Special Issue

Tobias Nipkow and Andrei Popescu*

Making security type systems less ad hoc

Abstract: We present a uniform, top-down design method for security type systems applied to a parallel while-language. The method takes the following route: from a notion of *end-to-end security* via a collection of stronger notions of *anytime security* targeting compositionality to a matching collection of type-system-like syntactic criteria. This method has emerged by distilling and unifying security type system results from the literature while formalizing them in a proof assistant. Unlike in our previous papers on this topic, here we focus entirely on high-level ideas instead of technical proof details.

Keywords: ACM CCS → Security and privacy → Software and application security, ACM CCS → Computing methodologies → Concurrent computing methodologies, ACM CCS → Theory of computation → Semantics and reasoning.

DOI 10.1515/itit-2014-1060
Received June 4, 2014; revised October 10, 2014; accepted October 17, 2014

1 Introduction

A type system is a syntactic representation of aspects of a program semantics. If a function f has type $\mathsf{int} \to \mathsf{bool}$, we know its semantics will map integers to booleans. A major use of a type system is to ensure that, if well-typed, a program "does not go wrong." Traditionally, this means that a program's execution does not get stuck or produce low-level errors. On the other hand, a *security type system* guarantees a quite different property: *a program's execution does not leak private information.* For example, if h stores high (confidential) data and l is a low (publicly observable) variable, then the assignment $l := h$ exhibits a direct leak from high to low and is rejected by typical security type systems. The following indirect leak, exposing in l the information on whether h is 0, is also rejected: $\mathsf{if}\ h = 0\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 2$.

Type-system-like syntactic criteria for leak prevention have been discussed as early as the seventies (e. g., Denning and Denning [5]). However, it was the work of Volpano, Smith and others in the nineties [14–16] that started the tradition of *semantically justified* security type systems. In this tradition, one provides a type system together with a *soundness theorem* that relates it with a *semantic notion of security* (formulated via the programs' semantics). To express semantic security, one typically assumes the program memory is separated into a *low*, or public, part, which an attacker is able to observe, and a *high*, or private, part, hidden to the attacker. A program is called *information-flow secure* (or *noninterfering*) if, upon running it, the high part of the initial memory does not affect the low part of the resulting memory. Thus, there is no information leak from high to low. The soundness theorem states that well-typed programs are secure.

A variety of pairs consisting of a security type system and a semantic notion of security connected by a soundness theorem were proposed in the meantime (e. g., [3, 4, 6, 7, 11–14]), covering different types of information leak channels, including termination and probabilistic channels. As part of our work [2] within the RS^3 program [1], we studied some of these security type systems in the light of the three RS^3 guiding themes: property-centric view of security (by studying in depth the underlying security properties before committing to type-system-like policies), semantically justified certification of security (by emphasizing formalized semantics as a foundation), and security in the large (by focusing on compositionality). Type system design is known to be tightly linked to a *compositional* representation of the properties one wishes to enforce, and Sabelfeld and Sands [13] explicitly remind us of this in the security context. However, this is *not* how the Volpano-Smith-style type systems appear to have been designed, as witnessed by their complex non-compositional soundness proofs (e. g., [3, 4, 14]).

Our goal was to investigate whether such proofs can be simplified using compositionality. This paper describes the positive outcome of this investigation, yielding a proposal for a "canonical" route for designing security type systems: End-to-end security ⇒ Anytime security ⇒ Type

***Corresponding author: Andrei Popescu,** Fakultät für Informatik, Technische Universität München, e-mail: uuomul@yahoo.com
Tobias Nipkow: Fakultät für Informatik, Technische Universität München

system. (More technical presentations of these ideas can be found in [8, 9] for possibilistic and in [10] for probabilistic security.)

We restrict our attention to a concurrent imperative language with a nondeterministic (possibilistic) semantics (Section 2). In this context, we tell the story of the top-down design of security type systems. We start with *end-to-end security*, describing a desired security property of the program in terms of the relationship between the initial states and the final states obtained from fully executing the program (Section 3). End-to-end security is not compositional – to address this, we introduce stronger properties, called *anytime security*, which refer to intermediate states during execution; we obtain a hierarchy of notions of anytime security (Section 4). Anytime security is compositional and hence ready for a type system: using the hierarchy graph and the compositionality properties, we extract sound type-system-like criteria *automatically* (Section 5). Remarkably, among these automatically generated systems, we find carefully designed and apparently intricate security type systems from the literature.

2 The programming language

Our parallel while-language has commands c, atomic commands a, arithmetic expressions e and boolean tests b defined below, where n represents integers and x program variables:

$$\begin{array}{ll} c ::= \mathsf{skip} \mid a \mid c_1 \,;\, c_2 \mid & a ::= x := e \\ \quad\ \ \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid & e ::= n \mid x \mid e_1 + e_2 \\ \quad\ \ \mathsf{while}\ b\ \mathsf{do}\ c_1 \mid c_1 \parallel c_2 & b ::= e_1 = e_2 \end{array}$$

A *state* is a function assigning integers to variables. In each state s, arithmetic expressions e evaluate to integers written $e(s)$ and boolean tests b to boolean values. Executing atomic commands $x := e$ changes s to $s[x \mapsto e(s)]$, i. e., updates s to assign $e(s)$ to x. For example, if s assigns 1 to x and 2 to y, then $x + y$ evaluates to 3 and $x := x + y$ changes s to $s[x \mapsto 3]$.

In general, we write $s \xrightarrow{c} s' \rhd c'$ to indicate that, in state s, command c takes *one step* changing the state to s and yielding the remainder c'. The remainder indicates what remains to be executed out of c – if the remainder is skip, the program has terminated and we write $s \xrightarrow{c} s'$. For example, one step taken by the sequential composition $x := 1 \,;\, y := 2$ in state s changes s to $s[x \mapsto 1]$ and yields the remainder $y := 2$ – this is written $s \xrightarrow{x := 1;\, y := 2} s[x \mapsto 1] \rhd y := 2$. Given the parallel composition command $x := 1 \parallel y := 2$, we have two possibilities: either the left component takes the step $s \xrightarrow{x := 1 \parallel y := 2} y := 2 \rhd s[x \mapsto 1]$, or the right component takes the step $s \xrightarrow{x := 1 \parallel y := 2} x := 1 \rhd s[y \mapsto 2]$. A terminating step example is $s \xrightarrow{x := 1} s[x \mapsto 1]$. We also allow the idle step $s \xrightarrow{\mathsf{skip}} s$ – this will not trivialize the notion of terminating computation, which we define as "reaching skip."

We also write $s \xRightarrow{c} s' \rhd c'$ to indicate that, in state s, command c takes *zero or more steps* changing the state to s and yielding the remainder c'; again, if the remainder is skip, we write $s \xRightarrow{c} s'$, e. g., $s \xRightarrow{x := 1;\, y := 2} s[x \mapsto 1, y \mapsto 2]$.

In short, we consider a standard *small-step semantics*, denoted using $\rightarrow$, and its multi-step extension $\Rightarrow$.

3 End-to-end security

Henceforth, we assume the variables are partitioned into *low* variables l, l' etc., and *high* variables h, h' etc. Low variables store public, low-confidentiality data and high variables store private, high-confidentiality data. Two states s_1 and s_2 are *low-equivalent*, written $s_1 \sim_{\mathsf{low}} s_2$, if they assign the same values to low variables. The attacker has a "low view", only seeing the values of low variables – i. e., the attacker cannot distinguish between two low-equivalent states.

A command c is *end-to-end secure* if, given low-equivalent states, the results after executing c are again low-equivalent. Intuitively, the attacker should not be able to tell anything about the high part of the state by changing its low part, running the command c, and inspecting the low part of the final state. In our nondeterministic semantics, a sensible formalization of this property, denoted by $\mathsf{secure}_{\mathsf{e2e}}(c)$, is depicted in Figure 1(A): for all $s_1 \sim_{\mathsf{low}} s_2$ and all s_1' such as $s_1 \xRightarrow{c} s_1'$, there exists s_2' such as $s_2 \xRightarrow{c} s_2'$ and $s_1' \sim_{\mathsf{low}} s_2'$.

A problem with end-to-end security is its lack of compositionality w. r. t. parallel composition: even if c_1 and c_2 are both secure, $c_1 \parallel c_2$ may be unsecure. This fact is obvious from the nature of the interleaving semantics, which is sensitive to the intermediate execution states. For example, if c_1 is $l := h \,;\, l := 4$ and c_2 is $l' := l$, then during the execution of $c_1 \parallel c_2$ the secret value of h may be leaked via l to l' immediately after l has received it from h, whereas executing c_1 alone would overwrite this secret value to 4 and hence would be secure. To address this, we strengthen end-to-end security by postulating security *at any time* during execution.

4 Anytime security

We obtain anytime security by subjecting end-to-end security to a series of modifications, as depicted in Figure 1. The first modification (A to B) is the most essential one: we look not only at complete (terminated) executions of a command c, but also at incomplete ones, yielding the remainders c_1' and c_2'. The second modification (B to C) first replaces on the left execution the multi-step $\Rightarrow$ by the single step $\rightarrow$, since analyzing how one step of the left is matched by the right is sufficient for knowing how multiple steps are matched; moreover, for the right execution we replace $\Rightarrow$ by $\rightsquigarrow$, a generic arrow that can encode restrictions on the number of matching steps – we will come back to this later.

Note that (C) states something like this: c is secure if for all s_1, s_1', s_2 and c_1' such that $s_1 \sim_{\mathsf{low}} s_2$ and $s_1 \xrightarrow{c} s_1' \triangleright c_1'$, there exist s_2' and c_2' such that $s_2 \overset{c}{\rightsquigarrow} s_2' \triangleright c_2'$ and $s_1' \sim_{\mathsf{low}} s_2'$. But how about the remainders c_1' and c_2'? They should certainly not be allowed to produce insecure behavior – in fact, it is natural to require for them a property similar to the one we required for c, i. e., that they resume in a secure way the so far secure execution; and the same for their own remainders, and so on, indefinitely. This motivates the last modification (C to D), which replaces the single command c desired to be secure by two commands c_1 and c_2 desired to be "mutually secure" according to a binary relation $\approx$, with the remainders c_1' and c_2' required to also be mutually secure.

Consequently, we define *anytime security*, $\approx$, as follows: $c_1 \approx c_2$ holds if and only if, for all s_1, s_2 such that $s_1 \sim_{\mathsf{low}} s_2$:

- for all c_1' and s_1' such that $s_1 \xrightarrow{c_1} s_1' \triangleright c_1'$, there exist c_2' and s_2' such that $s_2 \overset{c_2}{\rightsquigarrow} s_2' \triangleright c_2'$, $s_1' \sim_{\mathsf{low}} s_2'$ and $c_1' \approx c_2'$
- ditto, with (c_1, s_1) and (c_2, s_2) swapped

(In particular, we have $\mathsf{skip} \approx \mathsf{skip}$.) There is an apparent circularity in this definition, which explains $c_1 \approx c_2$ in terms of $c_1' \approx c_2'$. This is resolved by taking the *greatest fixed point* view of this equation, meaning that $\approx$ should hold for c_1 and c_2, and for their continuations c_1' and c_2', and for the continuations of these, etc. A command c is called $\approx$-secure if $c \approx c$ holds.

The bisimilarity-like relation $\approx$ is parameterized by the choice of $\rightsquigarrow$. Choosing particular $\rightsquigarrow$ relations leads to particular $\approx$ relations:

- if $\rightsquigarrow$ is $\rightarrow$, we obtain *strong bisimilarity*, denoted $\approx_{\mathsf{S}}$: one step needs to be matched by precisely one step
- if $\rightsquigarrow$ is $\Rightarrow$, we obtain *weak bisimilarity*, denoted $\approx_{\mathsf{W}}$: one step needs to be matched by zero or more steps
- if $\rightsquigarrow$ is "$\rightarrow$ or identity", we obtain $\approx_{01}$, an intermediate between $\approx_{\mathsf{S}}$ and $\approx_{\mathsf{W}}$ which we call 01-*bisimilarity*: one step matched by zero or one steps

A natural question arising here concerns termination: should the relation $\approx$ be further required to be *termination-sensitive*, i. e., in Figure 1(D), should the remainders c_1' and c_2' be required to be skip at the same time (meaning that the indicated steps or sequences of steps from c_1 and c_2 are either both terminating or both not terminating)? We do not commit to a yes-or-no answer, but factor in both possibilities. Whereas $\approx_{\mathsf{S}}$ is termination-sensitive by definition, adding the termination-sensitiveness to $\approx_{\mathsf{W}}$ and $\approx_{01}$ yields two new relations $\approx_{\mathsf{WT}}$ and $\approx_{01\mathsf{T}}$. Finally, we consider some degenerate strong notions of security. A command c is called:

- *self-isomorphic*, written $\mathsf{siso}\ (c)$, if it is strongly bisimilar with itself while keeping the remainders identical (as in Figure 1(D), except that $c_1 = c_2$ and $c_1' = c_2'$); intuitively, this means that not only the low-observable behavior, but also the program counter does not depend on the high part of the state
- *high*, written $\mathsf{high}\ (c)$, if its execution never updates a low variable – so no interesting observable behavior

Convention 1. If χ is one of the above binary relations, then $\chi(c)$ denotes its unary version (which gives the corre-

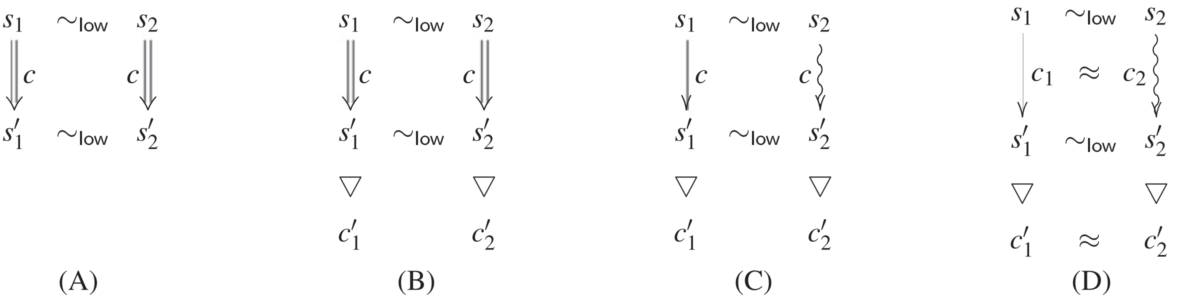

Figure 1: From end-to-end security to anytime security.

$$\begin{tikzcd}
 & \approx_{\mathsf{W}}(c) \\
\approx_{\mathsf{01}}(c) \arrow[ur] & \approx_{\mathsf{WT}}(c) \arrow[u] \\
 & \approx_{\mathsf{01T}}(c) \arrow[u] \arrow[ul] \\
\mathsf{high}(c) \arrow[uu] & \approx_{\mathsf{S}}(c) \arrow[u] \\
\mathsf{high}(c) \wedge \mathsf{term}(c) \arrow[u] \arrow[uuur] & \mathsf{siso}(c) \arrow[u]
\end{tikzcd}$$

c	$\mathsf{term}(c)$	$\mathsf{high}(c)$	$\varphi(c)$	$\psi(c)$
a	True	$\mathsf{highAtom}(a)$	$\mathsf{nonleak}(a)$	$\mathsf{nonleak}(a)$
$c_1\ ;\ c_2$	$\mathsf{term}(c_1)$ $\mathsf{term}(c_2)$	$\mathsf{high}(c_1)$ $\mathsf{high}(c_2)$	$\varphi(c_1)$ $\varphi(c_2)$	$\psi_{\mathsf{T}}(c_1)$ $\psi(c_2)$ ――― $\psi(c_1)$ $\mathsf{high}(c_2)$
if b then c_1 else c_2	$\mathsf{term}(c_1)$ $\mathsf{term}(c_2)$	$\mathsf{high}(c_1)$ $\mathsf{high}(c_2)$	$\mathsf{low}(b)$ $\varphi(c_1)$ $\varphi(c_2)$	$\mathsf{low}(b)$ $\psi(c_1)$ $\psi(c_2)$
while b do c_1	False	$\mathsf{high}(c_1)$	$\mathsf{low}(b)$ $\varphi(c_1)$	False
$c_1 \parallel c_2$	$\mathsf{term}(c_1)$ $\mathsf{term}(c_2)$	$\mathsf{high}(c_1)$ $\mathsf{high}(c_2)$	$\varphi(c_1)$ $\varphi(c_2)$	$\psi(c_1)$ $\psi(c_2)$

$$\varphi \in \{\mathsf{siso}, \approx_{\mathsf{S}}, \approx_{\mathsf{01T}}, \approx_{\mathsf{WT}}\} \qquad \psi \in \{\approx_{\mathsf{01}}, \approx_{\mathsf{W}}\} \qquad \psi_{\mathsf{T}} \equiv \begin{cases} \approx_{\mathsf{01T}}, & \text{if } \psi = \approx_{\mathsf{01}} \\ \approx_{\mathsf{WT}}, & \text{if } \psi = \approx_{\mathsf{W}} \end{cases}$$

Figure 2: Hierarchy and compositionality for anytime security.

sponding notion of c being secure) – e. g., $\approx_{\mathsf{S}}(c)$ denotes $c \ \approx_{\mathsf{S}}\ c$.

Recall that our intermediate goal was to strengthen end-to-end security in order to solve the $\parallel$ compositionality issue. Our notions of anytime security routinely achieve this goal, provided for the termination-insensitive notions we additionally assume termination of the program:

Proposition 1. Assume one of the following holds:

- χ is any of $\mathsf{siso}, \approx_{\mathsf{S}}, \approx_{\mathsf{WT}}, \approx_{\mathsf{01T}}$
- χ is any of high, $\approx_{\mathsf{W}}$, $\approx_{\mathsf{01}}$ and all executions of c (starting from any state) are terminating

Then $\chi(c)$ implies $\mathsf{secure}_{\mathsf{e2e}}(c)$.

We are ready to engage in the second part of our program: establish, for the anytime security notions, their hierarchy and their (partial, conditional) compositionality properties w. r. t. *all* language constructs, not only $\parallel$.

All these are depicted in Figure 2. In the graph, the arrows represent inclusions and $\mathsf{term}(c)$ states an interactive form of termination for c (in a possibly changing environment), formally: for all s, s', s'' and c' such that $s \overset{c}{\Rightarrow} s' \rhd c'$, there exists a terminating execution of c' starting from s''. Since $\mathsf{high}(c)$ implies $\approx_{\mathsf{WT}}(c)$ only for interactively terminating c, we include the predicate term in the graph. Note that all commands not containing while loops satisfy term.

The table contains compositionality properties of these notions w. r. t. the language constructs. The first column lists the possible forms of a command c: it may be an atomic command a, a sequential composition $c_1\ ;\ c_2$, etc. The next columns list conditions under which the predicates stated on the first row hold for c. For example, row 3 column 3 says: if $\mathsf{high}(c_1)$ and $\mathsf{high}(c_2)$ then $\mathsf{high}(c_1\ ;\ c_2)$. The conditions on the atomic commands are as follows: $x := e$ is *high* if x is high and *non-leaking* if there exists no variable in e which is strictly higher than x – i. e., if x is low, then no variable in e is high. A boolean test $e_1 = e_2$ is *low* if e_1 and e_2 contain only low variables. The horizontal line in row 3 column 5 represents a disjunction. Each table property was produced independently from the others, by confronting a given program construct with a given security notion.

Proposition 2. The implications and the compositionality facts from Figure 2 hold.

5 Syntactic criteria

We would like to stress that it is *not* crucial that the reader followed our definitions of the anytime security notions that inhabit Figure 2's graph, and in fact we shall never again need to refer to these definitions – what should be retained is that they were naturally produced as strengthenings of end-to-end security with the purpose of obtaining compositionality w. r. t. $\parallel$ (the last row in Figure 2's table).

Now we are ready to generate the type-system-like syntactic criteria. They simply follow by trying to prove security of a command *using only Figure 2's graph and table*. To show that c is secure according to some anytime security notion χ, we first try to reduce the goal to proving χ for the components of c; if this is impossible due to the failure of the side-condition in the table's entry for χ and the top construct of c, we move downward on the graph and try the proof for c and one of χ's predecessors (which is hopefully more compositional w. r. t. c's top construct). And the procedure continues recursively, exploring the predecessors depth-first.

For example, here is a table-and-graph proof for $\approx_{01}(c)$ where c is $l := 4\ ;\ \mathsf{if}\ h = 0\ \mathsf{then}\ h := 1\ \mathsf{else}\ h := 2$:

- We start positioning ourselves at $\approx_{01}$ in the graph. Since c has construct ";" at the top, we look in the table at $\approx_{01}$ versus ";" – the decomposition condition is a disjunction, and we choose the upper disjunct.
- The property we need to check for the left component of c, namely $l := 4$, is $\approx_{01T}$. Looking in the table at $\approx_{01T}$ versus atomic statements, we need to check that $l := 4$ is nonleaking, which is true.
- It remains to check $\approx_{01}$ for the right component of c, namely $\mathsf{if}\ h = 0\ \mathsf{then}\ h := 1\ \mathsf{else}\ h := 2$. Since, the required side-condition in the table's entry for $\approx_{01}$ versus if, namely $\mathsf{low}\ (h = 0)$, fails, we turn to the predecessors of $\approx_{01}$ from the graph, namely high and $\approx_{01T}$, from which we choose high.
- We restart the table-and-graph procedure, this time for high and $\mathsf{if}\ h = 0\ \mathsf{then}\ h := 1\ \mathsf{else}\ h := 2$. The proof now succeeds completely by the table.

Note that we appeal to the graph whenever the table result is not sufficiently flexible. Now, a syntactic criterion $\overline{\chi}$ emerges for each semantic notion χ by simply turning the above procedure into recursive function definitions. For example, the following recursive equations correspond to the semantic facts used in our sample table-and-graph proof:

$$\overline{\approx_{01}}\,(c_1\ ;\ c_2) \quad = \quad \begin{array}{l}(\overline{\approx_{01T}}(c_1)\ \wedge\ \overline{\approx_{01}}(c_2)) \vee \\ (\overline{\approx_{01}}(c_1)\ \wedge\ \overline{\mathsf{high}}\,(c_2))\end{array}$$

$$\overline{\approx_{01}}\,(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \quad = \quad \begin{cases}\overline{\approx_{01}}(c_1)\ \wedge\ \overline{\approx_{01}}(c_2), \\ \quad \text{if } \mathsf{low}\,(b) \\ \overline{\mathsf{high}}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \vee \\ \overline{\approx_{01T}}\,(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2), \\ \quad \text{otherwise}\end{cases}$$

$$\overline{\approx_{01T}}\,(a) \quad = \mathsf{nonleak}\ (a)$$

$$\overline{\mathsf{high}}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) = (\overline{\mathsf{high}}\,(c_1)\ \wedge\ \overline{\mathsf{high}}\,(c_2))$$

$$\overline{\mathsf{high}}\ (a) \quad = \mathsf{highAtom}\ (a)$$

In the first equation, the disjunction reflects the corresponding disjunction from the table's entry for $\approx_{01}$ versus ";". In the second equation, we see that in case the table side-condition (here $\mathsf{low}\ (b)$) fails, we turn to the graph – the disjunction emerges here from the existence of two graph predecessors of $\approx_{01}$, namely high and $\approx_{01T}$.

The soundness of the $\overline{\chi}$'s follow immediately by mutual induction, using χ's hierarchy and compositionality:

Theorem 1. The syntactic criteria $\overline{\chi}$ are sound for the security notions χ in Figure 2, in that $\overline{\chi}\,(c)$ implies $\chi\,(c)$. A fortiori, $\overline{\chi}$ are sound for end-to-end security of terminating programs.

Type systems from the literature. $\overline{\mathsf{siso}}$ corresponds to a type system from Smith and Volpano [14] for scheduler independent security – this criterion is extremely harsh, forbidding high tests at if and while. $\overline{\approx_{WT}}$ corresponds to another type system [14], where high tests are allowed at if provided the branches are high, but are disallowed at while. This harsh condition on while is the starting point of work by Boudol and Castellani [4], where a type system equivalent to $\overline{\approx_{01}}$ is introduced. $\overline{\approx_{01}}$ allows high tests for while provided the body of the while is high. This is possible because, unlike $\overline{\approx_{WT}}$, $\overline{\approx_{01}}$ can fall back on $\overline{\mathsf{high}}$. However, the price for this is a harsher clause for ";" (which is a limitation of the termination-insensitive notions). An improvement of $\overline{\approx_{01}}$ is proposed by Boudol [3], where, in the c_1 part of $c_1\ ;\ c_2$, one no longer restricts to low tests everywhere, but rather only in places that may affect termination (i. e., inside while loops). Interestingly, this condition on c_1 is the one imposed by $\overline{\approx_{WT}}$, and therefore Boudol's approach can be seen as a carefully designed combination of $\overline{\approx_{WT}}$ and $\overline{\approx_{01}}$ – it is in fact equivalent to $\overline{\approx_{W}}$.

Acknowledgement: This work was supported by the project Ni 491/13–2, part of the DFG priority program Reliably Secure Software Systems (RS^3). Dmitriy Traytel, Jasmin Blanchette and the reviewers made very useful suggestions to improve the presentation and corrected some errors.

References

1. Reliably secure software systems (RS^3). http://www.reliably-secure-software-systems.de, 2013.
2. Security type systems and deduction – RS^3 project. http://www21.in.tum.de/local_projects/rs3.html, 2013.
3. G. Boudol. On typing information flow. In *ICTAC*, pages 366–380, 2005.
4. G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP*, pages 382–395, 2001.
5. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
6. M. Keil and P. Thiemann. Type-based dependency analysis for javascript. In *PLAS*, pages 47–58, 2013.

7. A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
8. A. Popescu, J. Hölzl, and T. Nipkow. Proving concurrent noninterference. In *CPP*, pages 109–125, 2012.
9. A. Popescu, J. Hölzl, and T. Nipkow. Formal verification of language-based concurrent noninterference. *Journal of Formalized Reasoning*, 6(1), 2013.
10. A. Popescu, J. Hölzl, and T. Nipkow. Formalizing probabilistic noninterference. In *CPP*, pages 259–275, 2013.
11. A. Sabelfeld and H. Mantel. Securing communication in a concurrent language. In *SAS*, pages 376–394, 2002.
12. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
13. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE Computer Security Foundations Workshop*, pages 200–214, 1999.
14. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364, 1998.
15. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
16. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

Bionotes

Prof. Tobias Nipkow, Ph.D.
Fakultät für Informatik, Technische
Universität München, Boltzmannstr. 3,
85748 Garching, Germany,
Tel.: +49-89-289-17302,
Fax: +49-89-289-17301
nipkow@in.tum.de

Tobias Nipkow received his Diplom in Informatik (MSc in Computer Science) from the Technische Hochschule Darmstadt in 1982 and a PhD in Computer Science from The University of Manchester in 1987. He held post-doc positions at MIT and Cambridge University before becoming a professor at the Technische Universität München in 1992. He has worked on term rewriting, programming language semantics and theorem proving. For more than 20 years, Tobias Nipkow and his research group in Munich (jointly with Lawrence Paulson in Cambridge and Makarius Wenzel in Paris) have been developing the popular proof assistant Isabelle.

Dr. Andrei Popescu
Fakultät für Informatik, Technische
Universität München, Boltzmannstr. 3,
85748 Garching, Germany,
Tel.: +49-173-2609466,
Fax: +49-89-289-17301
uuomul@yahoo.com

Andrei Popescu received his BA in Computer Science from the University of Bucharest in 2001, a PhD in Mathematics from the same university in 2005, and a PhD in Computer Science from the University of Illinois at Urbana-Champaign in 2010. From 2010, he is working as a post-doc at the Technische Universität München. His main research interests are mechanical verification, type systems, category theory, information-flow security, and intersections of these areas.