They provide the following theorem:

Theorem 9 ([29]). *Whenever $V \hookrightarrow V' : Sl$, then $V' \cong_L V'$.*

That is, when the type check succeeds, then the transformed program is strongly secure. To integrate plugins for transforming type systems we extend the combining calculus with the transforming rules in Figure 7. The intuition of the judgment $\vdash C \hookrightarrow \mathsf{bls}(C')$ is that the program C is transformed into the possibilistic

$$[\text{SEQ'}]\frac{\vdash C \hookrightarrow \mathsf{bls}(C') \quad \vdash D \hookrightarrow \mathsf{bls}(D') \quad \mathsf{MS}(C')}{\vdash C; D \hookrightarrow \mathsf{bls}(C'; D')}$$

$$[\text{PAR'}]\frac{\vdash V \hookrightarrow \mathsf{bls}(V') \quad \vdash W \hookrightarrow \mathsf{bls}(W') \quad V' \gtrless W'}{\vdash VW \hookrightarrow \mathsf{bls}(V'W')}$$

$$[\text{ITE'}]\frac{\vdash C \hookrightarrow \mathsf{bls}(C') \quad \vdash D \hookrightarrow \mathsf{bls}(D') \quad B \equiv_L B}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ D\ \mathsf{fi} \hookrightarrow \mathsf{bls}(\mathsf{if}\ B\ \mathsf{then}\ C'\ \mathsf{else}\ D'\ \mathsf{fi})}$$

$$[\text{FRK'}]\frac{\vdash \langle C\rangle V \hookrightarrow \mathsf{bls}(\langle C'\rangle V')}{\vdash \mathsf{fork}(CV) \hookrightarrow \mathsf{bls}(\mathsf{fork}(C'V'))} \qquad [\text{SNC'}]\frac{\vdash C \hookrightarrow \mathsf{bls}(C')}{\vdash C; \mathsf{sync} \hookrightarrow \mathsf{bls}(C'; \mathsf{sync})}$$

$$[\text{MIX}_1]\frac{\vdash \mathsf{bls}(C)}{\vdash C \hookrightarrow \mathsf{bls}(C)} \qquad [\text{MIX}_2]\frac{\vdash C \hookrightarrow \mathsf{bls}(C')}{\vdash \mathsf{bls}(C')}$$

Fig. 7. Additional rules for the combining calculus

low secure program C'. The rules $[\text{MIX}_1]$ and $[\text{MIX}_2]$ permit the combination of transforming as well as non-transforming analysis techniques. The first one relies on the fact that a possibilistic low secure program can be securely transformed into itself.

The soundness proof of the extended calculus goes along the same lines as the proof of Theorem 5. We are now ready to add a plugin for Sabelfeld's and Sand's proof technique. The addition is sound due to Theorem 9 and Theorem 3.

$$[\text{T}_{SS}]\frac{V \hookrightarrow V' : Sl \quad V' \text{ is } \mathsf{sync}\text{-free}}{\vdash V \hookrightarrow \mathsf{bls}(V')}$$

5.3 Exemplary Type-based Security Analysis

We exemplify the use of the plugin rules $[\text{T}_{BC}]$ and $[\text{T}_{SS}]$ with a syntactical analysis of the program from Section 4. We already argued that some blocks of the program are strongly secure. Hence we use the combining calculus rules supporting transforming type systems. After applying rule [SEQ']

1. $\mathsf{MS}(\mathsf{fork}(\mathsf{getPortfolio},\ \mathsf{getEuroStoxx50}))$,
2. $\vdash \mathsf{fork}(\mathsf{getPortfolio},\ \mathsf{getEuroStoxx50}) \hookrightarrow \mathsf{bls}(C)$, and
3. $\vdash \mathsf{fork}(\mathsf{computeStatistics},\ \mathsf{generateOutput}); \mathsf{displayOutputAndCommercial} \hookrightarrow \mathsf{bls}(D)$

remain to be derived in the calculus. The first statement can be syntactically shown, since the first thread, getPortfolio, does not contain any conditionals and ends with a sync statement, while the second thread, getEuroStoxx50, does not contain any sync statements.

For the second proof obligation we do not use transforming type systems. We hence instantiate C with $\mathsf{fork}(\mathsf{getPortfolio},\ \mathsf{getEuroStoxx50})$ and apply rule $[\text{MIX}_1]$, obtaining $\vdash\ \mathsf{bls}(\mathsf{fork}(\mathsf{getPortfolio},\ \mathsf{getEuroStoxx50}))$. After applying rule [PAR], we get three new proof obligations, namely $\mathsf{getPortfolio} \gtrless \mathsf{getEuroStoxx50}$,

[1] The typing rules differ slightly from the ones used in [6]. The adapted rules and the soundness argument will be provided in a technical report.

$\vdash$ bls(getPortfolio) and $\vdash$ bls(getEuroStoxx50). The first statement can be easily verified syntactically. Since getPortfolio and getEuroStoxx50 are programs that always terminate given that the network always answers the network request and that can be checked automatically with the type system provided by Boudol and Castellani we apply the rule $[\mathrm{T}_{BC}]$ to the other two statements and obtain the new proof obligations $\Gamma \vdash$ getPortfolio $: (\tau, \sigma)$ *cmd* and $\Gamma \vdash$ getEuroStoxx50 $: (\tau', \sigma')$ *cmd*. Now we need to continue using rules of the adapted type system from [6]. One can deduce that getPortfolio can be typed with (L, H) *cmd* (getPortfolio contains assignments to low variables and high guards, but the low assignment happens before the loop), while getEuroStoxx50 can be typed with (L, L) *cmd* (getEuroStoxx50 contains assignments to low variables, but no high guards).

For the third proof obligation we apply the rule $[\mathrm{T}_{SS}]$, obtaining the obligation fork(computeStatistics, generateOutput); displayOutputAndCommercial $\hookrightarrow D : Sl$. For a deduction we use Sabelfeld's and Sand's transforming type system. Since neither computeStatistics, nor generateOutput, nor displayOutputAndCommercial contain high guards the type system does not perform any modification and we obtain $E \hookrightarrow E : Sl$ for some type Sl and $E =$ fork(computeStatistics, generateOutput); displayOutputAndCommercial.

Due to space restrictions we omit a more detailed derivation.

6 Conclusion

Obviously, the idea of combining different proof techniques is no novelty. The contribution of this article is the illustration of how one can benefit more concretely from combining proof techniques in the information flow analysis of a given program. To our knowledge, no such result was presented before. Moreover, we introduced the combining calculus as a deductive framework that is based on conditional compositionality results and an extensible set of plugin-rules for existing verification techniques. As examples, we presented plugin-rules for restrictive security characterizations (strong security and low-determinism security), which could be verified with general-purpose logics, and plugin-rules for typing judgments that can be derived with security type systems, i.e. special-purpose calculi. We illustrated both possibilities in a fairly realistic example program. The addition of further plugin-rules would be desirable, for instance, to support verification techniques with program-logics (see, e.g., [2, 7]).

Based on the experiences gained, our impression is that a baseline characterization of information flow security need not be fully compositional, which is in contrast, e.g., to the opinion stated in [21]. Nevertheless, the baseline characterization employed in the current article, which is a possibilistic property (like, e.g., in [31, 6]), requires further improvements, in particular, regarding scheduling aspects. We are currently researching a security definition that is scheduler independent, but less restrictive than strong security or low-determinism security (which are both scheduler independent). Strong security is known to be the least restrictive security definition that is scheduler independent and

compositional [27]. However, as we are not requiring full compositionality, less restrictive characterizations that can serve as a justification of our combining calculus exist (without changing the calculus), where the disjunction of strong security and low-determinism security is an obvious candidate.

Another direction is the migration to practically relevant languages such as Java source code or bytecode. In this context, approaches for sequential sub-languages are available (see, e.g., [1, 3]), and it is not obvious how to generalize them to a multi-threaded setting. Hence, the possibility of creating a combining calculus for Java with plugin-rules for such approaches is attractive and appears, in principle, possible with the help of a rule like [PAR].

Acknowledgments. This work was funded in part by the German Research Association (DFG) in the Computer Science Action Program and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST- 2005-015905 MOBIUS project. This article reflects only the authors' views and the Commission, the DFG, and the authors are not liable for any use that may be made of the information contained therein.

References

1. A. Banerjee and D. A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *IEEE Computer Security Foundations Workshop*, pages 155–169, 2003.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In *IEEE Computer Security Foundations Workshop*, pages 100–114, 2004.
3. G. Barthe and T. Rezk. Non-Interference for a JVM-like Language. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 103–112, 2005.
4. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Unwinding in Information Flow Security. *ENTCS 99*, pages 127–154, 2004.
5. A. Bossi, D. Macedonio, C. Piazza, and S. Rossi. Secure Contexts for Confidential Data. In *IEEE Computer Security Foundations Workshop*, pages 14–25, 2003.
6. Gérard Boudol and Ilaria Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.
7. Á. Darvas, R. Hähnle, and D. Sands. A Theorem Proving Approach to Analysis of Secure Information Flow. In *International Conference on Security in Pervasive Computing*, LNCS 3450, pages 193–209, 2005.
8. Z. Deng and G. Smith. Lenient Array Operations for Practical Secure Information Flow. In *IEEE Computer Security Foundations Workshop*, pages 115–124, 2004.
9. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
10. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.
11. R. Focardi and R. Gorrieri. A Classification of Security Properties for Process Algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
12. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

13. J. A. Goguen and J. Meseguer. Inference Control and Unwinding. In *IEEE Symposium on Security and Privacy*, pages 75–86, 1984.
14. J. Knudsen. *Networking, User Experience, and Threads*, 2002. http://developers.sun.com/techtopics/mobility/midp/articles/threading/.
15. B. Köpf and H. Mantel. Eliminating Implicit Information Leaks by Transformational Typing and Unification. In *International Workshop: Formal Aspects in Security and Trust, Revised Selected Papers*, LNCS 3866, pages 47–62. Springer-Verlag, 2006.
16. Q. H. Mahmoud. *Preventing Screen Lockups of Blocking Operations*, 2004. http://developers.sun.com/techtopics/mobility/midp/ttips/screenlock/.
17. H. Mantel. Possibilistic Definitions of Security – An Assembly Kit. In *IEEE Computer Security Foundations Workshop*, pages 185–199, 2000.
18. H. Mantel. Unwinding Possibilistic Security Properties. In *European Symposium on Research in Computer Security*, LNCS 1895, pages 238–254, 2000.
19. D. McCullough. Specifications for Multi-Level Security and a Hook-Up Property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
20. J. D. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.
21. J. K. Millen. Hookup Security for Synchronous Machines. In *IEEE Symposium on Research in Security and Privacy*, pages 84–90, 1990.
22. C. Pöpper. A Security Analyzer for Multi-Threaded Programs. Diploma thesis, ETH Zurich, March 2005.
23. A. W. Roscoe. CSP and Determinism in Security Modelling. In *IEEE Symposium on Security and Privacy*, pages 114–127, 1995.
24. J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.
25. A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *IEEE Computer Security Foundations Workshop*, 2006.
26. P. Y. A. Ryan and S. A. Schneider. Process Algebra and Non-interference. In *IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.
27. A. Sabelfeld. Confidentiality for Multithreaded Programs via Bisimulation. In *Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS 2890, pages 260–274, 2003.
28. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.
29. A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *IEEE Computer Security Foundations Workshop*, pages 200–215, 2000.
30. G. Smith. Probabilistic Noninterference through Weak Probabilistic Bisimulation. In *IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.
31. G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *ACM Symposium on Principles of Programming Languages*, pages 355–364, 1998.
32. D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *IEEE Computer Security Foundations Workshop*, pages 34–43, 1998.
33. S. Zdancewic and A. C. Myers. Observational Determinism for Concurrent Program Security. In *IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.