Executable Universal Composability with Effect Handlers

Markulf Kohlweiss[1], Sam Lindley[1], Sabine Oechsner[2], **Jesse Sigal**[1]

[1]University of Edinburgh, [2]Vrije Universiteit Amsterdam

Outline

Universal Composability (UC)

Expressing UC protocols

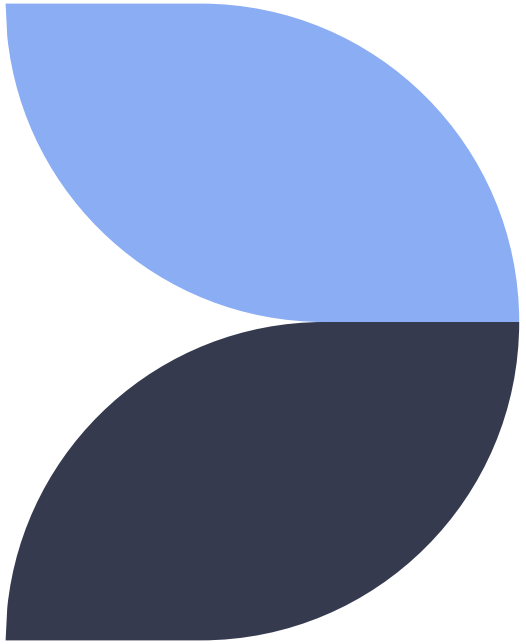
UC in more detail

Secure from authenticated channel example

Conclusion and further work

Universal Composability

What? Why? How?



Universal Composability

What?

Provable security

Composable security proofs

Low-level, e.g. Turing machines or transition functions

Why?

Specifications are integral to implementations and proofs

A formal system

(De)composition

How?

Model of computation

Four kinds of entities

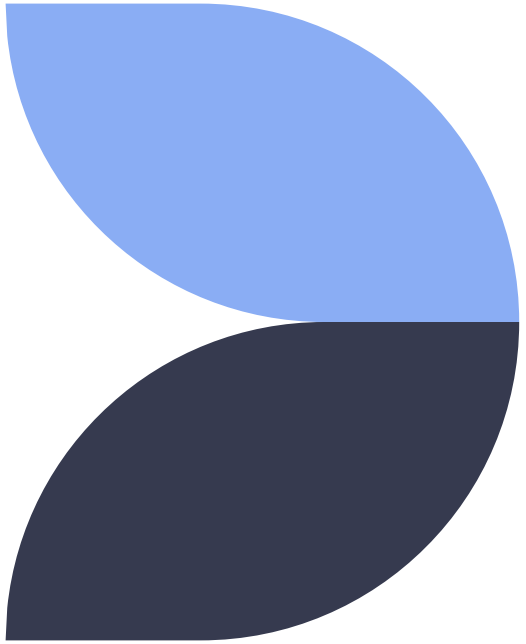
Prescribed interaction

Security as indistinguishability

Our aim: expressive and executable UC framework

Expressing UC protocols

Mind the gap!



Level of abstraction

> On the downside, we note that the ITM model, or “programming language” provides only a relatively low level abstraction of computer programs and protocols. In contrast, current literature describes protocols in a much higher-level (and often informal) language
>
> - Canetti

“Universally Composable Security: A New Paradigm for Cryptographic Protocol”, Canetti (2020)

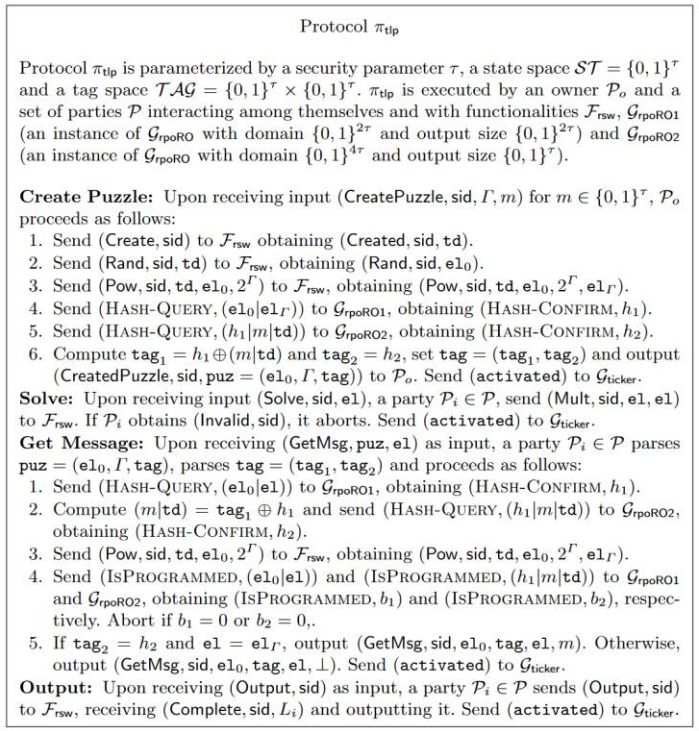
Example informal description

Functionality $\mathcal{F}_{\text{AUTH}}$

1. Upon invocation, with input $(\texttt{Send}, sid, R, m)$ from ITI S, send backdoor message $(\texttt{Sent}, sid, S, R, m)$ to the adversary.
2. Upon receiving backdoor message $(\texttt{ok}, sid)$: If not yet generated output, then output $(\texttt{Sent}, sid, S, R, m)$ to R.
3. Upon receiving backdoor message $(\texttt{Corrupt}, sid, m', R')$, record being corrupted. Next, if not yet generated output then output $(\texttt{Sent}, sid, S, R', m')$ to R'.
4. On input $(\texttt{ReportCorrupted}, sid)$ from S: If corrupted, output `yes` to S. Else output `no`.
5. Ignore all other inputs and backdoor messages.

"Universally Composable Security: A New Paradigm for Cryptographic Protocol", Canetti (2020)

Extract from larger example



Protocol π_{tlp}

Protocol π_{tlp} is parameterized by a security parameter τ, a state space $\mathcal{ST} = \{0,1\}^\tau$ and a tag space $\mathcal{TAG} = \{0,1\}^\tau \times \{0,1\}^\tau$. π_{tlp} is executed by an owner $\mathcal{P}_o$ and a set of parties $\mathcal{P}$ interacting among themselves and with functionalities $\mathcal{F}_{\mathsf{rsw}}$, $\mathcal{G}_{\mathsf{rpoRO1}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{2\tau}$ and output size $\{0,1\}^{2\tau}$) and $\mathcal{G}_{\mathsf{rpoRO2}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{4\tau}$ and output size $\{0,1\}^{\tau}$).

Create Puzzle: Upon receiving input $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ for $m \in \{0,1\}^\tau$, $\mathcal{P}_o$ proceeds as follows:

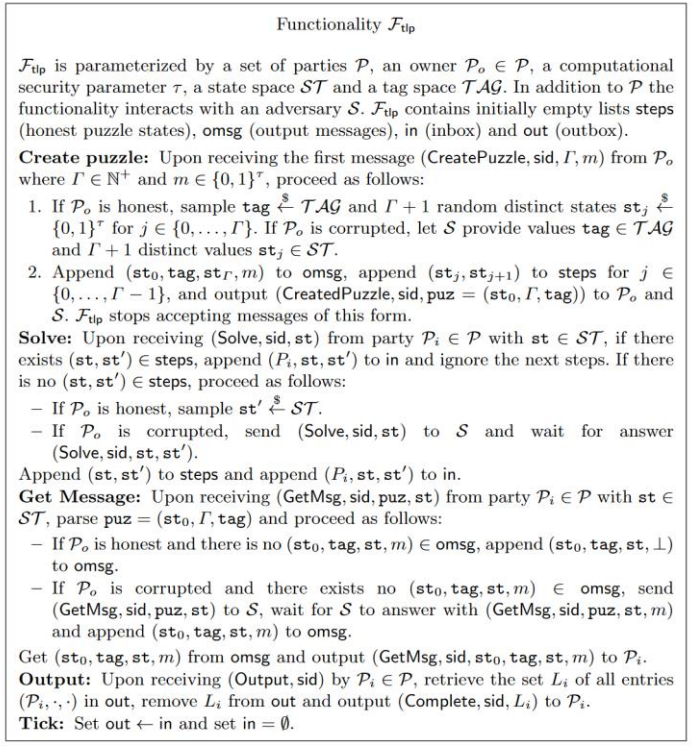
1. Send $(\mathsf{Create}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$ obtaining $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$.
2. Send $(\mathsf{Rand}, \mathsf{sid}, \mathtt{td})$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}_0)$.
3. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma, \mathtt{el}_\Gamma)$.
4. Send $(\textsc{Hash-Query}, (\mathtt{el}_0 | \mathtt{el}_\Gamma))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtaining $(\textsc{Hash-Confirm}, h_1)$.
5. Send $(\textsc{Hash-Query}, (h_1 | m | \mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$.
6. Compute $\mathtt{tag}_1 = h_1 \oplus (m|\mathtt{td})$ and $\mathtt{tag}_2 = h_2$, set $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag}))$ to $\mathcal{P}_o$. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Solve: Upon receiving input $(\mathsf{Solve}, \mathsf{sid}, \mathtt{el})$, a party $\mathcal{P}_i \in \mathcal{P}$, send $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{rsw}}$. If $\mathcal{P}_i$ obtains $(\mathsf{Invalid}, \mathsf{sid})$, it aborts. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathtt{puz}, \mathtt{el})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ parses $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$, parses $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ and proceeds as follows:

1. Send $(\textsc{Hash-Query}, (\mathtt{el}_0 | \mathtt{el}))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtaining $(\textsc{Hash-Confirm}, h_1)$.
2. Compute $(m|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$ and send $(\textsc{Hash-Query}, (h_1|m|\mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$.
3. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma, \mathtt{el}_\Gamma)$.
4. Send $(\textsc{IsProgrammed}, (\mathtt{el}_0|\mathtt{el}))$ and $(\textsc{IsProgrammed}, (h_1|m|\mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{IsProgrammed}, b_1)$ and $(\textsc{IsProgrammed}, b_2)$, respectively. Abort if $b_1 = 0$ or $b_2 = 0$,.
5. If $\mathtt{tag}_2 = h_2$ and $\mathtt{el} = \mathtt{el}_\Gamma$, output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, m)$. Otherwise, output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, \bot)$. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$, receiving $(\mathsf{Complete}, \mathsf{sid}, L_i)$ and outputting it. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.



Functionality $\mathcal{F}_{\mathsf{tlp}}$

$\mathcal{F}_{\mathsf{tlp}}$ is parameterized by a set of parties $\mathcal{P}$, an owner $\mathcal{P}_o \in \mathcal{P}$, a computational security parameter τ, a state space $\mathcal{ST}$ and a tag space $\mathcal{TAG}$. In addition to $\mathcal{P}$ the functionality interacts with an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ contains initially empty lists steps (honest puzzle states), omsg (output messages), in (inbox) and out (outbox).

Create puzzle: Upon receiving the first message $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ from $\mathcal{P}_o$ where $\Gamma \in \mathbb{N}^+$ and $m \in \{0,1\}^\tau$, proceed as follows:

1. If $\mathcal{P}_o$ is honest, sample $\mathtt{tag} \xleftarrow{\$} \mathcal{TAG}$ and $\Gamma + 1$ random distinct states $\mathtt{st}_j \xleftarrow{\$} \{0,1\}^\tau$ for $j \in \{0, \ldots, \Gamma\}$. If $\mathcal{P}_o$ is corrupted, let $\mathcal{S}$ provide values $\mathtt{tag} \in \mathcal{TAG}$ and $\Gamma + 1$ distinct values $\mathtt{st}_j \in \mathcal{ST}$.
2. Append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}_\Gamma, m)$ to omsg, append $(\mathtt{st}_j, \mathtt{st}_{j+1})$ to steps for $j \in \{0, \ldots, \Gamma - 1\}$, and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}))$ to $\mathcal{P}_o$ and $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ stops accepting messages of this form.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ from party $\mathcal{P}_i \in \mathcal{P}$ with $\mathtt{st} \in \mathcal{ST}$, if there exists $(\mathtt{st}, \mathtt{st}') \in$ steps, append $(P_i, \mathtt{st}, \mathtt{st}')$ to in and ignore the next steps. If there is no $(\mathtt{st}, \mathtt{st}') \in$ steps, proceed as follows:

- If $\mathcal{P}_o$ is honest, sample $\mathtt{st}' \xleftarrow{\$} \mathcal{ST}$.
- If $\mathcal{P}_o$ is corrupted, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{S}$ and wait for answer $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}, \mathtt{st}')$.

Append $(\mathtt{st}, \mathtt{st}')$ to steps and append $(P_i, \mathtt{st}, \mathtt{st}')$ to in.

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}, \mathtt{st})$ from party $\mathcal{P}_i \in \mathcal{P}$ with $\mathtt{st} \in \mathcal{ST}$, parse $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ and proceed as follows:

- If $\mathcal{P}_o$ is honest and there is no $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in$ omsg, append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, \bot)$ to omsg.
- If $\mathcal{P}_o$ is corrupted and there exists no $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in$ omsg, send $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}, \mathtt{st})$ to $\mathcal{S}$, wait for $\mathcal{S}$ to answer with $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}, \mathtt{st}, m)$ and append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ to omsg.

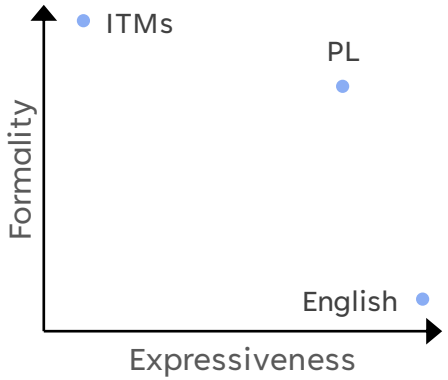
Get $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ from omsg and output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ to $\mathcal{P}_i$.

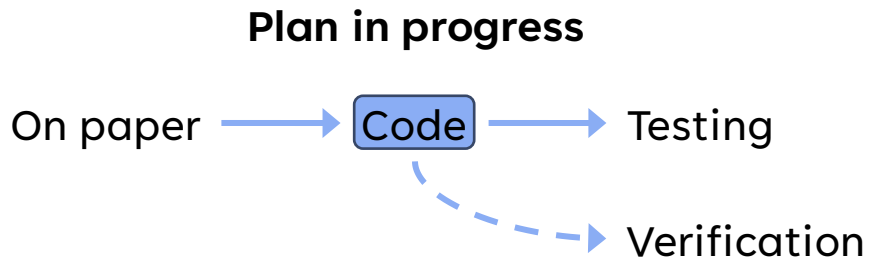
Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$, retrieve the set L_i of all entries $(\mathcal{P}_i, \cdot, \cdot)$ in out, remove L_i from out and output $(\mathsf{Complete}, \mathsf{sid}, L_i)$ to $\mathcal{P}_i$.

Tick: Set out $\leftarrow$ in and set in $= \emptyset$.

"TARDIS: A Foundation of Time-Lock Puzzles in UC", Baum, David, Dowsley, Nielsen, Oechsner (2021)

Goal: a useful middle ground





Solution

Use a high-level formal language expressive enough for UC

Even better, a *programming language*

Don't aim for proofs, just execution

Benefits

Precision

Testing

Debugging

Experimentation

Reuse existing entities

Program verification?

Relevance to proofs

Precision is necessary for proofs

Experimentation, debugging, and testing are fundamental in the creation of entities as well as the process of proving

Reusing the work of others lowers the burden

Variants of UC are easy to explore; different *meta-theories*

By phrasing UC as programs, *potential for UC as program verification*

Why effect handlers?

UC needs randomness, state, and messaging

UC prescribes a form of cooperative concurrency

Effect handlers are powerful enough to express all of these in a unified way

But the end user need not use them directly!

Benefits of effect handlers

Dynamic

Different random sampling distributions or even complete enumeration

Replace randomness with deterministic pseudo-randomness for testing

Replace randomness with a list of samples for low probability events

Configurable levels of behaviour observation

Compositional

UC variations are simple

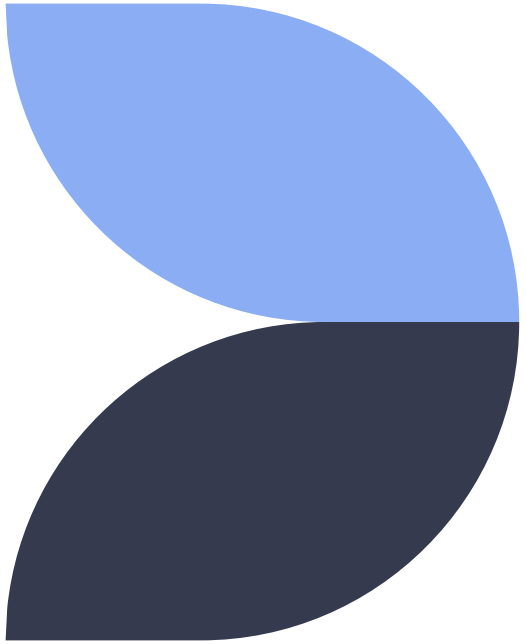
One entity in many UC variations

Composition of UC entities from composition of effect handlers

All with one implementation

Universal Composability

In more detail



Model of computation

Interactive Turing machines (Canetti): Turing machines with multiple tapes, some readable and writeable from other machines, some read only. *Original definition.*

Interactive agents (CDN): Based on probabilistic transition functions. Take an input state and activation point, return an output state and command. *Variation.*

"Universally Composable Security: A New Paradigm for Cryptographic Protocol", Canetti (2001, 2020)
"Secure Multiparty Computation and Secret Sharing", Cramer, Damgård, Nielsen (2015)

Four kinds of entities

Resources (ideal functionalities): ideal description of a (communication) protocol

Parties (form a protocol): agents using a resources to make a new resource

Simulators: technical feature to phrase security

Environments: adversary which can interact with and observe the system

"Secure Multiparty Computation and Secret Sharing", Cramer, Damgård, Nielsen (2015)

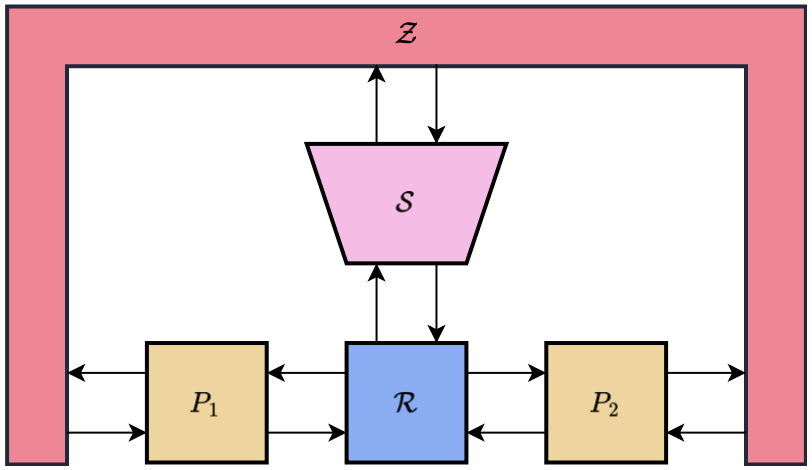
Prescribed interaction

Fixed

Composition

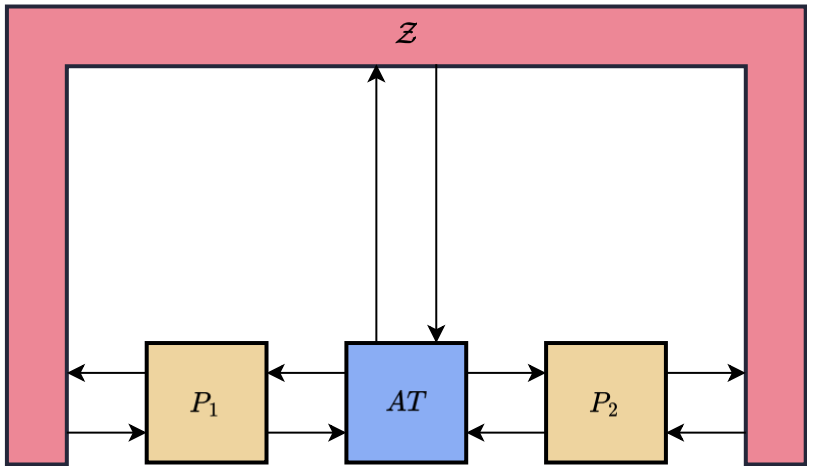
Concurrency

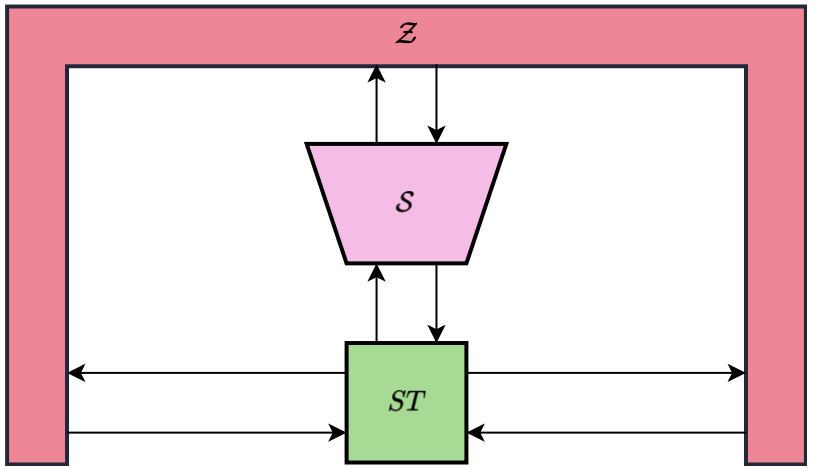
Message passing



"Secure Multiparty Computation and Secret Sharing", Cramer, Damgård, Nielsen (2015)

Security as indistinguishability

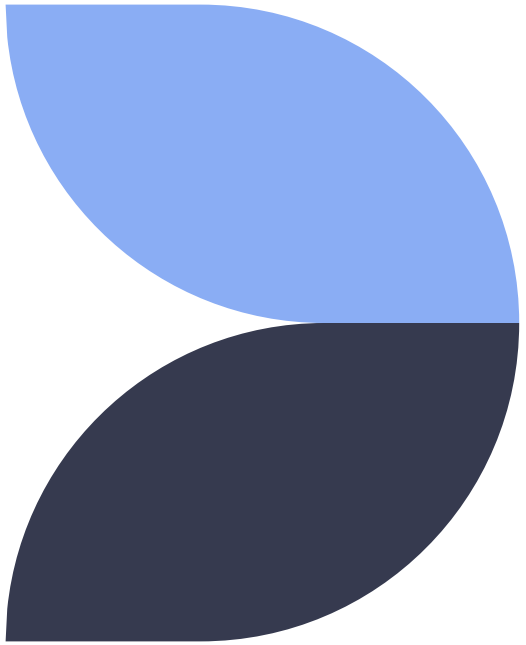




"Secure Multiparty Computation and Secret Sharing", Cramer, Damgård, Nielsen (2015)

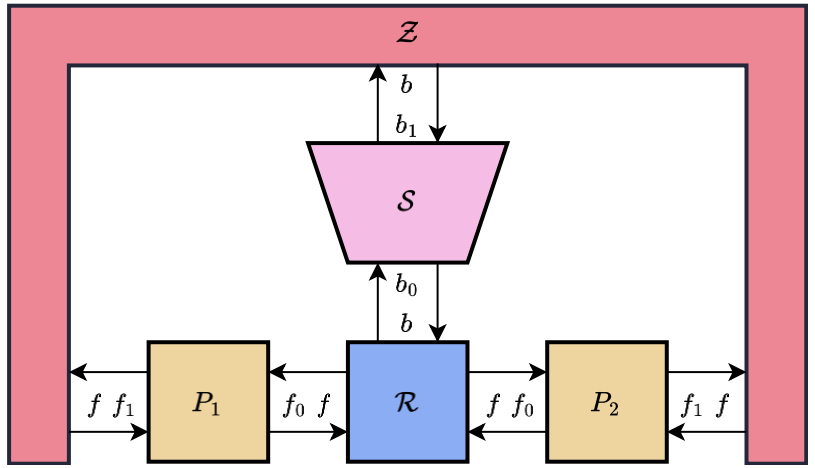
Secure from authenticated channel

A simple example



Framework: the entities

```
resource<f,b,e>
party<f0,f1,e>
protocol<f0,f1,e>
simulator<b0,b1,e>
environment<f,b,e>
```



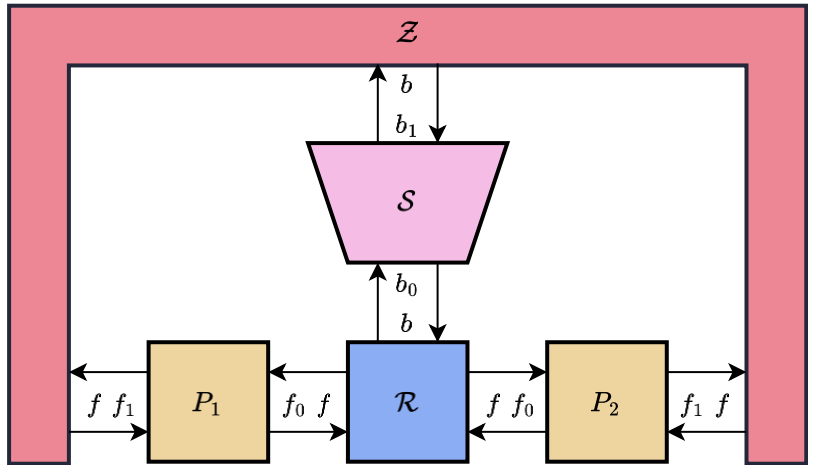
Framework: combining entities

```
fun create-protocol(
  pars : list<party<f0,f1,e>>
) : protocol<f0,f1,e>

fun using-resource(
  pro : protocol<f0,f1,e>, res : resource<f0,b,e>
) : <error> resource<f1,b,e>

fun applying-simulator(
  res : resource<f,b0,e>, sim : simulator<b0,b1,e>
) : <error> resource<f,b1,e>

fun in-environment(
  res : resource<f,b,e>, env : environment<f,b,e>
) : <distinguish,error,div|e> void
```

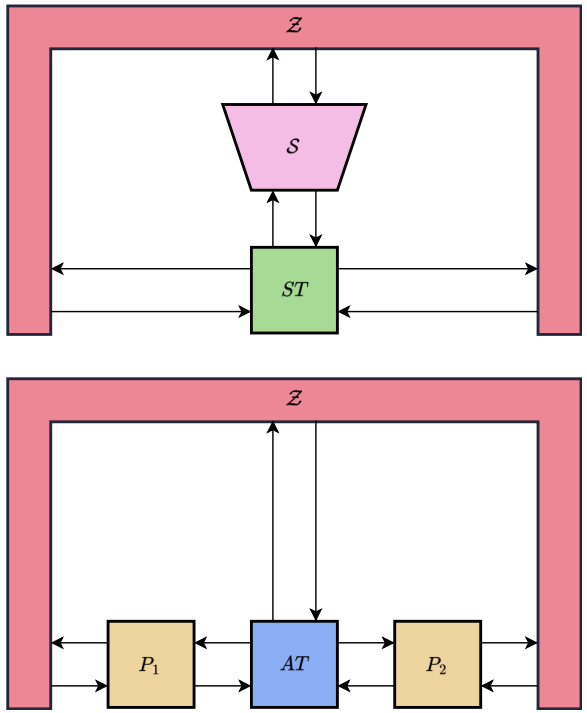


User: Secure from authenticated

```
val resource-st : resource<st-func,st-back,<console>>
val resource-at : resource<at-func,at-back,<console>>
val party-one : party<at-func,st-func,<console>>
val party-two : party<at-func,st-func,<console>>
val protocol : protocol<at-func,st-func,<console>> =
  create-protocol([party-one, party-two])
val simulator : simulator<st-back,at-back,<console>>
val environment : environment<st-func,at-back,<console>>

protocol
  .using-resource(resource-at)
  .in-environment(environment)

resource-st
  .applying-simulator(simulator)
  .in-environment(environment)
```

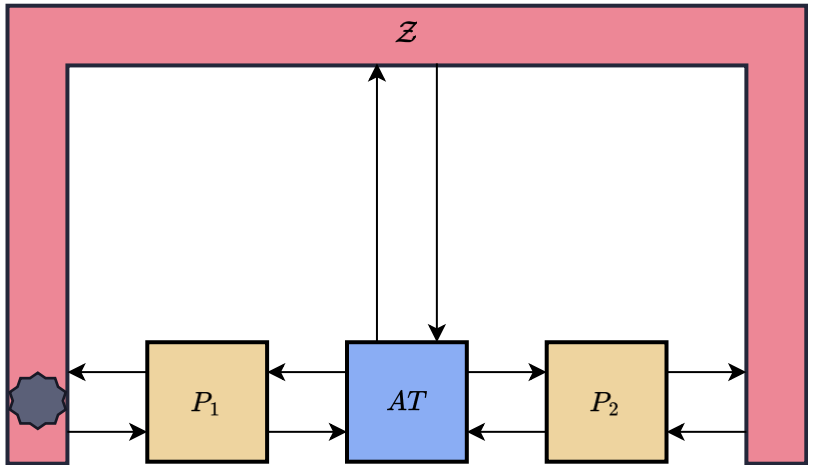


Note we wrote a concrete environment!

Executions

AT and protocol

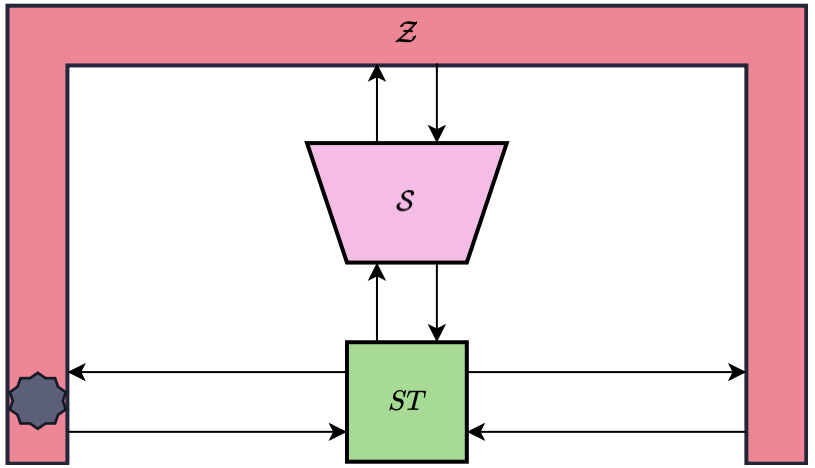
```
En: started
P1: activated on func port: STInOne
AT: activated on func port: ATInOne
En: activated on back port: ATLeak
AT: activated on back port: ATInfl
P2: activated on func port: ATOutTwo
AT: activated on func port: ATInTwo
En: activated on back port: ATLeak
AT: activated on back port: ATInfl
P1: activated on func port: ATOutOne
AT: activated on func port: ATInOne
En: activated on back port: ATLeak
AT: activated on back port: ATInfl
P2: activated on func port: ATOutTwo
En: activated on func port: STOutTwo
```



Executions

ST and simulator

```
En: started
ST: activated on func port: STInOne
Sm: activated on back port: STLeak
En: activated on back port: ATLeak
Sm: activated on back port: ATInfl
En: activated on back port: ATLeak
Sm: activated on back port: ATInfl
En: activated on back port: ATLeak
Sm: activated on back port: ATInfl
ST: activated on back port: STInfl
En: activated on func port: STOutTwo
```



Environment view

AT and protocol

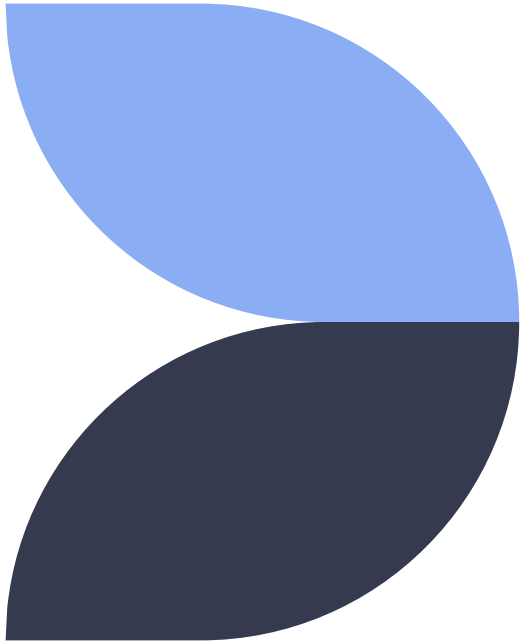
```
En: started
En: sending message:
    STIn(MID(1), PID(2), Msg(test))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(1), PID(2), Msg(HELLO))
En: instructing delivery with:
    ATInfl(MID(1), PID(1), PID(2))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(2), PID(1), Msg(REAL-KEY))
En: instructing delivery with:
    ATInfl(MID(1), PID(2), PID(1))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(1), PID(2), Msg(REAL-KEYtest))
En: instructing delivery with:
    ATInfl(MID(1), PID(1), PID(2))
En: activated on func port: STOutTwo
En: got message id:
    STOut(MID(1), PID(1), Msg(test))
```

ST and simulator

```
En: started
En: sending message:
    STIn(MID(1), PID(2), Msg(test))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(1), PID(2), Msg(HELLO))
En: instructing delivery with:
    ATInfl(MID(1), PID(1), PID(2))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(2), PID(1), Msg(SIM-KEY))
En: instructing delivery with:
    ATInfl(MID(1), PID(2), PID(1))
En: activated on back port: ATLeak
En: got leak:
    ATLeak(MID(1), PID(1), PID(2), Msg(SIM-KEY0000))
En: instructing delivery with:
    ATInfl(MID(1), PID(1), PID(2))
En: activated on func port: STOutTwo
En: got message id:
    STOut(MID(1), PID(1), Msg(test))
```

Conclusion and further work

Into the future!



Finishing up

Conclusion

UC is a formal framework for composable security proofs

UC is too low-level, so protocols are still too informal

PL and effect handlers provide an executable & expressive formal system

Further work

Better user support

Full library

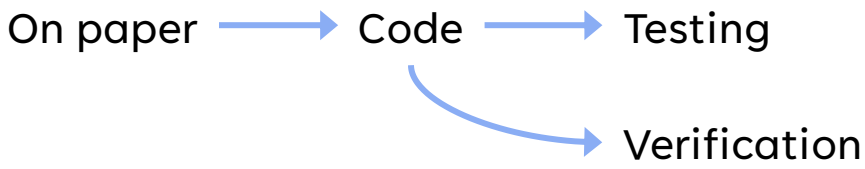
Bigger case studies

Testing tools

Reasoning

Thank you!

Related work



Verification, low-level

EasyUC (EasyCrypt) [Canetti, Stoughton, Varia, 2019]

Constructive Cryptography (Isabelle/CryptHOL) [Lochbihler et al., 2019]

On paper, low-level

ILC [Liao, Hammer, Miller, 2019]

Encode UC in existing system

EasyUC (EasyCrypt) [Canetti, Stoughton, Varia, 2019]

Cost logic (EasyCrypt) [Barbosa et al., 2021]

High-level PL

IPDL [Gancher et al., 2023]

This work