SaUCy: Super Amazing Universal ComposabilitY

Kevin Liao

Abstract. This is a report for class.

1 Introduction

Proving that a protocol is secure is an essential component in cryptographic protocol design. To do so, we must rigorously define what security means, and then demonstrate that the protocol lives up to the definition. Initial security definitions for cryptographic tasks consider only a standalone execution of the protocol. While this simplifies analyses, these definitions are insufficient when analyzing the protocol in more complex contexts. Extended security definitions have been formulated to remedy these drawbacks, but are often complex and ad hoc.

The Universal Composability (UC) framework by Canetti [2] is a general framework for reasoning about the security of practically any cryptographic protocol in a unified and systematic way. The idea is to formulate a security definition and analyze the protocol as in the standalone model, but to additionally show that the protocol composes securely with other arbitrary protocols, i.e., the protocol is secure in *any context*. The benefit of composability is that complex, UC-secure protocols can then be modularly constructed from smaller UC-secure "building blocks".

While universally composable security is a powerful guarantee, elaborating UC proofs is highly nontrivial. They are notoriously complex and exist primarily as "pen-and-paper" proofs, which makes many of them essentially unverifiable. Thus, the goal of this work is to place the UC framework on a proper analytic foundation by developing computer-aided tools and techniques for elaborating UC proofs.

Contributions. The main contributions of this work are the following:

1. We design and implement a programming language called the Interactive Lambda Calculus (ILC for short) for constructing algorithmic entities within the UC execution model.
2. We implement the UC execution model in ILC along with a "library" of example algorithmic entities (ideal functionalities).
3. We build a compiler for translating ILC programs into EasyCrypt [1] modules, which allows us to construct cryptographic proofs using EasyCrypt's interactive theorem proving facilities.

Organization. In Section . . .

2 Background

Demonstrating that a protocol "does its job securely" is an essential component in cryptographic protocol design. One main challenge in formulating the security of cryptographic protocols is capturing threats coming from the execution environment. Another challenge is coming up with security definitions that allow building and analyzing large cryptographic protocols from simpler building blocks while preserving security. Addressing both of these challenges is the focus of the Universal Composability framework.

Previous definitions of security, which considered only standalone execution or sequential execution of protocols, have been shown to be insufficient in many contexts where protocols are deployed within more general environments. Extended security definitions which directly represent a given environment were also insufficient, often complex and limited in scope.

In Universal Composability, protocols are analyzed *in vitro*, that is, in isolation as a single protocol instance, which simplifies the analysis. Security *in vivo*, that is, in realistic settings where the protocol may run concurrently with other protocols, is guaranteed by making sure security is preserved under a general composition operation on protocols.

The high-level approach for determining whether a protocol is secure for some cryptographic task goes back to [3]. First envision an *ideal protocol* ϕ that is secure by construction for carrying out the cryptographic task. In the ideal protocol, parties hand their inputs to a trusted party, called an *ideal functionality* $\mathcal{F}$, who locally computes and hands to each party their corresponding output. We can think of this ideal protocol as a formal specification of the security requirements of the task. Then, we say that a *real world* protocol π is secure if it *realizes* (emulates) ϕ. Specifically, for any real world adversary $\mathcal{A}$, there exists an ideal process adversary $\mathcal{S}$ (called a simulator), such that the output of running π with $\mathcal{A}$ is indistinguishable from the output of running ϕ with $\mathcal{S}$.

This security suffices for achieving standalone security, in which a single protocol instance runs in isolation, but may be composed *sequentially* (but not concurrently) with other protocol instances. To generalize secure composition to the concurrent setting, the UC framework adds a new algorithmic entity to the setup described above, called the *environment*. Intuitively, the environment represents everything external to the current protocol execution. Additionally, the UC framework has a different notion of emulation. Namely, the environment $\mathcal{E}$ can interact freely with the adversary $\mathcal{A}$ throughout the protocol execution. Thus, the environment "interactively distinguishes" between whether it is interacting with the real-world setup with π and $\mathcal{A}$ or the ideal-world setup with ϕ and $\mathcal{S}$. We say that π UC-realizes ϕ if for all environments $\mathcal{E}$ and for all adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot tell with greater than negligible probability whether it is interacting with π and $\mathcal{A}$ or ϕ and $\mathcal{S}$.

Definition 1 (Protocol Emulation). *Let π and ϕ be protocols. We say that π UC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for* *any balanced PPT environment $\mathcal{E}$ we have:*

$$\mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{E}}.$$

Theorem 1 (Universal composition: General statement). *Let ρ, π, ϕ be PPT protocols such that π UC-emulates ϕ and both π and ϕ are subroutine respecting. Then protocol $\rho^{\phi\rightarrow\pi}$ UC-emulates protocol ρ.*

3 SaUCy Execution

4 ILC Language Definition

5 Related Work

6 Conclusion

References

1. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer, 2011.
2. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.
3. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

```
execUC(ℰ, π, 𝒜, ℱ)

ν z2p z2f z2a p2f p2a a2f.
// The environment chooses SID, conf, and corrupted parties
let (Corrupted, SID, conf) = ℰ{z2p, z2a, z2f}
// The protocol determines conf′
let conf′ = π.cmap(SID, conf)
| 𝒜{SID, conf, Corrupted, a2z, a2p, a2f}
| ℱ{SID, conf′, Corrupted, f2z, f2p, f2a}
// Create instances of parties on demand
let partyMap = ref empty
let newPartyPID = do
  ν f2pp z2pp.
  @partyMap[PID].f2p := f2pp
  @partyMap[PID].z2p := z2pp
  | forever do {m ← pp2f; (PID, m) → f2p}
  | forever do {m ← pp2z; (PID, m) → z2p}
  | π{SID, conf, p2f/pp2z, p2z/pp2z}
let getParty PID =
    if PID ∉ partyMap then newParty PID
    return @partyMap[PID]
| forever do
  (PID, m) ← z2p
  if PID ∈ Corrupted then Z2P(PID, m) → p2a
  else m → (getParty PID).z2p
| forever do
  (PID, m) ← f2p
  if PID ∈ Corrupted then F2P(PID, m) → p2a
  else m → (getParty PID).f2p
| forever do
  | A2P2F(PID, m) ← a2p
   if PID ∈ Corrupted then (PID, m) → p2f
  | A2P2Z(PID, m) ← a2p
   if PID ∈ Corrupted then (PID, m) → p2z
```

Fig. 1. Definition of the SaUCy execution model. The environment, are run as concurrent processes. A new instance of the protocol π is created, on demand, for each party PID. Messages sent to honest parties are routed according to their PID; messages sent to corrupted parties are instead diverted to the adversary.

Value Types	A, B	$::=$	x	Value variable
			$\mid$ unit	Unit value
			$\mid$ nat	Natural number
			$\mid$ $A \times B$	Product
			$\mid$ $A + B$	Sum type
			$\mid$ $\mathbf{!}A$	Intuitionistic type
			$\mid$ $\mathbf{Rd}\, A$	Read channel
			$\mid$ $\mathbf{Wr}\, A$	Write channel
			$\mid$ $\mathbf{U}\, C$	Thunk type
Computation Types	C, D	$::=$	$A \to C$	Value-consuming computation
			$\mid$ $\mathbf{F}\, A$	Value-producing computation
Linear Typing Contexts	Δ	$::=$	$\cdot \mid \Delta, x : A$	
Intuitionisitic Typing Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$	

Fig. 2. Syntax of types and typing contexts

Values	v	$::=$	x	
			$\mid$ $\texttt{()}$	Unit value
			$\mid$ n	Natural number
			$\mid$ (v_1, v_2)	Pair of values
			$\mid$ $\texttt{inj}_i(v)$	Injected value
			$\mid$ $\texttt{chan}(c)$	Channel (either read or write end)
			$\mid$ $\texttt{thunk}(e)$	Thunk (suspended, closed expression)
Expressions	e	$::=$	$\texttt{split}(v, x_1.x_2.e)$	Pair elimination
			$\mid$ $\texttt{case}(v, x_1.e_1, x_2.e_2)$	Injection elimination
			$\mid$ $\texttt{ret}(v)$	Value-producing computation
			$\mid$ $\texttt{let}(e_1, x.e_2)$	Let-binding/sequencing
			$\mid$ $e\, v$	Function application
			$\mid$ $\lambda x.\, e$	Function abstraction
			$\mid$ $\texttt{force}(v)$	Unsuspend (force) a thunk
			$\mid$ $\texttt{wr}(v_1 \leftarrow v_2)$	Write channel v_1 with value v_2
			$\mid$ $\texttt{rd}(v)$	Read channel v
			$\mid$ $\nu x.\, e$	Allocate channel as x in e
			$\mid$ $e_1 \mid\!\triangleright\, e_2$	Fork e_1, continue as e_2
			$\mid$ $e_1 \oplus e_2$	External choice between e_1 and e_2

Fig. 3. Syntax of values and expressions

Modes m, n, p ::= $\mathsf{W} \mid \mathsf{R} \mid \mathsf{V}$ (Write, Read and Value)

$\boxed{m \parallel n \Rightarrow p}$ The parallel composition of modes m and n is mode p.

$$\frac{m \parallel n \Rightarrow p}{n \parallel m \Rightarrow p}\ \text{sym} \qquad \frac{}{\mathsf{W} \parallel \mathsf{V} \Rightarrow \mathsf{W}}\ \text{wv} \qquad \frac{}{\mathsf{W} \parallel \mathsf{R} \Rightarrow \mathsf{W}}\ \text{wr} \qquad \frac{}{\mathsf{R} \parallel \mathsf{R} \Rightarrow \mathsf{R}}\ \text{rr}$$

$\boxed{m\ ;\ n \Rightarrow p}$ The sequential composition of modes m and n is mode p.

$$\frac{}{\mathsf{V}\ ;\ n \Rightarrow n}\ \text{v}{*} \qquad \frac{}{\mathsf{W}\ ;\ \mathsf{V} \Rightarrow \mathsf{W}}\ \text{wv} \qquad \frac{}{\mathsf{R}\ ;\ n \Rightarrow \mathsf{R}}\ \text{r}{*}$$

$$\frac{}{\mathsf{W}\ ;\ \mathsf{R} \Rightarrow \mathsf{W}}\ \text{wr}$$

Note that in particular, the following mode compositions are *not derivable*:

- $\mathsf{W} \parallel \mathsf{W} \Rightarrow p$ is *not* derivable for any mode p
- $\mathsf{W}\ ;\ \mathsf{W} \Rightarrow p$ is *not* derivable for any mode p

Fig. 4. Syntax of modes; sequential and parallel mode composition.

$\boxed{\Delta; \Gamma \vdash e : C \rhd m}$ Under Δ and Γ, expression e has type C and mode m.

$$\frac{\Delta; \Gamma \vdash v : A}{\Delta; \Gamma \vdash \mathtt{ret}(v) : \mathbf{F}\, A \rhd \mathsf{V}}\ \text{ret} \qquad \frac{\begin{array}{c} m_1\ ;\ m_2 \Rightarrow m_3 \\ \Delta_1; \Gamma \vdash e_1 : \mathbf{F}\, A \rhd m_1 \\ \Delta_2, x : A; \Gamma \vdash e_2 : C \rhd m_2 \end{array}}{\Delta_1, \Delta_2; \Gamma, x : A \vdash \mathtt{let}(e_1, x.e_2) : C \rhd m_3}\ \text{let}$$

$$\frac{\cdot; \Gamma \vdash v : A}{\cdot; \Gamma \vdash \mathtt{ret}(v) : \mathbf{F}\,(\mathbf{!}A) \rhd \mathsf{V}}\ \text{ret!} \qquad \frac{\Delta_1; \Gamma \vdash v : \mathbf{!}A \qquad \Delta_2; \Gamma, x : A \vdash e : C \rhd m}{\Delta_1, \Delta_2; \Gamma, x : A \vdash \mathtt{let!}(v, x.e) : C \rhd m}\ \text{let!}$$

$$\frac{\Delta; \Gamma \vdash e : C \rhd m}{\Delta; \Gamma \vdash \lambda x.\, e : A \to C \rhd m}\ \text{lam} \qquad \frac{\Delta_1; \Gamma \vdash v : A \qquad \Delta_2; \Gamma \vdash e : A \to C \rhd m}{\Delta_1, \Delta_2; \Gamma \vdash e\, v : C \rhd m}\ \text{app}$$

$$\frac{\Delta, x : \big(\mathbf{Rd}\, A \times \mathbf{!}(\mathbf{Wr}\, A)\big); \Gamma \vdash e : C \rhd m}{\Delta; \Gamma \vdash \nu x.\, e : C \rhd m}\ \text{nu}$$

$$\frac{\Delta; \Gamma \vdash v : \mathbf{Rd}\, A}{\Delta \vdash \mathtt{rd}(v) : \mathbf{F}\,(A \times (\mathbf{Rd}\, A)) \rhd \mathsf{R}}\ \text{rd} \qquad \frac{\Delta_1; \Gamma \vdash v_1 : \mathbf{Wr}\, A \qquad \Delta_2; \Gamma \vdash v_2 : A}{\Delta_1, \Delta_2 \vdash \mathtt{wr}(v_1 \leftarrow v_2) : \mathbf{F}\, \mathsf{unit} \rhd \mathsf{W}}\ \text{wr}$$

$$\frac{\begin{array}{c} m_1 \parallel m_2 \Rightarrow m_3 \\ \Delta_1; \Gamma \vdash e_1 : C \rhd m_1 \\ \Delta_2; \Gamma \vdash e_2 : D \rhd m_2 \end{array}}{\Delta_1, \Delta_2 \vdash e_1 \mathrel{|\!\rhd} e_2 : D \rhd m_3}\ \text{fork} \qquad \frac{\begin{array}{c} \Delta_1; \Gamma \vdash e_1 : C \rhd \mathsf{R} \\ \Delta_2; \Gamma \vdash e_2 : C \rhd \mathsf{R} \end{array}}{\Delta_1, \Delta_2 \vdash e_1 \oplus e_2 : C \rhd \mathsf{R}}\ \text{choice}$$

$$
\begin{array}{lrcl}
\text{Channels} & \Sigma & ::= & \varepsilon \mid \Sigma, c \\
\text{Process pool} & \pi & ::= & \varepsilon \mid \pi, e \\
\text{Configurations} & C & ::= & \langle \Sigma; \pi \rangle \\
\text{Evaluation contexts} & E & ::= & \mathtt{let}(E, x.e) \\
 & & & \mid E\, v \\
 & & & \mid \bullet \\
\text{Read contexts} & R & ::= & \mathtt{rd}(\mathtt{chan}(c)) \oplus R \\
 & & & \mid R \oplus \mathtt{rd}(\mathtt{chan}(c)) \\
 & & & \mid \bullet
\end{array}
$$

$\boxed{e \longrightarrow e'}$ Expression e_1 reduces to e_2.

$$
\frac{}{\mathtt{let}(\mathtt{ret}(v), x.e) \longrightarrow [v/x]e}\ \text{let}
\qquad
\frac{}{(\lambda x.\, e)\, v \longrightarrow [v/x]e}\ \text{app}
\qquad
\frac{}{\mathtt{force}(\mathtt{thunk}(e)) \longrightarrow e}\ \text{force}
$$

$$
\frac{}{\mathtt{split}((v_1, v_2), x.y.e) \longrightarrow [v_1/x][v_2/y]e}\ \text{split}
\qquad
\frac{}{\mathtt{case}(\mathtt{inj}_i(v), x_1.e_1, x_2.e_2) \longrightarrow e_i[v/x_i]}\ \text{case}
$$

$\boxed{C_1 \equiv C_2}$ Configurations C_1 and C_2 are equivalent.

$$
\frac{\pi_1 \equiv_{\mathsf{perm}} \pi_2}{\langle \Sigma; \pi_1 \rangle \equiv \langle \Sigma; \pi_2 \rangle}\ \text{permProcs}
$$

$\boxed{C_1 \longrightarrow C_2}$ Configuration C_1 reduces to C_2.

$$
\frac{e \longrightarrow e'}{\langle \Sigma; \pi, E[e] \rangle \longrightarrow \left\langle \Sigma; \pi, E[e]' \right\rangle}\ \text{local}
\qquad
\frac{}{\langle \Sigma; \pi, E[e_1 \mid\!\triangleright\, e_2] \rangle \longrightarrow \langle \Sigma; \pi, e_1, E[e_2] \rangle}\ \text{fork}
$$

$$
\frac{C_1 \equiv C_1' \qquad C_1' \longrightarrow C_2 \qquad C_2 \equiv C_2'}{C_1 \longrightarrow C_2'}\ \text{congr}
$$

$$
\frac{c \notin \Sigma}{\langle \Sigma; \pi, E[\nu x.\, e] \rangle \longrightarrow \langle \Sigma, c; \pi, E[[(\mathtt{chan}(c), \mathtt{chan}(c))/x]e] \rangle}\ \text{nu}
$$

$$
\frac{}{\langle \Sigma; \pi, E_1[R[\mathtt{rd}(\mathtt{chan}(c))]], E_2[\mathtt{wr}(\mathtt{chan}(c) \leftarrow v)] \rangle \longrightarrow \langle \Sigma; \pi, E_1[v], E_2[()] \rangle}\ \text{rw}
$$