“Logic Wins!”

Jean Goubault-Larrecq*

LSV, ENS Cachan, CNRS, INRIA Saclay
ENS Cachan, 61, avenue du président Wilson, 94230 Cachan, France
goubault@lsv.ens-cachan.fr

Abstract. Clever algorithm design is sometimes superseded by simple encodings into logic. We apply this motto to a few case studies in the formal verification of security properties. In particular, we examine confidentiality objectives in hardware circuit descriptions written in VHDL.

1 Introduction

As a computer scientist, I tend to be fond of clever, efficient algorithmic solutions to any particular problem I may have. Probably like many computer scientists, I have long thought that the key to success was *clever algorithm design*, together with well-crafted data structures, clever implementation techniques and some hack power. I certainly did not believe seriously that elegant semantic foundations, nice encodings into logic, or similar mathematically enjoyable concerns could actually help *in practice*, although I took some delight in these as well. Over the years, I came to realize I was wrong[1], and I will illustrate this on a few examples. In all these examples, my concern will be to find algorithms to verify security properties of protocols, and logic will be instrumental in finding elegant and efficient solutions.

Now, in this paper, by logic I will mean fragments of first-order logic expressible as finite sets of Horn clauses. I will also concentrate on security properties, and in fact on abstract interpretation frameworks for security properties. One sometimes loses precision in abstract interpretation, and I will take this for granted: I won’t use logic to solve

* Partially funded by RNTL project Prouvé.

[1] An anecdote to explain the title. In 1996, I realized that binary decision diagrams (BDDs), an extremely efficient way of handling Boolean functions [4], could be used to provide a basis for tableaux-style proof search in *non-classical* (modal, intuitionistic) logics, only orders of magnitude faster than previous implementations [13]. I needed to compare my algorithm to other implementations. Since there were not too many non-classical provers available at that time, I compared my clever implementation, specialized to the case of the run-of-the-mill system **LJ** for intuitionistic propositional logic, to a quick, naive implementation of proof-search in Roy Dyckhoff’s contraction-free sequent calculus **LJT** for intuitionistic logic [10]. Now **LJT** is only meant to avoid a rather painful check for loops during proof search that is inherent to **LJ**, but is otherwise not intended to be a basis for efficient implementations. Despite this, my naive implementation of **LJT** beat my sophisticated, state-of-the-art BDD-based implementation of **LJ** flat-handed. This has taught me a lesson, which I have been meditating over ever since. When I told this to Roy Dyckhoff at the Tableaux conference in 1996, his reaction was simply “Logic wins!”, in a deep voice, and with clear pleasure in his eyes.

security problems exactly but to obtain reasonably precise, terminating algorithms. One may say that my motto is the infamous 80-20 rule: do 80% of the work with only 20% of the effort. Logic will be crucial to reach this goal.

After some preliminaries, I'll show how this motto helps us analyze weak secrecy and correspondence assertions in the spi-calculus, expanding on work by Nielson, Nielson and Seidl [19] (Section 2). I'll then comment on generic abstraction algorithms in Section 3, and proceed to something new in Section 4: verifying confidentiality objectives in hardware circuit descriptions written in VHDL. I'll conclude by proposing some open problems in Section 5.

Acknowledgments. We thank David Lubicz, Nicolas Guillermin, and Riccardo Bresciani for fruitful interaction on the question of static analysis of VHDL for security.

Preliminaries. We shall consider terms s, t, u, v, ..., over a fixed, usually implicit, finite signature. We assume finitely many predicate symbols p, q, ..., and countably many variables X, Y, Z, ... *Atoms* are expressions of the form $p(t)$. Notice that all our predicates are unary. This incurs no loss of generality, as e.g., $p(t, u)$ is easily encoded as $p(c(t, u))$ for some fresh binary function symbol c.

Substitutions σ are finite maps from variables to terms, e.g., $[X_1 := t_1, \ldots, X_n := t_n]$, and substitution application $t\sigma$ works in parallel, i.e., $X_i\sigma = t_i$, $X\sigma = X$ if $X \notin \{X_1, \ldots, X_n\}$, $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

Horn clauses C are of the form $H \Leftarrow \mathcal{B}$ where the *head* H is either an atom $p(t)$ or $\bot$, and the *body* $\mathcal{B}$ is a finite set $A_1, \ldots, A_n$ of atoms. The meaning is that C is true iff H is true whenever all of A_1, ..., A_n are. If $\mathcal{B}$ is empty ($n = 0$), then $C = H$ is a *fact*.

We always assume that there is at least one constant, i.e., one function symbol of arity 0, so that the set of ground atoms is non-empty. An atom or a term is *ground* iff it contains no variable. A *Herbrand model* I of a set of clauses is a set of ground atoms; intuitively, those that we want to consider as true. Any satisfiable set S of Horn clauses has a least Herbrand model, in the sense that it is a Herbrand model, and is contained in every other Herbrand model of S. This can be defined as the least fixpoint $\mathrm{lfp}\, T_S$ of the monotone operator $T_S(I) = \{A\sigma \mid A \Leftarrow A_1, \ldots, A_n \in S, A\sigma \text{ ground}, A_1\sigma \in I, \ldots, A_n\sigma \in I\}$. If $\bot \in \mathrm{lfp}\, T_S$, then S is unsatisfiable. Otherwise, S is satisfiable, and $\mathrm{lfp}\, T_S$ is a set of ground atoms, which happens to be the least Herbrand model of S. Note that $A \in \mathrm{lfp}\, T_S$ iff A is *deducible* by finitely many applications of clauses of S, seen as rules allowing one to deduce heads from the corresponding bodies.

Given a finite set S of Horn clauses, it is undecidable whether S is satisfiable, i.e., whether $\bot \notin \mathrm{lfp}\, T_S$, even in very constrained cases [8]. However, some specific formats of Horn clauses *are* decidable. One that I find most remarkable is the $\mathcal{H}_1$ class, which was identified as such by Nielson, Nielson and Seidl [19], but had been introduced with a different name by Weidenbach [21]. Using the presentation of [15], $\mathcal{H}_1$ clauses are Horn clauses whose head is restricted to be of the form $\bot$, $p(X)$ for some variable X, or $p(f(X_1, \ldots, X_n))$ for pairwise distinct variables X_1, ..., X_n ($f(X_1, \ldots, X_n)$ is then called a *flat* term). Note that bodies of $\mathcal{H}_1$ clauses are not restricted in any way. Deciding $\mathcal{H}_1$ clause sets is decidable, and EXPTIME-complete [19, 15]. The `h1` tool in the `h1` tool suite [14] is an efficient implementation of the resolution algorithm of [15].

We shall exert considerable freedom in writing formulae. For example, we shall use predicates of the form $P \vdash e \simeq \_$, where the hole $\_$ is meant to denote the missing argument, and we shall write $[P \vdash e \simeq t]$ instead of $P \vdash e \simeq \_\,(t)$.

Let us quickly turn to abstract interpretation. All extensional and non-trivial properties of programs are undecidable, by Rice's theorem. The reachability problem in the more constrained world of cryptographic protocols is also undecidable [9]. In verifying such properties, one must therefore choose between correct and complete procedures that will fail to terminate on some inputs, or correct, terminating algorithms that cannot be complete. The latter strand is the tradition in the so-called *abstract interpretation* community. We also speak of *over-approximation*, because the set of inputs that are accepted by an abstract interpretation algorithm is only guaranteed to be a superset of those satisfying the intended property.

2 Reachability and Correspondence Assertions in the Spi-Calculus

Nielson *et al.* [19] introduced $\mathcal{H}_1$ as a convenient tool in deciding reachability in (an over-approximated semantics of) the spi-calculus [1]. As such, they literally applied the "Logic wins!" motto, going through logic instead of defining an ad hoc algorithm.

Reachability questions include so-called *weak secrecy* questions, since the message M remains secret in the protocol P iff $P|DY(I_0, c)$ does not rewrite in any finite number of steps to a process where M is sent over channel c, where $DY(I_0, c)$ is a *Dolev-Yao observer*, see below. We shall provide a slightly more precise over-approximation, and also deal with authentication properties, inter alia.

The main trick is to express a few relatively trivial facts about spi-calculus processes, in the guise of Horn clauses. Then, we pay special attention to the form of these clauses, so that they fall into the $\mathcal{H}_1$ class.

$P, Q, R, \ldots ::=$	$\texttt{stop}$	stop
	$!_x P$	replication
	$P \mid Q$	parallel composition
	$\nu x; P$	fresh name creation
	$\texttt{out}(e_1, e_2); P$	writing to a channel
	$\texttt{in}(e_1, x); P$	reading from a channel
	$\texttt{let}\ x = e\ \texttt{in}\ P$	local definition
	$\texttt{case}\ e_1\ \texttt{of}\ f(x_1, \ldots, x_n) \Rightarrow P\ \texttt{else}\ Q$	constructor pattern-matching
	$\texttt{case}\ e_1\ \texttt{of}\ \{x\}_{e_2} \Rightarrow P\ \texttt{else}\ Q$	symmetric decryption
	$\texttt{case}\ e_1\ \texttt{of}\ [x]_{e_2^{-1}} \Rightarrow P\ \texttt{else}\ Q$	asymmetric decryption
	$\texttt{if}\ e_1 = e_2\ \texttt{then}\ P\ \texttt{else}\ Q$	equality test
	$\texttt{event}\ f\langle e_1 \rangle; P$	event

Fig. 1. The spi-calculus

Expressions e in the spi-calculus are defined as variables x (distinct from the logical variables X), constructor applications $f(e_1, \ldots, e_n)$, symmetric encryptions $\{e_1\}_{e_2}$

and asymmetric encryptions $[e_1]_{e_2}$, where e_2 serves as key in the latter two forms. *Processes* are described in Figure 1. Note that decryption is handled through pattern-matching. In the case of constructor pattern-matching, $x_1, \ldots, x_n$ are pairwise distinct. Replication $!_x P$ launches several copies of P in parallel, keeping a unique integer id of each in variable x; we write $!P$ when x is irrelevant. We also write $\mathtt{out}(e_1, e_2)$ instead of $\mathtt{out}(e_1, e_2); \mathtt{stop}$ and similarly for other actions. Finally, events $\mathtt{event}\ f\langle e_1 \rangle$ are meant to express correspondence assertions, i.e., certain forms of authentication.

The semantics of this language is standard: see [1]. The essential rules are the communication rule $\mathtt{out}(e_1, e_2); P \mid \mathtt{in}(e_1, x); Q \rightarrow (P \mid Q[x := e_2])$, the fact that $P \equiv \rightarrow \equiv Q$ implies $P \rightarrow Q$, where structural congruence $\equiv$ obeys some obvious laws, plus the extrusion law $(\nu x; P) \mid Q \equiv \nu x; (P \mid Q)$ if x is not free in Q.

The Dolev-Yao observer $DY(I_0, c)$ mentioned above, for example, is $\nu id; !A$, where A is the parallel composition of $\mathtt{out}(c, I_0)$ (the Dolev-Yao attacker can emit the *initial knowledge* expression I_0 on channel c), $\mathtt{out}(c, id)$ (it can emit its own identity), $\mathtt{in}(c, x); \mathtt{out}(c, x); \mathtt{out}(c, x)$ (it can duplicate messages), $\nu N; \mathtt{out}(c, N)$ (emit fresh names), $\mathtt{in}(c, x_1); \mathtt{in}(c, x_2); \mathtt{out}(c, \{x_1\}_{x_2})$ and $\mathtt{in}(c, x_1); \mathtt{in}(c, x_2); \mathtt{out}(c, [x_1]_{x_2})$ (encrypt), $\mathtt{in}(c, x); \mathtt{in}(c, x_2); \mathtt{case}\ x\ \mathtt{of}\ \{x_1\}_{x_2} \Rightarrow \mathtt{out}(c, x_1)\ \mathtt{else\ stop}$ and $\mathtt{in}(c, x); \mathtt{in}(c, x_2); \mathtt{case}\ x\ \mathtt{of}\ [x_1]_{x_2^{-1}} \Rightarrow \mathtt{out}(c, x_1)\ \mathtt{else\ stop}$ (decrypt), and various processes of the form $\mathtt{in}(c, x_1); \ldots; \mathtt{in}(c, x_n); \mathtt{out}(c, f(x_1, \ldots, x_n))$ or $\mathtt{in}(c, x); \mathtt{case}\ x\ \mathtt{of}\ f(x_1, \ldots, x_n) \Rightarrow (\mathtt{out}(c, x_1) | \ldots | \mathtt{out}(c, x_n))\ \mathtt{else\ stop}$, depending whether f is a constructor or a function, and whether it is private or public in the terminology of ProVerif [3].

Fix a spi-calculus process P_0. We define an approximate semantics, specialized to P_0, as follows. First, for every subprocess P of P_0, and every list of variables Ξ, meant to denote the list of variables bound above P in P_0 (except by ν, which will be dealt with differently), we collect all pairs $(Q; \Xi')$ where Q is a subprocess of P, and Ξ' is the list of variables bound above Q in P, in a set $Sub_\Xi(P)$. Formally, let $Sub_\Xi(P) = \{(P; \Xi)\} \cup Sub^+_\Xi(P)$, where $Sub^+_\Xi(\mathtt{stop}) = \emptyset$, $Sub^+_\Xi(!_x P) = Sub_{x,\Xi}(P)$, $Sub^+_\Xi(\mathtt{out}(e_1, e_2); P) = Sub^+_\Xi(\mathtt{event}\ f\langle e_1 \rangle; P) = Sub_\Xi(P)$, $Sub^+_\Xi(\mathtt{in}(e_1, x); P) = Sub_{x,\Xi}(P)$, $Sub^+_\Xi(P|Q) = Sub^+_\Xi(\mathtt{if}\ e_1 = e_2\ \mathtt{then}\ P\ \mathtt{else}\ Q) = Sub_\Xi(P) \cup Sub_\Xi(Q)$, $Sub^+_\Xi(\mathtt{let}\ x = e\ \mathtt{in}\ P) = Sub_{x,\Xi}(P)$, $Sub^+_\Xi(\mathtt{case}\ e_1\ \mathtt{of}\ pat \Rightarrow P\ \mathtt{else}\ Q) = Sub_{\Xi',\Xi}(P) \cup Sub_\Xi(Q)$ (where Ξ' is $x_1, \ldots, x_n$ in constructor pattern-matching, and x in the cases of decryption); finally, if $\Xi = x_1, \ldots, x_k$, we let $Sub^+_\Xi(\nu x; P) = Sub_\Xi(P[x := \ulcorner \nu x; P \urcorner(x_1, \ldots, x_k)])$, where $\ulcorner \nu x; P \urcorner$ is a (fresh) function symbol, one for each process starting with a name creation action. This is Blanchet's fresh name creation as skolemization trick [3].

We shall also need to represent *contexts* ρ, i.e., finite mappings $[x_1 \mapsto t_1, \ldots, x_k \rightarrow t_k]$ from variables to terms, as terms. We choose to represent such mappings as $c_\Xi(t_1, \ldots, t_k)$, where $\Xi = x_1, \ldots, x_k$ and c_Ξ is a fresh function symbol of arity k.

Without loss of generality, we shall assume that P_0 is *well-formed*, in the intuitive sense that the only expressions e that occur in P_0 are either spi-calculus variables x bound by !, `in`, `let`, or `case` (but not by ν), or are to the right of an equals sign $=$ in a `let`-expression. This is easily achieved by adding extra `let` constructs in P_0. Formally, we require that whenever $(Q; \Xi') \in Sub_\emptyset(P_0)$, then Q is given by the grammar obtained from Figure 1 by requiring e_1 and e_2, wherever they appear, to be variables

x, y; moreover, x and y must be in Ξ' and $x \neq y$. We also require that whenever $(\texttt{let}\ x = e\ \texttt{in}\ Q; \Xi') \in Sub_\emptyset(P_0)$, then all the variables in e must occur in Ξ'.

Our approximate semantics will use the following predicate symbols. First, $\rightarrow^* P\langle\!\langle \_ \rangle\!\rangle$, for each $(P; \Xi) \in Sub_\emptyset(P_0)$: $[\rightarrow^* P\langle\!\langle \rho \rangle\!\rangle]$ states that execution may reach P, with values of variables given by context ρ; second, $\_ \triangleright \_$: $[t \triangleright u]$ means that message u was sent on channel t; $\texttt{event}\ f\langle \_ \rangle$: $\texttt{event}\ f\langle t \rangle$ says that we have passed the corresponding event in P_0; and auxiliary predicates $\_ \in \mathbb{N}$ ($[t \in \mathbb{N}]$ means that t denotes an integer process id), and for each expression e in P_0, a predicate symbol $P \vdash e \simeq \_$ ($[P \vdash e \simeq t]$ means that expression e may have value t when we reach P in P_0).

We now compile P_0 to a set $S(P_0)$ of $\mathcal{H}_1$ clauses. First, execution starts at P_0: write the fact $[\rightarrow^* \texttt{main}\langle\!\langle c_\epsilon() \rangle\!\rangle]$, where ϵ is the empty sequence, so that $c_\epsilon()$ is the empty context. Then, for each $(P; \Xi) \in Sub_\emptyset(P_0)$, do a case analysis on P. If P is a replication $!_x P_1$, then create a fresh process identifier X and proceed to P_1:

$$[\rightarrow^* P_1\langle\!\langle c_{x,\Xi}(X, X_1, \ldots, X_k) \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle], [X \in \mathbb{N}] \quad (1)$$

$$[\texttt{0} \in \mathbb{N}] \qquad [\texttt{s}(X) \in \mathbb{N}] \Leftarrow [X \in \mathbb{N}] \quad (2)$$

If P is a parallel composition $P_1 \mid Q_1$, then state that from P, execution may proceed to P_1 or to Q_1, as in [19], while the context, denoted by Z, does not change:

$$[\rightarrow^* P_1\langle\!\langle Z \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle Z \rangle\!\rangle] \qquad [\rightarrow^* Q_1\langle\!\langle Z \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle Z \rangle\!\rangle] \quad (3)$$

If $P = \nu x; P_1$, then use Blanchet's skolemization trick. Write $\Xi = x_1, \ldots, x_k$, and output the clause:

$$[\rightarrow^* P_1[x := \ulcorner \nu x; P_1 \urcorner(x_1, \ldots, x_k)]\langle\!\langle Z \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle Z \rangle\!\rangle] \quad (4)$$

At this point, it is probably good to realize why these are $\mathcal{H}_1$ clauses. The head of (4) is just one big predicate symbol $\rightarrow^* P_1[x := \ulcorner \nu x; P_1 \urcorner(x_1, \ldots, x_k)]\langle\!\langle \_ \rangle\!\rangle$ applied to the logical variable Z, for instance. The spi-calculus variables x_1, ..., x_k do not serve as logical variables, and are only part of the *name* of the predicate. Similarly, the head of (1) is the predicate $\rightarrow^* P_1\langle\!\langle \_ \rangle\!\rangle$ applied to the flat term $c_{x,\Xi}(X, X_1, \ldots, X_k)$.

Let us return to the subprocesses P of P_0. When P is an $\texttt{out}$ command, remember that P_0 is well-formed, so P must be of the form $\texttt{out}(x_i, x_j); P_1$, where $\Xi = x_1, \ldots, x_k$ and $1 \leq i \neq j \leq k$. We then write one clause (5) to state that the value X_j of x_j has now been sent on (the value X_i of) the channel x_i, and another one (6) to state that execution should proceed to P_1:

$$[X_i \triangleright X_j] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle] \quad (5)$$

$$[\rightarrow^* P_1\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle] \quad (6)$$

The case of events, i.e., when $P = \texttt{event}\ f\langle x_i \rangle; P_1$, is very similar:

$$\texttt{event}\ f\langle X_i \rangle \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle] \quad (7)$$

When $P = \texttt{in}(x_i, x); P_1$, where again $\Xi = x_1, \ldots, x_k$ and $1 \leq i \leq k$, we write one clause stating that one may proceed to P_1 after binding x to any value X found on (the value X_i of) the channel x_i:

$$[\rightarrow^* P_1\langle\!\langle c_{x,\Xi}(X, X_1, \ldots, X_k) \rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k) \rangle\!\rangle], [X_i \triangleright X] \quad (8)$$

When $P = \texttt{let}\ x = e\ \texttt{in}\ P_1$, we produce:

$$[\rightarrow^* P_1\langle\!\langle c_{x,\Xi}(X, X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle], [P \vdash e \simeq X] \quad (9)$$

where we define the evaluation of expressions e, given a list of bound variables $\Xi = x_1, \ldots, x_k$, as follows. Clause (10) below states that x_i equals X_i in any context $[x_1 \mapsto X_1, \ldots, x_k \mapsto X_k]$ (represented as the term $c_\Xi(X_1, \ldots, X_k)$). Names are evaluated in clause (11), where we observe that the arguments to $\ulcorner \nu x; Q \urcorner$ must be a suffix $x_i, \ldots, x_k$ of Ξ. Clause (12) deals with subexpressions where a function symbol is applied to pairwise distinct variables (a special case where we can still write an $\mathcal{H}_1$ clause). Finally, clause (13) deals with the remaining cases.

$$[P \vdash x_i \simeq X_i] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle] \quad (10)$$

$$[P \vdash \ulcorner \nu x; Q \urcorner(x_i, \ldots, x_k) \quad (11)$$
$$\simeq \ulcorner \nu x; Q \urcorner(X_i, \ldots, X_k)] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle]$$

$$[P \vdash f(x_{i_1}, \ldots, x_{i_n}) \simeq f(X_{i_1}, \ldots, X_{i_n})] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle] \quad (12)$$
$$(i_1, \ldots, i_j \text{ pairwise distinct})$$

$$[P \vdash f(e_1, \ldots, e_n) \simeq f(X_1, \ldots, X_n)] \Leftarrow [P \vdash e_1 \simeq X_1], \ldots, [P \vdash e_n \simeq X_n] \quad (13)$$
$$(e_1, \ldots, e_n \text{ not pairwise distinct variables})$$

When $P = \texttt{case}\ x_i\ \texttt{of}\ f(y_1, \ldots, y_n) \Rightarrow P_1\ \texttt{else}\ Q_1$, with $\Xi = x_1, \ldots, x_k$ and $1 \leq i \leq k$, we write:

$$[\rightarrow^* P_1\langle\!\langle c_{y_1,\ldots,y_n,\Xi}(Y_1, \ldots, Y_n, X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow \quad (14)$$
$$[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_i, \ldots X_k)\rangle\!\rangle], [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, \underbrace{f(Y_1, \ldots, Y_n)}_{i}, \ldots X_k)\rangle\!\rangle]$$

to handle the case of a match. In the second premise, the argument $f(Y_1, \ldots, Y_n)$ occurs at argument position i, in lieu of X_i; we have made this explicit with an underbrace. Our intent was really to write the clause:

$$[\rightarrow^* P_1\langle\!\langle c_{y_1,\ldots,y_n,\Xi}(Y_1, \ldots, Y_n, X_1, \ldots, f(Y_1, \ldots, Y_n), \ldots, X_k)\rangle\!\rangle] \Leftarrow$$
$$[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, f(Y_1, \ldots, Y_n), \ldots X_k)\rangle\!\rangle] \quad (15)$$

however (15) falls outside $\mathcal{H}_1$. But (14) is a safe over-approximation of the latter, in the sense that if $[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, f(Y_1, \ldots, Y_n), \ldots X_k)\rangle\!\rangle]$ holds, then certainly both premises of (14) hold, with $X_i = f(Y_1, \ldots, Y_n)$, and assuming this equality, the head of (14) implies that of (15). So (14) includes at least all the behaviors intended in (15).

The cases of a failed match are handled by the following clauses, where g ranges over all function symbols other than f:

$$[\rightarrow^* Q_1\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_i, \ldots X_k)\rangle\!\rangle], \quad (16)$$
$$[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, g(Y_1, \ldots, Y_n), \ldots X_k)\rangle\!\rangle]$$

When $P = \texttt{case}\ x_i\ \texttt{of}\ [x]_{x_j^{-1}} \Rightarrow P_1\ \texttt{else}\ Q_1$ (with $i \neq j$), we produce the following, where we assume that the only keys that can be used are of the form $\texttt{pub}(X)$ or $\texttt{prv}(X)$,

and that each form is the inverse of the other:

$$[\rightarrow^* P_1\langle\!\langle c_{x,\Xi}(X, X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle],$$
$$[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, [X]_{\mathtt{prv}(Y)}, \ldots, \mathtt{pub}(Y), \ldots, X_k)\rangle\!\rangle] \quad (17)$$

$$[\rightarrow^* P_1\langle\!\langle c_{x,\Xi}(X, X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, \underbrace{[X]_{\mathtt{pub}(Y)}}_{i}, \ldots, \underbrace{\mathtt{prv}(Y)}_{j}, \ldots, X_k)\rangle\!\rangle] \quad (18)$$

Finally, we over-approximate the failed match case bluntly, by estimating that one can always go from P to Q_1; this is benign, as in most protocols $Q_1 = \mathtt{stop}$:

$$[\rightarrow^* Q_1\langle\!\langle Z\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle Z\rangle\!\rangle] \quad (19)$$

We leave symmetric decryption, $P = \mathtt{case}\ x_i\ \mathtt{of}\ \{x\}_{x_j} \Rightarrow P_1\ \mathtt{else}\ Q_1$, as an exercise to the reader. Tests $P = \mathtt{if}\ x_i = x_j\ \mathtt{then}\ P_1\ \mathtt{else}\ Q_1$ are handled similarly:

$$[\rightarrow^* P_1\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, X_k)\rangle\!\rangle], \quad (20)$$
$$[\rightarrow^* P\langle\!\langle c_\Xi(X_1, \ldots, \underbrace{X_i}_{i}, \ldots, \underbrace{X_i}_{j}, \ldots, X_k)\rangle\!\rangle]$$

$$[\rightarrow^* Q_1\langle\!\langle Z\rangle\!\rangle] \Leftarrow [\rightarrow^* P\langle\!\langle Z\rangle\!\rangle] \quad (21)$$

This terminates our description of the set $S(P_0)$ of clauses.

Assume we wish to prove that some message M remains secret to the Dolev-Yao attacker throughout the execution of some system, represented as a process P. This is equivalent to saying that $P|DY(I_0, c)$ cannot ever send M over the public channel c. (We assume that all public communications of P goes through channel c, which is shared with the attacker.) In turn, this is implied by the fact that $[c \triangleright M]$ is not a logical consequence of $S(P|DY(I_0, c))$, because the latter is an over-approximation of the behavior of $P|DY(I_0, c)$. But $[c \triangleright M]$ is not a logical consequence of $S(P|DY(I_0, c))$ if and only if $S(P|DY(I_0, c))$ plus the single clause $\bot \Leftarrow [c \triangleright M]$ is *satisfiable*.

All these clauses are in $\mathcal{H}_1$, and satisfiability in $\mathcal{H}_1$ is decidable, so we have it: a terminating, sound algorithm for weak secrecy in the spi-calculus, in an abstract interpretation setting. In practice, just use `h1` [14], for example, to decide $S(P|DY(I_0, c))$ plus $\bot \Leftarrow [c \triangleright M]$. One may formalize this thus:

Theorem 1. *Let Spi$_1$ be the spi-calculus of Figure 1, with semantics of processes P_0 given by predicates defined through the clause set $S(P_0)$. Then weak secrecy is decidable in Spi$_1$, in exponential time.*

One can do much more with $\mathcal{H}_1$. Nielson, Nielson and Seidl observed [19] that $\mathcal{H}_1$ defined *strongly regular relations*, and this makes a deep connection to automata theory. The set $S(P_0)$ is a collection of *definite* clauses, i.e., no clause in $S(P_0)$ has head $\bot$. Sets of definite clauses are always satisfiable: take the Herbrand model containing *all* ground atoms. So $S = S(P_0)$ has a least Herbrand model $\mathrm{lfp}\, T_S$. Now, for each predicate symbol p, let the *language* $L_p(S)$ be the set of ground terms t such that $p(t) \in \mathrm{lfp}\, T_S$; we also say that t is *recognized* at p in S. This generalizes the corresponding notions for tree automata to arbitrary satisfiable sets S of Horn clauses. In particular, we retrieve tree automata [6] by encoding the transition $f(q_1, \ldots, q_n) \rightarrow q$ (whose effect is that

whenever t_1 is recognized at q_1, ..., t_n is recognized at q_n, then $f(t_1, \ldots, t_n)$ should be recognized at state q) as the clause $q(f(X_1, \ldots, X_n)) \Leftarrow q_1(X_1), \ldots, q_n(X_n)$.

Nielson, Nielson and Seidl observed that, whenever S was a set of definite $\mathcal{H}_1$ clauses, $L_p(S)$ was always a regular language, i.e., one of the form $L_p(\mathcal{A})$ for some tree automaton $\mathcal{A}$, and that one could compute $\mathcal{A}$ from S in exponential time. In fact, one can compute a tree automaton $\mathcal{A}$ that is *equivalent* to S in exponential time, meaning that $L_p(S) = L_p(\mathcal{A})$ for all predicate symbols p at once. This was refined, following the "Logic wins!" motto, in [15], where I showed that no new algorithm was needed for this: good old, well-known variants of the resolution proof-search rule [2] achieve this already. This is in fact what I implemented in `h1` [14].

We use this to decide correspondence assertions as follows. A *correspondence assertion* is a (non-Horn) clause of the form $A \Rightarrow A_1 \vee \ldots \vee A_k$, for some atoms A, A_1, ..., A_k, and our goal will be to check whether such a clause holds in $\mathrm{lfp}\, T_{S(P_0)}$. Typically, one checks authentication of Alice by Bob, in the form of (non-injective) agreement, following Woo and Lam [22], by writing an `event` $begin\langle e_1 \rangle$ at the end of a subprocess describing Alice's role, where she sent message e_1, and `event` $end\langle e_2 \rangle$ in Bob's role, where Bob just received message e_2. We now check that $end\langle X \rangle \Rightarrow begin\langle X \rangle$ holds in $\mathrm{lfp}\, T_{S(P_0)}$, i.e., that there is no term t such that $end\langle t \rangle$ holds in $\mathrm{lfp}\, T_{S(P_0)}$ (Bob received t) but $begin\langle t \rangle$ does not (i.e., Alice never actually sent t).

Although my purpose was different then, I have shown that the model-checking problem for clauses C (in particular, of the form $A \Rightarrow A_1 \vee \ldots \vee A_k$) against models $\mathrm{lfp}\, T_S$ described in the form of $\mathcal{H}_1$ clause sets S, was decidable in [16]: convert S to an equivalent so-called alternating tree automaton $\mathcal{A}$, using the resolution algorithm of [15] for instance, then check whether C holds in the least Herbrand model of $\mathcal{A}$. The latter takes exponential time in the number of predicate symbols of $\mathcal{A}$, equivalently of S, so that the whole process still only takes exponential time. This was implemented in the `h1mc` model-checker, another part of the `h1` tool suite [14]. A bonus is that it also generates a Coq proof of the fact that C indeed holds in $\mathrm{lfp}\, T_S$, which was the main topic of [16]. Anyway, these logical considerations immediately entail:

Theorem 2. *Non-injective agreement is decidable in Spi$_1$, in exponential time. This is also true of any correspondence assertion $A \Rightarrow A_1 \vee \ldots \vee A_k$, or in fact of any property expressed as a finite set of (not necessarily Horn) clauses.*

3 Logic Programs as Types for Logic Programs

However, we have not gone as far as we could in Section 2. In particular, we have been careful so as to define clauses in the decidable class $\mathcal{H}_1$. But logic allows us not to care, or at least to care less. The idea is to write general Horn clauses, and to let a generic algorithm do the abstraction work for us. I.e., this algorithm must take any set S of clauses, and return a new set S' satisfying: (a) if S' is satisfiable, then so is S, and (b) S' is in some fixed decidable class, e.g., $\mathcal{H}_1$.

That such generic abstraction algorithms (for different decidable classes) exist was discovered by Frühwirth *et al.* [12] in a remarkable piece of work. In the case of $\mathcal{H}_1$, the generic abstraction algorithm is simple, and for the main part consists in introducing fresh predicate symbols to name terms t in heads that are too deep [15]. Formally, let a

one-hole context $\mathcal{C}[]$ be a term with a distinguished occurrence of the *hole* $[]$. $\mathcal{C}[u]$ is $\mathcal{C}[]$ with u in place of the hole. $\mathcal{C}[]$ is *non-trivial* iff $\mathcal{C}[] \neq []$. Then define the rewrite relation $\rightsquigarrow$ on clause sets by:

$$p(\mathcal{C}[t]) \Leftarrow \mathcal{B} \quad \rightsquigarrow \quad \begin{cases} p(\mathcal{C}[Z]) \Leftarrow \mathcal{B}, q(Z) & (Z \text{ fresh}) \\ q(t) \Leftarrow \mathcal{B} \end{cases} \tag{22}$$

where t is not a variable, q is fresh, and $\mathcal{C}[]$ is a non-trivial one-hole context, and:

$$p(\mathcal{C}[X]) \Leftarrow \mathcal{B} \quad \rightsquigarrow \quad p(\mathcal{C}[Y]) \Leftarrow \mathcal{B}, \mathcal{B}[X := Y] \tag{23}$$

where X occurs at least twice in $\mathcal{C}[X]$, and Y is a fresh variable. The relation $\rightsquigarrow$ terminates, and any normal form S' of a clause set S satisfies (a) and (b) [15].

So instead of writing clauses, carefully crafted to be in $\mathcal{H}_1$, one may be sloppier and rely on the generic abstraction algorithm. (The `h1` tool does this automatically in case the clause set given to it as input is not in $\mathcal{H}_1$.) For example, we *can* write clause (15) as we intended. While replacing it by clause (14) looked like a hack, one can instead use rule (22) in the definition of $\rightsquigarrow$: take $p = \rightarrow^* P_1 \langle\langle \_ \rangle\rangle$, $\mathcal{C}[] = c_{y_1,\ldots,y_n,\Xi}(Y_1, \ldots, Y_n, X_1, \ldots, [], \ldots, X_k)$, $t = f(Y_1, \ldots, Y_n)$, then (15) rewrites to clauses that are about as over-approximated as (14). (We let the reader do the exercise.)

Generic abstraction algorithms give us considerable freedom. I have long been interested in static analysis frameworks for the security of various actual languages, and an early example is a piece of work I did with F. Parrennes in 2003–2005 [17]. Our `csur` static analyzer takes a C program and security objectives as input, and outputs sets of Horn clauses that over-approximate the system. However, these clauses are in general not Horn, and we rely on the above generic abstraction algorithm to produce $\mathcal{H}_1$ clauses that `h1` can work on.

4 Analyzing Hardware Circuits in VHDL

Until now, I have only stated principles that I have been using in the past. What about tackling a new problem? Over the past few years, several people from industrial and military milieus have asked me whether one could design algorithms to verify cryptographic hardware automatically. These are circuits, described in languages such as VHDL [20], with modules implementing pseudo-random number generation, encryption, decryption, hashing, and signatures. One needs to check whether no sensitive datum inside the circuit ever gets leaked out, and this is done by hand to this date.

However, it seems like techniques such as those that we have used above, or in [17], should apply. The following is a first attempt, on a cryptographic variant of a small subset of VHDL, and should be considered as a proof of concept.

Consider the following variant of behavioral VHDL, obtained by enriching the core language used by Hymans [18] with additional cryptographic primitives. We assume a finite set of *signals* x, y, z, $\ldots$; "signal" is the VHDL name for a program variable. These will take values from a domain we leave implicit, but which should include cryptographic terms. Expressions e are built from signals as in Section 2. *Processes* are now

described by the following grammar:

$P, Q, R, \ldots ::=$	$\texttt{stop}$	stop
	$\mid \texttt{proc}; P$	loop
	$\mid x <= e; P$	signal assignment
	$\mid \texttt{wait on } W \texttt{ for } m; P$	suspension
	$\mid \texttt{if } e_1 = e_2 \texttt{ then } P \texttt{ else } Q$	equality test
	$\mid x <= \nu; P$	fresh name creation
	$\mid f(x_1, \ldots, x_n) <= e_1 \texttt{ in } P \texttt{ else } Q$	constructor pattern-matching
	$\mid \{x\}_{e_2} <= e_1 \texttt{ in } P \texttt{ else } Q$	symmetric decryption
	$\mid [x]_{e_2^{-1}} <= e_1 \texttt{ in } P \texttt{ else } Q$	asymmetric decryption

Constructs above the line are from [18], while constructs below the line are extra cryptographic constructs. (Encryption, hashing, etc., are handled in expressions e, e_1, e_2 as in Section 2, through specific function symbols.) E.g., $f(x_1, \ldots, x_n) <= e_1 \texttt{ in } P \texttt{ else } Q$ is similar to $\texttt{case } e_1 \texttt{ of } f(x_1, \ldots, x_n) \Rightarrow P \texttt{ else } Q$ in the spi-calculus, binding the signals x_1, ..., x_n, except with a signal assignment semantics (see below). In $\texttt{wait on } W \texttt{ for } m; P$, W is a finite set of signals, and $m \in \mathbb{N} \cup \{\infty\}$: this process waits until some signal in W changes, or until m units of time have elapsed, and then proceeds to P. (For complexity purposes, we assume m is written in unary.) A VHDL *program* is a parallel composition of a fixed number of processes $P_1, \ldots, P_n$.

We assume that VHDL programs are well-formed. The critical point is that no two processes in parallel are allowed to write to the same signal. We shall therefore assume that we are given pairwise disjoint sets of signals $A_1, \ldots, A_n$, such that any signal assignment $x <= e; P$ in P_i satisfies $x \in A_i$ ($1 \leq i \leq n$). A_i will be called the *domain* of P_i; Hymans [18] uses a slightly more general definition.

Again, we won't give a formal definition of the semantics, the non-cryptographic part of which can be found in Hymans (op. cit.). The loop $\texttt{proc}; P$ executes P in an infinite loop, i.e., it behaves just like $P; \texttt{proc}; P$, where $P; Q$ denotes sequential composition of P and Q, obtained by replacing $\texttt{stop}$ by Q everywhere in P. We need to explain the peculiar semantics of signal assignment, and how suspension is achieved. One should first realize that execution proceeds in successions of *simulation cycles*, where each process P_i runs sequentially until it stops, i.e., until it reaches $\texttt{stop}$ or a suspension $\texttt{wait on } W_i \texttt{ for } m_i; Q_i$; in this case we say that P_i is *waiting* on W_i for m_i units of time, and Q_i is its *continuation*.

Signal assignment $x <= e$ is peculiar in that it does not assign the value of e to x, but instead *schedules* this change of values to happen at the next simulation cycle. Several assignments to the same signal x are allowed in each process P_i, and the value scheduled for the next simulation cycle is the last one to be assigned to x during the simulation cycle. Say that x *has changed* during a simulation cycle if its scheduled new value is different from its current value.

A simulation cycle terminates once every P_i has reached $\texttt{stop}$ or a suspension; simulation cycles may fail to terminate, but this will be irrelevant. Let E be the set of signals that have changed during the simulation cycle. At the end of the simulation cycle, execution proceeds to the next one. First, all signals are updated to their scheduled new values. Then, say that P_i is *resumable* if it is waiting on a set W_i of signals that

meets E, i.e., such that $W_i \cap E \neq \emptyset$, or it is waiting for $m_i = 0$ unit of time. (The latter case is not considered in [18], since $m_i \neq 0$ there; allowing for $m_i = 0$ will simplify our clauses below.) If at least one process is resumable, then resume all resumable processes, by executing their continuation. Otherwise, let *time pass*, and resume the first processes whose timeout m_i expires. Time passes in this case only.

We define an abstract semantics of a fixed VHDL program by writing clauses defining some predicates indexed by the simulation cycle number k, ranging from 0 to K. It is indeed important not to abstract away this timing information. Practically, this means that we shall write one clause set per simulation cycle, and we shall therefore be limited to a fixed number K of simulation cycles, a situation not uncommon in model-checking. (We shall lift this restriction later.)

Assume our fixed VHDL program P^0 is the parallel composition of P_1^0, ..., P_n^0, and write the clause set $S_{\text{VHDL}}(P^0, K)$ defining our abstract semantics for P^0 during K cycles. The process P_i starts at P_i^0 in simulation cycle 0. It is customary to assume that VHDL signals must be assigned before they are used, so that the initial context ρ is irrelevant. We shall therefore start in a context mapping each signal in A_i to a dummy value $\bot$. Let Ξ_i be the *domain list* obtained by sorting A_i in some fixed way, and let a_i be the length of Ξ, i.e., the cardinality of A_i. We write:

$$[\rightarrow_0^* P_i^0 \langle\!\langle c_{\Xi_i}(\underbrace{\bot, \ldots, \bot}_{a_i \text{ times}})\rangle\!\rangle] \tag{24}$$

where the subscript (0 here) to $\rightarrow^*$ is the simulation cycle number. We shall need another predicate $\bigcirc_{ki}\langle\!\langle \_ \rangle\!\rangle$ ("next cycle") recognizing the scheduled environments for simulation cycle k, $1 \leq k \leq K$, and process i, $1 \leq i \leq n$. Initially:

$$\bigcirc_{ki}\langle\!\langle c_{\Xi_i}(\bot, \ldots, \bot)\rangle\!\rangle \tag{25}$$

Now consider the various forms that a process P among P_1, ..., P_n may assume at simulation cycle k, $0 \leq k \leq K-1$. We shall enumerate clauses, one for each $P \in \bigcup_{i=1}^n Sub(P_i)$, where $Sub(P)$ is the set of *subprocesses* of P. We require $\texttt{wait on}\ W\ \texttt{for}\ m'; Q$ to be a subprocess of $\texttt{wait on}\ W\ \texttt{for}\ m; Q$ for all $m' \leq m$, and $Sub(\texttt{proc}; Q)$ to contain $Q; \texttt{proc}; Q$ and all its subprocesses. A definition such as $Sub(\texttt{proc}; Q) = \{\texttt{proc}; Q\} \cup Sub(Q; \texttt{proc}; Q)$ would be ill-formed, so use the Fischer-Ladner closure trick [11]. Let $Sub(Q) = Sub_{\texttt{stop}}(Q)$ where $Sub_P(\texttt{stop}) = \{P\}$, $Sub_P(\texttt{proc}; Q) = \{\texttt{proc}; Q; P\} \cup Sub_{\texttt{proc};Q;P}(Q)$, $Sub_P(\texttt{wait on}\ W\ \texttt{for}\ m; Q) = \{\texttt{wait on}\ W\ \texttt{for}\ m'; Q; P \mid m' \leq m\} \cup Sub_P(Q)$, $Sub_P(x <= e; Q) = \{x <= e; Q; P\} \cup Sub_P(Q)$ (and similarly for $x <= \nu$), $Sub_P(\texttt{if}\ e_1 = e_2\ \texttt{then}\ P_1\ \texttt{else}\ Q_1) = \{\texttt{if}\ e_1 = e_2\ \texttt{then}\ P_1; P\ \texttt{else}\ Q_1; P\} \cup Sub_P(P_1) \cup Sub_P(Q_1)$, and similarly for constructor pattern-matching and decryption. Clearly, $Sub_P(Q)$ is finite. Moreover, an easy induction on Q shows that $Sub_P(Q) = Sub(Q; P)$, so that indeed $Sub(\texttt{proc}; Q) = \{\texttt{proc}; Q\} \cup Sub(Q; \texttt{proc}; Q)$.

Now enumerate i and k, $1 \leq i \leq n$, $0 \leq k \leq K-1$, and then enumerate $P \in Sub(P_i)$. If $P = \texttt{proc}; Q$, we just write:

$$[\rightarrow_k^* Q; \texttt{proc}; Q\langle\!\langle Z\rangle\!\rangle] \Leftarrow [\rightarrow_k^* \texttt{proc}; Q\langle\!\langle Z\rangle\!\rangle] \tag{26}$$

Assignment is subtler, as we have said. In general, let x_{ij} be the jth signal in Ξ_i. The assignment $P = x_{ij} <= e; Q$ will proceed to Q with its current context Z *unchanged* (clause (27)); only the scheduled value of x_{ij} will change (clause (28)). Clause (29) defines $\texttt{changed}_{kij}\langle\!\langle\rho\rangle\!\rangle$ to over-approximate the cases where x_{ij} has changed during simulation cycle k, with context ρ: we estimate that x_{ij} may have changed if some value has been assigned to it, even when this is x_{ij}'s old value.

$$[\rightarrow_k^* Q\langle\!\langle Z\rangle\!\rangle] \Leftarrow [\rightarrow_k^* x_{ij} <= e; Q\langle\!\langle Z\rangle\!\rangle] \tag{27}$$

$$\bigcirc_{(k+1)i}\langle\!\langle c_{\Xi_i}(Y_1, \ldots, \underbrace{X}_{j}, \ldots, Y_{a_i})\rangle\!\rangle \Leftarrow \bigcirc_{(k+1)i}\langle\!\langle c_{\Xi_i}(Y_1, \ldots, Y_{a_i})\rangle\!\rangle, \quad [\rightarrow_k^* x_{ij} <= e; Q\langle\!\langle Z\rangle\!\rangle], \quad [x_{ij} <= e; Q \vdash_k e \simeq X] \tag{28}$$

$$\texttt{changed}_{kij}\langle\!\langle Z\rangle\!\rangle \Leftarrow [\rightarrow_k^* x_{ij} <= e; Q\langle\!\langle Z\rangle\!\rangle] \tag{29}$$

In (28), we need to make sense of the predicate $P \vdash_k e \simeq \_$, and this is done by clauses similar to (10), (12) and (13), only with the subscript k added to $\vdash$ and $\rightarrow^*$.

The clauses for $x <= \nu$, `if` and `case` constructs are obtained from the corresponding clauses in the spi-calculus by adding k subscripts, and replacing updates of signals by scheduling of new values. E.g., when $P = x_{ij} <= \nu; Q$, we produce the clauses:

$$[\rightarrow_k^* Q\langle\!\langle Z\rangle\!\rangle] \Leftarrow [\rightarrow_k^* x_{ij} <= \nu; Q\langle\!\langle Z\rangle\!\rangle] \tag{30}$$

$$\bigcirc_{(k+1)i}\langle\!\langle c_{\Xi_i}(Y_1, \ldots, \underbrace{\ulcorner P \urcorner(Z)}_{j}, \ldots, Y_{a_i})\rangle\!\rangle \Leftarrow [\rightarrow_k^* x_{ij} <= \nu; Q\langle\!\langle Z\rangle\!\rangle], \bigcirc_{(k+1)i}\langle\!\langle c_{\Xi_i}(Y_1, \ldots, Y_{a_i})\rangle\!\rangle \tag{31}$$

$$\texttt{changed}_{kij}\langle\!\langle Z\rangle\!\rangle \Leftarrow [\rightarrow_k^* x_{ij} <= \nu; Q\langle\!\langle Z\rangle\!\rangle] \tag{32}$$

where $\ulcorner P \urcorner = \ulcorner x_{ij} <= \nu; Q \urcorner$ is a fresh function symbol. Note that (31) is not in $\mathcal{H}_1$ as we have defined it. However it is in Nielson *et al.*'s version of $\mathcal{H}_1$, and in this case the generic abstraction algorithm of [15] is exact; i.e., although we could have given an equivalent set of clauses in (our definition of) $\mathcal{H}_1$, we could afford to be lazy.

The case of resumptions is more interesting. Clause (33) handles the case where P (which we recall is in $Sub(P_i)$) is of the form $\texttt{wait on}\ W\ \texttt{for}\ m; Q$, and some signal $x_{i'j}$ in W, for some i', $1 \leq i' \leq n$ and j, $1 \leq j \leq a_{i'}$, has changed during simulation cycle k. We write one such clause for each value of i' and j with $x_{i'j} \in W$. Clause (34) handles the case of a timeout.

$$[\rightarrow_{k+1}^* Q\langle\!\langle Z'\rangle\!\rangle] \Leftarrow [\rightarrow_k^* \texttt{wait on}\ W\ \texttt{for}\ m; Q\langle\!\langle Z\rangle\!\rangle], \texttt{changed}_{ki'j}\langle\!\langle Z\rangle\!\rangle, \bigcirc_{(k+1)i}\langle\!\langle Z'\rangle\!\rangle \tag{33}$$

$$[\rightarrow_{k+1}^* Q\langle\!\langle Z'\rangle\!\rangle] \Leftarrow [\rightarrow_k^* \texttt{wait on}\ W\ \texttt{for}\ 0; Q\langle\!\langle Z\rangle\!\rangle], \bigcirc_{(k+1)i}\langle\!\langle Z'\rangle\!\rangle \tag{34}$$

Finally, we must write clauses to handle the cases where time passes. We use our right to over-approximate, and estimate that time may pass even when some subprocess was resumable. Let $P = \texttt{wait on}\ W\ \texttt{for}\ m'; Q \in Sub(P_i)$, where $m' = m + 1$ is a non-zero timeout, different from ∞, in (35):

$$[\rightarrow_{k+1}^* \texttt{wait on}\ W\ \texttt{for}\ m; Q\langle\!\langle Z'\rangle\!\rangle] \Leftarrow \bigcirc_{(k+1)i}\langle\!\langle Z'\rangle\!\rangle, [\rightarrow_k^* \texttt{wait on}\ W\ \texttt{for}\ m+1; Q\langle\!\langle Z\rangle\!\rangle] \tag{35}$$

For each P of the form `wait on` W `for` $m; Q \in Sub(P_i)$ ($m \in \mathbb{N} \cup \{\infty\}$), we also add a clause (36) that states that, when execution is resumed at Q with context Z', the context of scheduled values for the next simulation cycle starts out being just Z':

$$\bigcirc_{(k+1)i}\langle\!\langle Z' \rangle\!\rangle \Leftarrow [\rightarrow_k^* Q\langle\!\langle Z' \rangle\!\rangle] \tag{36}$$

This completes the description of the clause set $S_{\text{VHDL}}(P^0, K)$.

We still need to model interaction with an attacker, which we shall again take to be a Dolev-Yao attacker. To this end, we use a predicate $\mathtt{att}_k$ so that $\mathtt{att}_k(t)$ holds whenever the attacker is able to infer the value of t during simulation cycle k. Letting I_0 denote a predicate recognizing the messages initial known to the attacker, we write:

$$\mathtt{att}_0(X) \Leftarrow I_0(X) \tag{37}$$
$$\mathtt{att}_{k+1}(X) \Leftarrow \mathtt{att}_k(X) \tag{38}$$
$$\mathtt{att}_k(\{X\}_Y) \Leftarrow \mathtt{att}_k(X), \mathtt{att}_k(Y) \quad \text{(sym. encryption)} \tag{39}$$
$$\mathtt{att}_k(X) \Leftarrow \mathtt{att}_k(\{X\}_Y), \mathtt{att}_k(Y) \quad \text{(sym. decryption)} \tag{40}$$
$$\mathtt{att}_k([X]_Y) \Leftarrow \mathtt{att}_k(X), \mathtt{att}_k(Y) \quad \text{(asym. encryption)} \tag{41}$$
$$\mathtt{att}_k(X) \Leftarrow \mathtt{att}_k([X]_{\mathtt{pub}(Y)}), \mathtt{att}_k(\mathtt{prv}(Y)) \quad \text{(asymmetric} \tag{42}$$
$$\mathtt{att}_k(X) \Leftarrow \mathtt{att}_k([X]_{\mathtt{prv}(Y)}), \mathtt{att}_k(\mathtt{pub}(Y)) \quad \text{decryption)} \tag{43}$$
$$\mathtt{att}_k(f(X_1, \ldots, X_k)) \Leftarrow \mathtt{att}_k(X_1), \ldots, \mathtt{att}_k(X_k) \tag{44}$$
$$\mathtt{att}_k(X_j) \Leftarrow \mathtt{att}_k(f(X_1, \ldots, X_n)) \tag{45}$$

where $0 \leq k \leq K$ (except $k \leq K-1$ in (38)), $1 \leq j \leq n$ in (45), and f is usually restricted to sets of so-called public functions in (44), and public constructors in (45), in ProVerif parlance [3]. Note that clause (38) states that the attacker remembers from one simulation cycle to the next one. Call this set of clauses S_{DY}.

Next, we model which signals the attacker can read from and write to, yielding a clause set S_{pub}. For any signal x_{ij} that the attacker can read from, write the following clause (46), and for any signal x_{ij} the attacker can write to, write (47), $1 \leq k \leq K$:

$$\mathtt{att}_k(X_j) \Leftarrow \bigcirc_{ki}\langle\!\langle c_{\Xi_i}(X_1, \ldots, X_{a_i}) \rangle\!\rangle \tag{46}$$
$$\bigcirc_{ki}\langle\!\langle c_{\Xi_i}(X_1, \ldots, \underbrace{X}_{j}, \ldots, X_{a_i}) \rangle\!\rangle \Leftarrow \mathtt{att}_k(X), \bigcirc_{ki}\langle\!\langle c_{\Xi_i}(X_1, \ldots, X_{a_i}) \rangle\!\rangle \tag{47}$$

This is the *interface* of the circuit to the attacker.

We finally write clauses stating which signals are *sensitive*, i.e., which signals hold data that the attacker should never learn, and at which simulation cycles. Let $Sens$ be a set of triples (k, i, j): our goal is to ensure that the contents of signal x_{ij} at simulation cycle k is secret for each $(k, i, j) \in Sens$. Then write the collection $S_{\text{sec}}(Sens)$ of all clauses of the following form, $(k, i, j) \in Sens$:

$$\bot \Leftarrow \mathtt{att}_K(X_j), \bigcirc_{ki}\langle\!\langle c_{\Xi_i}(X_1, \ldots, X_{a_i}) \rangle\!\rangle \tag{48}$$

$S = S_{\text{VHDL}}(P^0, K) \cup S_{\text{DY}} \cup S_{\text{pub}} \cup S_{\text{sec}}(Sens)$ is a collection of $\mathcal{H}_1$ clauses, modulo our remark about (31). Moreover, if S is satisfiable, then no Dolev-Yao attacker can ever obtain the values of sensitive signals at the designated simulation cycles.

Taking the approximate semantics defined by $S_{\text{VHDL}}(P^0, K)$ as a reference semantics for a language that we shall call $\text{VHDL}_1(K)$, we say that the *confidentiality objective* $Sens$ is met by P_0 in VHDL_1, with given interface, iff the Dolev-Yao attacker cannot obtain the value of any signal x_{ij} at any simulation time k for any $(k, i, j) \in Sens$.

Theorem 3. *For any $K \in \mathbb{N}$, for any VHDL$_1(K)$ program P^0, any interface, and any finite set $Sens$, it is decidable in exponential time whether the confidentiality objective $Sens$ is met by P^0 with the given interface.*

This only deals with the case of at most K simulation cycles. One way want to overcome this limitation. The standard cures in abstract interpretation are called *widenings*. Here, there is a trivial, *logic-based* widening: fix k_0 with $0 \leq k_0 < K$, and decide to equate all simulation cycles $k \geq k_0$ that are equal modulo $K - k_0$. This is easy to achieve: realizing that all our predicate symbols have a k subscript, say p_k, just add the clauses $p_{k_0}(X) \Leftarrow p_K(X)$ for all p, in effect adding a loop in time. Call $\text{VHDL}_2(k_0, K)$ the language whose semantics is defined by these clauses in addition to $S_{\text{VHDL}}(P^0)$, for each VHDL program P_0. Since these clauses are again in $\mathcal{H}_1$, we obtain:

Theorem 4. *For any $k_0 < K$, for every VHDL$_2(k_0, K)$ program P^0, for any interface, and any finite set $Sens$, it is decidable in exponential time whether the confidentiality objective $Sens$ is met by P^0 with the given interface.*

5 Conclusion

We hope to have shown how logical encodings afforded us easy terminating abstract interpretation algorithms for security. Doing so, we have made the first forays into the design of algorithms that verify cryptographic circuits. Much still has to be done, though.

For one, we have not considered *equational theories*; e.g., if one ever uses the bitwise exclusive-or $\oplus$, a common practice in hardware circuits, one should really reason modulo the fact that $\oplus$ is associative, commutative, idempotent, and has a unit. One promising avenue stems from [16], where I showed that all that `h1` did was essentially looking for finite models. If one replaces `h1` by a finite model-finder such as Paradox [5], and applies the latter to the various clause sets described in this paper, with additional equality clauses that axiomatize the ambient equational theory, one may hopefully obtain proofs of security as finite models, with few states [16, Section 8].

Second, one should really consider computational proofs of security instead of the useful, but unsatisfactory, Dolev-Yao model. Approximating a Hoare logic such as [7] through Horn clauses is probably a good starting point for this.

Finally, when one comes to hardware circuits, one is led to evaluate the impact of timing attacks (which we cannot yet, since time was left implicit in our model—only simulation cycles, not time, were handled), but also of fault injections, of differential power analysis, of electromagnetic leaks… this opens a whole can of worms.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols. *Information and Computation*, 148(1):1–70, Jan. 1999.

2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. North-Holland, 2001.
3. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th Computer Security Foundations Workshop*, pages 82–96. IEEE, 2001.
4. R. E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Trans. Comp.*, C35(8):677–692, 1986.
5. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model building. In P. Baumgartner, editor, *Proc. CADE-19 Workshop W4*, Miami, Florida, July 2003.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. www.grappa.univ-lille3.fr/tata, 1997. Version of Sep. 6, 2005.
7. J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *Proc. 15th ACM Conf. Computer and Communications Security*, pages 371–380. ACM Press, Oct. 2008.
8. P. Devienne, P. Lebègue, A. Parrain, J.-C. Routier, and J. Würtz. Smallest Horn clause programs. *Journal of Logic Programming*, 27(3):227–267, 1994.
9. N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. Workshop on Formal Methods and Security Protocols, July 1999.
10. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.
11. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.
12. T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th Symp. Logic in Computer Science*, pages 300–309. IEEE, 1991.
13. J. Goubault-Larrecq. Implementing tableaux by decision diagrams. Interner Bericht 1996-32, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, 1996.
14. J. Goubault-Larrecq. *The* `h1` *Tool Suite*. LSV, ENS Cachan, CNRS, INRIA projet SECSI, 2003. http://www.lsv.ens-cachan.fr/ goubault/H1.dist/dh1index.html.
15. J. Goubault-Larrecq. Deciding $\mathcal{H}_1$ by resolution. *Inf. Proc. Letters*, 95(3):401–408, 2005.
16. J. Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 2009. To appear. Long version of "Towards producing formally checkable security proofs, automatically", in *Proc. 21st Computer Security Foundations Symposium*, pages 224–238, IEEE, 2008.
17. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *Proc. 6th Intl. Conf. Verification, Model-Checking and Abstract Interpretation (VMCAI'05)*, pages 363–379. Springer Verlag LNCS 3385, 2005. Long version, with mistakes corrected, submitted to a journal, June 2005; available as LSV Research Report 2009-18, July 2009.
18. C. Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In M. V. Hermenegildo and G. Puebla, editors, *Proc. 9th Intl. Static Analysis Symposium*, pages 444–460. Springer Verlag LNCS 2477, 2002.
19. F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *Proc. 9th Intl. Static Analysis Symposium*, pages 20–35. Springer Verlag LNCS 2477, 2002.
20. VHDL synthesis interoperability working group. http://www.eda.org/siwg/, Apr. 1998.
21. C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Proc. 16th Intl. Conf. Automated Deduction*, pages 314–328. Springer-Verlag LNAI 1632, 1999.
22. T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194. IEEE, 1993.