Extending Rabbit towards verified networked systems with user-defined semantics for system calls

Atsushi Igarashi[1] Yutaka Ishikawa[2] Sewon Park[1] Taro Sekiyama[2]

1 Kyoto University, Kyoto *2 National Institute of Informatics, Tokyo*

Poster Presented at 第27回プログラミングおよびプログラミング言語ワークショップ *(PPL 2025, Japan)*

Rabbit - Verified Networked Systems

Client Process

```
data = something_sensitive

main():
  var conn = connect(rpc)
  var sig = invoke(conn, "sign_data", data)
  event [Sending (data)]
  var _ = send(udp, data, sig)
  var _ = close(conn)
```

TEE Process

```
privkey = private_key_stored_safely

fun sign_data (data):
 return sign(data, privkey)

main():
  var _ = accept_conn(rpc)
```

Server Process

```
main():
  var (data, sig) = recv(udp)
  if verify(sig, pubkey) then
    event [Validated (data)]
  else
    event [Invalidated (data)]
  end
```

Example system

1. The **Client** sends a sensitive data to a Trusted Execution Environment (**TEE**)
2. The **TEE**, using a private key (*privkey*), signs the data and sends it back
3. The **Client** forwards the signed data to the **Server**
4. The **Server** verifies the signature using the public key (*pubkey*)

Properties to verify

- **Liveness**
  "There exists a path where Validated (data) happens"
- **Authentication**
  "Validated (data) can happen only after Sending (data)"

Rabbit[1]

- A ***modeling language*** for expressing ***networked systems*** and security assertions in *an imperative style*
- Supports a back-end in ***Tamarin Prover***[2], a model checker to *automatically* verify or refute the assertions

Limitation: *Supports only a fixed set of primitive* ***system calls***

Extended Rabbit for System Calls

Expression *given a set of literals and primitive functions*

$e ::= x$	*variable*
$\mid \ell$	*literal (including* strings, integers, ...)
$\mid f(e_1, \ldots, e_n)$	*primitive function application*

Facts

$A, B ::= A(e_1, \ldots, e_n)$	*fact holds locally in the process calling a system call*
$\mid c :: A(e_1, \ldots, e_n)$	*fact holds in channel c*
$\mid :: A(e_1, \ldots, e_n)$	*fact holds globally in the entire system*
$\mid :: Fr(x)$	*"x" is a fresh nonce*
$\mid e_1 = e_2 \mid e_1 \neq e_2 \mid \cdots$	some *built-in primitive facts*

Commands *subsuming expressions*

$c ::= e$	*expression (regarded as a return value)*
$\mid$ **var** $x = e$ **in** c	*local variable (x bound in c)*
$\mid x := e$	*assignment*
$\mid x := f(e_1, \ldots, e_n) \mid x := s(e_1, \ldots, e_n)$	*function application and and system call*
$\mid c_1; c_2$	*sequencing*
$\mid$ **case** $[\vec{A}_1] \rightarrow c_1$ $\mid \cdots \mid [\vec{A}_n] \rightarrow c_n$	*guarded cases; non-deterministically run c_i whose guard $\vec{A}_i$ holds*
$\mid$ **repeat** $[\vec{A}_1] \rightarrow c_1 \mid \cdots \mid [\vec{A}_n] \rightarrow c_n$ **until** $[\vec{B}_1] \rightarrow c'_1 \mid \cdots \mid [\vec{B}_m] \rightarrow c'_m$	*guarded loop; repeat guarded cases; when $\vec{B}_i$ holds, may run c_i then exit*
$\mid$ **put** $[\vec{A}]$	*adding facts $\vec{A}$ to the environment*
$\mid$ **event** $[\vec{A}]$	*labelling events $\vec{A}$*

Example - Remote Process Call

```
syscall accept (c):
  (...)
  repeat
  | [ c :: Invoke(f, args) ] →
    case
    | [ f = "sign" ] →
      var y := sign(args, privkey) in
      event [ :: Sending(y) ]
      put [ c :: Return(y) ]
    | [ f ≠ "sign" ] →
      event [ :: WrongFunName(f) ]
  until
  | [ c :: Exit( ) ] →
      event [ :: Exited( ) ]
```

```
syscall invoke (conn, f, args):
  case [ ConnectedBy(c, conn) ] →
    put [ c :: Invoke(f, args) ];
    case [ c :: Return(y) ] →
      event [ :: Received(y) ]; y
```

Assertions

Reachability of :: Exited( )

The system is well-defined in that there is a transition exits the RPC loop

Non-reachability of :: WrongFunName(f)

There is no case or attack scenario that invokes an RPC call with an invalid function name

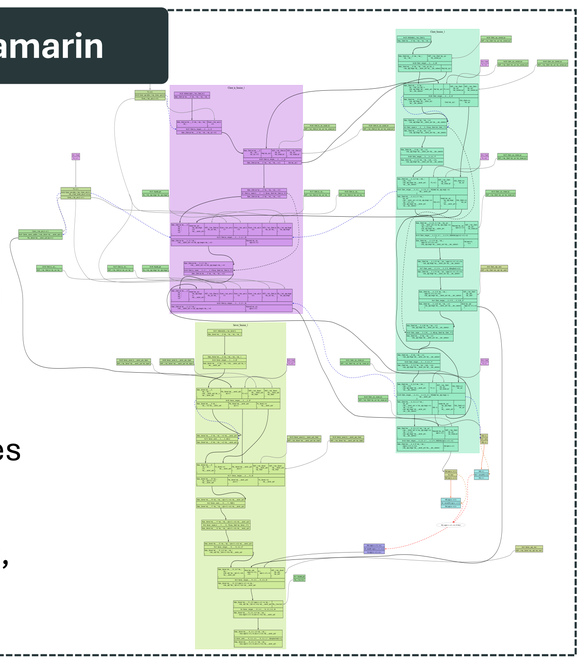
Correspondence of

:: Received(y) ⇝ :: Sending(y)

Whenever an invoke call receives something, it must have been sent by the RPC loop

Implementation and Translation to Tamarin

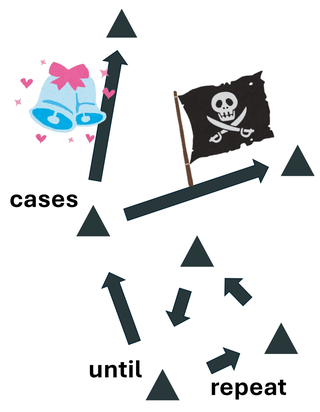
- Implemented in **OCaml** (~ 4k LOC)
- The implementation parses and translates Rabbit programs (including the extended language for user-defined semantics of system calls and assertions) into the **Tamarin prover**
- The Tamarin prover *automatically* proves or disproves the assertions
- If a validating or invalidating path exists, the Tamarin prover prints the path:



Semantics and Assertions

Semantics

- A multi-agent transition system
- Agents communicate through channel- and global facts
- Some transitions are labelled by ***event***s
- *There can be loops and parallel edges (due to* **repeat** *and* **case**)

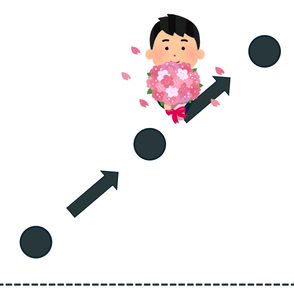


(Non-) Reachability *of* A

- *There* ***exists*** *a path of transitions reaching one labeled* A
- Used to express **liveness** of a system
- Used to ensure that a **failure scenario** never happens

Correspondence *of* $B \rightsquigarrow A$

- In **all** paths leading to B, there must have been A in the past
- Used to express **authentication**



Future Work

- Define various system calls using the extended language as a standard prelude library for Rabbit programs
- Further extend Rabbit toward **program refinement** by providing different abstraction levels for system calls and library calls
- Pre-process and **optimize** the generated Tamarin models to improve performance in automatic verification
- Integrate the **type system** developed by Lelio Brun and Taro Sekiyama for security protocol verification

[1] Terunobu Inaba, Yutaka Ishikawa, Atsushi Igarashi, and Taro Sekiyama. "Rabbit: A Language to Model and Verify Data Flow in Networked Systems." *Proceedings of the 2024 International Symposium on Networks, Computers and Communications (ISNCC 2024)*, October 2024, Washington, D.C., USA.
[2] Tamarin Prover. Available at: https://tamarin-prover.com

JST CREST 「形式検証とシステムソフトウェアの協働によるゼロトラストIoT」 *https://zt-iot.nii.ac.jp/*
国立情報学研究所，京都大学，情報セキュリティ大学院大学，東京情報デザイン専門職大学

ZT-IoT