StatVerif: verification of stateful processes

Mark D. Ryan

joint work with Myrto Arapinis, Joshua Phillips and Eike Ritter

FOSAD 2012
Mutable shared state

Agents that have **mutable state shared between runs**:

- Hardware tokens
  - Smart cards: capabilities, ...
  - RFID tags: pseudonyms, ...
  - TPMs: PCRs, ...
  - HSM: PIN codes, ...

- Trusted parties in contract signing

- Mobile phones: TMSIs, location

- Web servers, database servers, ...

- VANETS: pseudonyms

- ...
The Trusted Platform Module (TPM)

1. Secure storage
2. Platform authentication
3. Platform integrity reporting

- 200M in existence (laptops, desktops, servers)
- specified by Trusted Computing Group (> 700 pages)
TPM functionality

Secure storage
- TPM stores keys and other sensitive data in shielded memory
- A user can store content that is encrypted by keys only available to the TPM

Platform authentication
- Each TPM chip has a unique and secret key
- A platform can obtain keys by which it can authenticate itself

Platform measurement and reporting
- TPM contains some internal memory slots called PCRs, and some keys can be locked to a particular PCR value
- PCR values can be modified, but only in a specific way
Digital rights management

Secure environment

unforgeable configuration report
“With a plan they call trusted computing, large media corporations, together with computer companies such as Microsoft and Intel, are planning to make your computer obey them instead of you.”

He calls it “treacherous computing”.

**Richard Stallman**
Creator of GNU, Emacs, GCC, GPL, the Free Software Foundation
Attestation from cloud
let TPM = $\nu p.((!TPM_{session}) \mid TPM_{state})$

case TPM

let TPM_{session} = $\mathit{p}(x). \quad \text{read & lock state}$
$\mathit{c}(y). \quad \text{read command}$
$\ldots \quad \text{process command}$
$\bar{c}\langle output \rangle. \quad \text{return output}$
$p\langle \text{newstate} \rangle. \quad \text{store & release state}$

let TPM_{state} = $\bar{p}\langle \text{initstate} \rangle. \quad \text{provide initial state}$
$!(p(x). \quad \text{receive value into state}$
$p\langle x \rangle)) \quad \text{return stored state}$

This is correct in applied pi calculus. However, because of the abstractions ProVerif makes, one cannot use this encoding to prove properties of the TPM in ProVerif.
Outline of the talk

1. StatVerif syntax
2. StatVerif semantics
3. Clauses
4. Abstractions for termination
5. Conclusion
StatVerif syntax: processes

\[ P, Q ::= \]
\[ \text{out } (M, N); P \]
\[ \text{in } (M, x); P \]
\[ P \parallel Q \]
\[ !P \]
\[ \text{new } a; P \]
\[ \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q \]
\[ \text{if } M = N \text{ then } P \text{ else } Q \]
\[ [s \mapsto M] \]
\[ \text{read } s_1, \ldots, s_n \text{ as } x_1, \ldots, x_n; P \]
\[ s_1, \ldots, s_n := M_1, \ldots, M_n; P \]
\[ \text{lock } s_1, \ldots, s_n; P \]
\[ \text{unlock } s_1, \ldots, s_n; P \]
Example

\[
\text{let } \text{TPMsession } = \text{lock } s; \\
\text{read } s \text{ as } x_{\text{state}}; \\
\text{in } (c, x_{\text{command}}); \\
\ldots \text{process command and compute new state} \ldots \\
\text{s } := \text{newstate}; \\
\text{unlock } s; \\
\text{out } (c, \text{output})
\]
Outline of the talk

1. StatVerif syntax
2. StatVerif semantics
3. Clauses
4. Abstractions for termination
5. Conclusion
Semantic configurations are tuples \((\mathcal{E}, S, \mathcal{P})\) where:

- \(\mathcal{E}\) is a set of names,
- \(S\) maps cell names to their current values, and
- \(\mathcal{P}\) is a multiset of pairs \((P, \lambda)\) with \(P\) a process and \(\lambda\) a set of cell names. \(s \in \lambda\) iff \(P\) has exclusive access to cell \(s\).

Initial configuration for a process \(P\): \((\text{fn}(P), \emptyset, \{(P, \emptyset)\})\)
\((E, S, P \cup \{(s \mapsto M), \emptyset\}) \rightarrow (E, S \cup \{s \mapsto M\}, P)\)  
if \(s \in \text{dom}(E)\) and \(s \notin \text{dom}(S)\)

\((E, S, P \cup \{(\text{lock } s_1, \ldots, s_n; P, \lambda)\}) \rightarrow (E, S, P \cup \{(P, \lambda \cup \{s_1, \ldots, s_n\})\})\)  
if \(\forall (Q, \lambda') \in P. \lambda' \cap \{s_1, \ldots, s_n\} = \emptyset\)

\((E, S, P \cup \{(\text{unlock } s_1, \ldots, s_n; P, \lambda)\}) \rightarrow (E, S, P \cup \{(P, \lambda \setminus \{s_1, \ldots, s_n\})\})\)

\((E, S, P \cup \{(\text{read } s_1, \ldots, s_n \text{ as } x_1, \ldots, x_n; P, \lambda)\}) \rightarrow (E, S, P \cup \{(P\{S(s_1)/x_1, \ldots, S(s_n)/x_n\}, \lambda)\})\)  
if \(\{s_1, \ldots, s_n\} \subseteq \text{dom}(S)\) and \(\forall (Q, \lambda') \in P. \lambda' \cap \{s_1, \ldots, s_n\} = \emptyset\)

\((E, S, P \cup \{(s_1, \ldots, s_n := M_1, \ldots, M_n; P, \lambda)\}) \rightarrow (E, S[s_1 \leftarrow M_1, \ldots, s_n \leftarrow M_n], P \cup \{(P, \lambda)\})\)  
if \(\{s_1, \ldots, s_n\} \subseteq \text{dom}(S)\) and \(\forall (Q, \lambda') \in P. \lambda' \cap \{s_1, \ldots, s_n\} = \emptyset\)
Our translation only applies to StatVerif processes of the form:

\[ \text{new } \tilde{m}; ([s_1 \mapsto M_1] \mid \cdots \mid [s_n \mapsto M_n] \mid P) \]

such that \( P \) contains no \( [s \mapsto M] \).
Outline of the talk

1. StatVerif syntax
2. StatVerif semantics
3. Clauses
4. Abstractions for termination
5. Conclusion
The translation of a StatVerif process generates clauses built around the following two predicates

- \( \text{attacker}(\tilde{M}, N) \) means that state \( \tilde{M} \) is reachable and in that state the attacker might know the value \( N \);

- \( \text{message}(\tilde{M}, K, N) \) means that state \( \tilde{M} \) is reachable and in that state the value \( N \) is available on channel \( K \).
Attacker clauses: constructors and destructors

- The attacker can build new messages by applying any **constructor** to messages he knows

Asymmetric encryption

\[
\text{attacker}(xs, xk) \rightarrow \text{attacker}(xs, pk(xk)) \\
\text{attacker}(xs, xk) \land \text{attacker}(xs, xm) \rightarrow \text{attacker}(xs, aenc(xk, xm))
\]

- The attacker can analyse messages by applying any **destructor** to messages he knows

Asymmetric-key decryption

\[
\text{attacker}(xs, xk) \land \text{attacker}(xs, aenc(pk(xk), xm)) \rightarrow \text{attacker}(xs, xm)
\]
The attacker can **send messages** on public channels

\[ \text{attacker}(xs, xc) \land \text{attacker}(xs, xm) \rightarrow \text{message}(xs, xc, xm) \]

The attacker can **eavesdrop** on public channels

\[ \text{attacker}(xs, xc) \land \text{message}(xs, xc, xm) \rightarrow \text{attacker}(xs, xm) \]
Example of protocol clauses: the hardware token

\[
\begin{align*}
&\text{let config = lock } s; \\
&\quad \text{in } (c, x); \\
&\quad \text{if } y = \text{init} \text{ then} \\
&\quad s := x; \\
&\quad \text{unlock } s
\end{align*}
\]

A tag not yet configured (if \( y = \text{init} \) then) can be configured with any value \( x \) sent to it:

\[
\begin{align*}
&\text{message}\left(\text{init}[\cdot], c[\cdot], x\right) \land \text{message}\left(\text{init}[\cdot], xc, xm\right) \\
&\quad \land \text{message}\left(\text{init}[\cdot], yc, ym\right) \rightarrow \text{message}\left(x, yc, ym\right)
\end{align*}
\]

\[
\begin{align*}
&\text{message}\left(\text{init}[\cdot], c[\cdot], x\right) \land \text{message}\left(\text{init}[\cdot], xc, xm\right) \\
&\quad \land \text{attacker}\left(\text{init}[\cdot], ym\right) \rightarrow \text{attacker}\left(x, ym\right)
\end{align*}
\]
Protocol clauses

\[
[0] \rho H \cdot \phi \lambda = \emptyset \\
[Q_1 \mid Q_2] \rho H \cdot \phi \lambda = [Q_1] \rho H \cdot \phi \lambda \cup [Q_2] \rho H \cdot \phi \lambda \\
[!]Q] \rho H \cdot \phi \lambda = [Q] \rho H \cdot \phi \lambda \\
[new a; Q] \rho H \cdot \phi \lambda = \{[Q](\rho \cup \{a \mapsto [a]\})H \cdot \phi \lambda \} \\
[in (M, x); Q] \rho H \cdot \phi \lambda = [Q] \rho H \cdot \phi \lambda \\
\text{where } \phi' = \phi [k \mapsto \text{fres}\mid k \not\in \lambda ] \text{ and } \rho' = \rho \cup \{x \mapsto x\} \cup \{\phi_k' \mapsto \phi_k' \mid k \not\in \lambda \} \text{ and } H' = H \land \text{message}(\phi', \rho(M), x) \\
[out (M, N); Q] \rho H \cdot \phi \lambda = \{H \Rightarrow \text{message}(\phi, \rho(M), \rho(N)) \cup [Q] \rho H \cdot \phi \lambda \\
\text{let } x = g(M_1, \ldots, M_n) \text{ in } Q_1 \text{ else } Q_2] \rho H \cdot \phi \lambda = [Q_1](\rho \sigma)(H \sigma)(\iota \sigma)(\phi \sigma) \lambda \cup [Q_2] \rho H \cdot \phi \lambda \\
\text{where } \sigma = \text{mgu}(\rho(M), \rho(N)) \\
\text{and } \rho' = \{x \mapsto \rho a'\} \cup \{z a' \mapsto z a' \mid z \in \text{fv}(g(p_1, \ldots, p_n))\} \\
\text{and } \rho' = \{x \mapsto \rho a'\} \cup \{z a' \mapsto z a' \mid z \in \text{fv}(g(p_1, \ldots, p_n))\} \\
[if M = N \text{ then } Q_1 \text{ else } Q_2] \rho H \cdot \phi \lambda = [Q_1](\rho \sigma)(H \sigma)(\iota \sigma)(\phi \sigma) \lambda \cup [Q_2] \rho H \cdot \phi \lambda \\
\text{where } \sigma = \text{mgu}(\rho(M), \rho(N)) \\
\text{and } \rho' = \{x \mapsto \rho a'\} \cup \{z a' \mapsto z a' \mid z \in \text{fv}(g(p_1, \ldots, p_n))\} \\
\text{and } \rho' = \{x \mapsto \rho a'\} \cup \{z a' \mapsto z a' \mid z \in \text{fv}(g(p_1, \ldots, p_n))\} \\
[lock s_i_1, \ldots, s_i_m; Q] \rho H \cdot \phi \lambda = [Q](\rho \cup \{\phi_k' \mapsto \phi_k' \mid k \not\in \lambda \})H \cdot \phi'(\lambda \cup \{i_1, \ldots, i_m\}) \\
\text{where } \phi' = \phi [k \mapsto \text{fres}\mid k \not\in \lambda ] \\
[unlock s_i_1, \ldots, s_i_m; Q] \rho H \cdot \phi \lambda = [Q](\rho \cup \{\phi_k' \mapsto \phi_k' \mid k \not\in \lambda \})H \cdot \phi'(\lambda \cup \{i_1, \ldots, i_m\}) \\
\text{where } \phi' = \phi [k \mapsto \text{fres}\mid k \not\in \lambda ] \\
[read s_i_1, \ldots, s_i_m \text{ as } x_1, \ldots, x_m; Q] \rho H \cdot \phi \lambda = [Q] \rho' H'(x_1 :: \ldots :: x_m :: \iota) \phi' \lambda \\
\text{where } \rho' = \rho \cup \{x_j \mapsto \phi_j' \mid 1 \leq j \leq m\} \cup \{\phi_k' \mapsto \phi_k' \mid k \not\in \lambda \} \cup \{vc \mapsto vc, vm \mapsto vm\} \\
\text{and } \phi' = \phi [k \mapsto \text{fres}\mid k \not\in \lambda ] \\
\text{and } H' = H \land \text{message}(\phi', vc, vm) \\
\text{and } vc, vm \text{ fresh} \\
[s_i_1, \ldots, s_i_m := M_1, \ldots, M_m; Q] \rho H \cdot \phi \lambda = [Q](\rho \cup \{\phi_k' \mapsto \phi_k' \mid k \not\in \lambda \})H \cdot \phi'' \lambda \\
\cup \{H \land \text{message}(\phi', vc, vm) \Rightarrow \text{message}(\phi'', vc, vm)\} \\
\cup \{H \land \text{attacker}(\phi', vm) \Rightarrow \text{attacker}(\phi'', vm)\} \\
\text{where } \phi' = \phi [k \mapsto \text{fres}\mid k \not\in \lambda ] \\
\text{and } \phi'' = \phi [i_j \mapsto \rho(M_j) \mid 1 \leq j \leq m] \\
\text{and } vc, vm \text{ fresh}
Protocol clauses

\[
[0] \rho H_\phi \lambda = \emptyset \\
[Q_1 | Q_2] \rho H_\phi \emptyset = [Q_1] \rho H_\phi \emptyset \cup [Q_2] \rho H_\phi \emptyset \\
[!] Q] \rho H_\phi \emptyset = [Q] \rho H_\phi \emptyset \\
[new a; Q] \rho H_\phi \lambda = \begin{cases} [Q](\rho \cup \{a \mapsto a[i]\}) H_\phi \lambda & \text{if } a \in \text{bn}(P) \\ [Q](\rho \cup \{a \mapsto \text{attn}[]\}) H_\phi \lambda & \text{otherwise} \end{cases} \\
in (M, x); Q \rho H_\phi \lambda = [Q](\rho H_\phi \emptyset) \\
out (M, N); Q \rho H_\phi \lambda = \begin{cases} [H \mapsto \text{message}(\phi', \rho(M), x)] \cup [Q] \rho H_\phi \lambda & \text{where } \phi' = \phi[k \mapsto \text{fresh} \mid k \not\in \lambda] \text{ and } \rho' = \rho \cup \{x \mapsto x\} \cup \{\phi'_k \mapsto \phi'_k \mid k \not\in \lambda\} \text{ and } H' = H \land \text{message}(\phi', \rho(M), x) \\
\text{let } x = g(M_1, \ldots, M_n) \text{ in } Q, \text{ else } Q_2 \rho H_\phi \lambda = [Q_2] \rho H_\phi \lambda \\
\end{cases}
\]

Assignments

\[
[[S_{i_1}, \ldots, S_{i_m}] \overset{=}{:=} M_1, \ldots, M_m; Q] \rho H_\phi \lambda = [Q](\rho \cup \{\phi'_k \mapsto \phi'_k \mid k \not\in \lambda\}) H_\phi'' \lambda \\
\cup \{H \land \text{message}(\phi', \text{vc}, \text{vm}) \Rightarrow \text{message}(\phi'', \text{vc}, \text{vm})\} \\
\cup \{H \land \text{attacker}(\phi', \text{vm}) \Rightarrow \text{attacker}(\phi'', \text{vm})\} \\
\text{where } \phi' = \phi[k \mapsto \text{fresh} \mid k \not\in \lambda] \text{ and } \phi'' = \phi'[i_j \mapsto \rho(M_j) \mid 1 \leq j \leq m] \\
\text{and } \text{vc}, \text{vm} \text{ fresh}
\]

\[
[[s_{i_1}, \ldots, s_{i_m} : M_1, \ldots, M_m; Q] \rho H_\phi \lambda = [Q](\rho \cup \{\phi'_k \mapsto \phi'_k \mid k \not\in \lambda\}) H_\phi'' \lambda \\
\cup \{H \land \text{message}(\phi', \text{vc}, \text{vm}) \Rightarrow \text{message}(\phi'', \text{vc}, \text{vm})\} \\
\cup \{H \land \text{attacker}(\phi', \text{vm}) \Rightarrow \text{attacker}(\phi'', \text{vm})\} \\
\text{where } \phi' = \phi[k \mapsto \text{fresh} \mid k \not\in \lambda] \text{ and } \phi'' = \phi'[i_j \mapsto \rho(M_j) \mid 1 \leq j \leq m] \\
\text{and } \text{vc}, \text{vm} \text{ fresh}
\]
Main result

**Theorem (The StatVerif compiler is correct)**

Let $M$ be a message. Let $P$ be a protocol of the form

\[
\text{new } \tilde{m}; ([s_1 \mapsto M_1] \mid \cdots \mid [s_n \mapsto M_n] \mid Q)
\]

such that $P$ contains no $[s \mapsto M]$.

\[
\text{Clauses}(P) \vdash \text{secrecy}(M) \Rightarrow P \models \text{secrecy}(M)
\]

If $\forall \tilde{K}$ the fact $\text{attacker}(\tilde{K}, M)$ is not derivable from $\text{Clauses}(P)$, then $P$ preserves the secrecy of $M$. 
Outline of the talk

1. StatVerif syntax
2. StatVerif semantics
3. Clauses
4. Abstractions for termination
5. Conclusion
Theorem: There is no selection function such that the saturation of 
\{ \text{attacker} \left( u, x \right) \rightarrow \text{attacker} \left( h \left( u, v \right), x \right), \text{attacker} \left( u, \text{enc} \left( u, m \right) \right) \rightarrow \text{attacker} \left( u, m \right) \} 
will always terminate.
Theorem: There is no selection function such that the saturation of

\[
\{ \text{attacker}(u, x) \rightarrow \text{attacker}(h(u, v), x),
\text{attacker}(u, \text{enc}(u, m)) \rightarrow \text{attacker}(u, m) \}
\]

will always terminate.
Safe abstraction:

Replace

\[ \text{attacker}(x_p, x_v) \land \text{attacker}(x_p, x) \rightarrow \text{attacker}(h(x_p, x_v), x) \]

with \( n \) instances, in which \( x_p \) is

- \( \text{zero}[] \)
- \( h(\text{zero}[], x_1) \)
- \( h(h(\text{zero}[], x_1), x_2) \)
- \( h(h(h(\text{zero}[], x_1), x_2), x_3) \)
- \ldots \)
Notion of $k$-stability

**Definition: $k$-stable**

A rule $R$ is **$k$-stable** if for any substitution $\theta$ grounding for $R$, for any PCR value $u = h(u_1, u_2)$ such that $\text{length}_{pcr}(u) > k$ we have that:

- either $(R\theta)[h(u_1, u_2) \rightarrow u_1] = R(\theta[h(u_1, u_2) \rightarrow u_1])$,
- or $(R\theta)[h(u_1, u_2) \rightarrow u_1]$ is a tautology.

**Examples**

- $\text{attacker}(x_p, \text{certkey}(\text{aik}[]), \text{pair}(x_{pk}, h(\text{zero}[], a_1))) \rightarrow \text{attacker}(x_p, \text{aenc}(x_{pk}, s_1))$
- $\text{attacker}(x_p, x_v) \land \text{attacker}(x_p, x) \rightarrow \text{attacker}(h(x_p, x_v), x)$

**Proposition**

Let $\mathcal{R}$ be a finite set of rules and $Q$ be a query such that $\mathcal{R}$ and $Q$ are $k$-stable. If $Q$ is satisfiable then there exists a $k$-bounded derivation witnessing this fact.
Lemma

Let $k \geq 0$ be an integer and $R = H \rightarrow C$ be a rule such that:

1. for all $h(v_1, v_2) \in st(R)$, $\text{length}_{pcr}(v_1, v_2) \leq k$;
2. for all $h(v_1, v_2) \in st(H)$, we have that $v_1 \notin X$;
3. for all $h(v_1, v_2) \in st(C)$ such that $v_1 \in X$, we have that $C[h(v_1, v_2) \rightarrow v_1] \in H$.

Then, we have that the rule is $k$-stable.

Examples

- $\text{attacker}(x_p, \text{certkey}(\text{ai}k[]), \text{pair}(x_p, h(\text{zero}[], a_1))) \rightarrow \text{attacker}(x_p, \text{aenc}(x_{pk}, s_1))$

- $\text{attacker}(x_p, x_v) \land \text{attacker}(x_p, x) \rightarrow \text{attacker}(h(x_p, x_v), x)$

→ Going back to our running example, it is sufficient to consider $1$-bounded derivation when checking satisfiability of a query.
Outline of the talk

1. StatVerif syntax
2. StatVerif semantics
3. Clauses
4. Abstractions for termination
5. Conclusion
Conclusion

- Defined the **StatVerif calculus**
- Designed **StatVerif compiler**
- **Proved correctness** of the StatVerif compiler
- Proved security of:
  - our *hardware token device*
  - the Mackenzie *et al.* *optimistic contract signing protocol*
  - **TPM authentication protocols**
Current and future work

- Prototype implementation

- More case studies
  - UMTS protocols,
  - RFID protocols,
  - ...

- Extension to authentication properties

- Extension to observational equivalence