Universal Composability is Secure Compilation

Marco Patrignani
Stanford University
CISPA Helmholz Center for Information Security

Riad S. Wahby
Stanford University

Robert Künnemann
CISPA Helmholz Center for Information Security

Abstract

Universal composability is a framework for the specification and analysis of cryptographic protocols with a strong compositionality guarantee: UC protocols remain secure even when composed with other protocols. Secure compilation studies whether compiled programs are as secure as their source-level counterparts, no matter what target-level code they interact with. Although at present these disciplines are studied in isolation, we argue that there is a deep connection between them whose exploration will benefit both.

This paper outlines the connection between universal composability and robust compilation, the latest of secure compilation theories. We show how to read the universal composability theorem in terms of a robust compilation theorem, and vice-versa. This, in turn, shows which element of one theory corresponds to which element in the other. We believe this is the first step towards understanding how secure compilation theories can be used in universal composability settings, and vice-versa.

To better explain and clarify notions, this paper uses colours.
For a better experience, please print or view this paper in colour.

1 Introduction

Universal composability (UC) is a framework for the specification and analysis of cryptographic protocols with a key guarantee about compositionality [6, 7]. Several variations of UC exist [4, 11, 14, 17] but in this paper we focus on the original model of Canetti [6]. If a protocol is proven UC, that protocol behaves analogously to some high-level, secure-by-construction *ideal functionality* no matter what the protocol interacts with. As such, if that protocol is used within a larger protocol, in order to reason about the latter, we can replace the former protocol with its ideal functionality and just reason about the rest. In other words, UC protocols are secure even when composed with other protocols.

Secure compilation (*SC*) is a discipline that studies how to prove that a compiler from a source to a target language preserves the security properties of source programs in the target programs it produces. Several criteria for secure compilation have been proposed in the literature, ranging from relational, equivalence-based notions [1, 21] to more recent notions based on preserving traces [2, 19, 20].

While these two worlds seem to deal with quite different notions, we argue that they are deeply connected: both worlds are concerned with abstract notions (ideal functionalities, source programs) which are generally deemed secure, and more concrete notions (protocols, target programs) whose security must be proven *against arbitrary opponents*. What's more, proving that a protocol is UC, or that a compiler is secure, ensures that any attack at the concrete level (attacker, target program context) can be simulated at the abstract level (simulator, source program context).

We dig deeper into this analogy and show that there are benefits to be gained for both worlds by understanding their correspondence more deeply. After presenting UC and *SC*, this paper briefly outlines those benefits.

1.1 Universal Composability

We begin by briefly sketching how the UC framework is used. First, cryptographers craft a concrete cryptographic protocol ″ of interest, for example, a key-agreement and message transmission protocol like TLS [10]. In order to prove that ″ is UC (″ $\vdash_{\mathsf{UC}}$ F), they must come up with an ideal functionality F that (a) is secure by construction—in this example, a secret channel—and (b) such that ″ is at least as secure as F. What is key here is that the behaviour of ″ is given with respect to malicious attackers A that try to violate the cryptographic guarantees of ″. The ideal functionality, on the other hand, exists in a secure world where it interacts with "safe" attackers S that cannot break the ideal functionality. Roughly speaking, a UC proof demonstrates that it is possible to reduce any A to some S, effectively showing that any possible adversarial behaviour against ″ in the real world is also possible against F in the secure world. Because the secure world is secure by construction, the protocol must be secure in the real world.

The behaviour of protocols, ideal functionalities and attackers is observed by an *environment* Z that outputs a boolean value 0/1 representing some abstract observation.

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Intuitively, if Z's output is the same in both worlds, its observations of (″, A) and (F, S) match. However, since protocols deal with elements such as keys, which are guessable finite bitstrings, allowing all possible attackers would break any scheme. To sidestep this issue, Z's outputs in the two worlds must be the same except with very small probability (≈), and attackers must be polynomially bounded.

We can formalise UC as follows [6].

Definition 1.1 (UC (Informally)). $\text{″} \vdash_{\mathsf{UC}} \mathsf{F} \overset{\text{def}}{=} \forall \mathsf{A}, \exists \mathsf{S}, \forall \mathsf{Z}$ such that A is poly-bound. Then the diagram holds.

$$\begin{tikzcd}
 & 0/1 & & & & 0/1 & \\
 & \mathsf{Z} \arrow[u, Rightarrow] \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & & \approx & & \mathsf{Z} \arrow[u, Rightarrow] \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\text{″} \arrow[rr, leftrightarrow] & & \mathsf{A} & & \mathsf{F} \arrow[rr, leftrightarrow] & & \mathsf{S}
\end{tikzcd}$$

Bidirectional arrows (↔) represent the ability to communicate between two parties. ⊟

A key result of UC is that we can reason about protocols *compositionally*. Thus, if $\text{″}_1 \vdash_{\mathsf{UC}} \mathsf{F}_1$ and we also have a larger protocol ″_{big} that uses ″_1 inside ($\text{″}_{\mathsf{big}}[\text{″}_1]$), then we can just reason about the larger protocol using the ideal functionality instead ($\text{″}_{\mathsf{big}}[\mathsf{F}_1]$). This simplifies the reasoning process: the ″_1 sub-part is secure, since it behaves like F_1.

1.2 Robust Compilation

Recent developments in secure compilation have created a set of criteria that preserve classes of hyperproperties (read, arbitrary program behaviours [8]) dubbed *robust compilation* [2]. These criteria (like previous secure compilation criteria [1, 5, 19, 21]) are *robust*, i.e., they talk about *arbitrary* target-level attackers that compiled code is linked against. Our candidate from the robust compilation set is Robust Hyperproperty-Preserving Compilation (RHC), whose formal definition is given below. A compiler satisfying RHC produces compiled code that upholds the same hyperproperties as its source-level counterpart. Informally, a compiler satisfies RHC if, no matter what target-level program context ($\forall \mathbf{A}$) the compiled code ($[\![P]\!]$) is linked against ($\bowtie$)[1] the target behaviour ($\rightsquigarrow \bar{t}$) can also be reproduced ($\rightsquigarrow \bar{t}$) by the source program (P) linked with ($\bowtie$) a source-level program context ($\exists A$). Formally, RHC is defined as follows:

Definition 1.2 (RHC). $[\![\cdot]\!] \vdash \mathsf{RHC} \overset{\text{def}}{=} \forall P, \mathbf{A}.\ \exists A.\ \forall \bar{t}.$
$\mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \bar{t} \iff A \bowtie P \rightsquigarrow \bar{t}$ ⊟

To make the connection with UC clearer, below is a diagrammatic representation of RHC. We invite the reader to compare it with the UC diagram presented before.

$$\begin{tikzcd}
 & \bar{t} & & & & \bar{t} & \\
{[\![P]\!]} & \bowtie \arrow[u, rightsquigarrow] & \mathbf{A} & \Longleftrightarrow & P & \bowtie \arrow[u, rightsquigarrow] & A
\end{tikzcd}$$

1.3 A Bridge Between Two Worlds

At this point, we start to see a connection between the two worlds: it looks like all elements from each system exist in the other. We capture this intuition in the table below.

UC		*SC*	
protocol	″	$[\![P]\!]$	compiled program
concrete attacker	A	$\mathbf{A}$	target context
ideal functionality	F	P	source program
simulator	S	A	source context
environment, output	Z, 0/1	$\bar{t}$, $\rightsquigarrow$	trace, semantics
communication	↔	$\bowtie$	linking
probabilistic equiv.	≈	$\iff$	trace equality
human translation	$\text{″} \to \mathsf{F}$	$[\![\cdot]\!]: P \to \mathbf{P}$	compiler

While most of this table should, at this point, not be surprising, we want to focus on the final line, since our primary insights revolve around it.

While decades of work has yielded automated tools to generate binaries for our computers in the form of compilers, the same is far from true for cryptographic protocols—and devising ideal functionalities for complex cryptographic protocols *by hand* is a tedious and error-prone process. But the above analogy suggests that secure compilation may point the way towards *generating* concrete cryptographic protocols from high-level specifications, much like binaries are created from high-level programming languages. This would have a plethora of benefits, a few of which we list below. First, secure compilation for cryptography would open the development of new cryptographic protocols to a broad audience. Second, having compilers for cryptographic protocols would let us draw upon years of knowledge in proof mechanisation, both to mechanise UC proofs and to automate the secure implementation of cryptographic protocols (along the lines of CompCert [16] and CakeML [13]).

Finally, UC proofs are often very complex, and that complexity is in fact a major hurdle for widespread adoption of the framework. For example, it took years and several unsuccessful attempts to prove results for cryptographic primitives as seemingly basic as digital signatures [3] and symmetric encryption [15] (a particular challenge is the definition of polynomial runtime bounds [12]). Fortunately, recent advances in proving compilers secure have given us well-understood proof techniques called *backtranslations* [2, 9, 18, 20]—and we believe these techniques can be employed to rigorously and automatically generate UC proofs.

On the other side, reasoning about composition for securely-compiled programs may lead to new insights for *SC*. Consider some securely-compiled code $[\![P1]\!]^{A}$ linked with other securely compiled code $[\![P2]\!]^{B}$. What this entails at the level

[1] We do not use the more conventional programming-language notation for plugging a program in a context, namely $A[P]$, to draw a neater analogy with UC. Effectively, these two notations are equivalent: $A[P]$ and $A \bowtie P$.

of the resulting program $[\![P1]\!]^{\mathbf{A}} \bowtie [\![P2]\!]^{\mathbf{B}}$ is unknown, because each compiled program may be proven secure in the sense of preserving a distinct class of hyperproperties. We believe that insights from the UC world will help us reason about securely-compiled program composition.

Acknowledgements: This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] Martín Abadi. Protection in programming-language translations. In *ICALP’98*, pages 868–883, 1998.

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32th Computer Security Foundations Symposium*, CSF 2019, June 2019.

[3] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In *International Conference on Information Security*, pages 61–72. Springer, 2004.

[4] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007. doi: 10.1016/j.ic.2007.05.002. URL https://doi.org/10.1016/j.ic.2007.05.002.

[5] William J. Bowman and Amal Ahmed. Noninterference for free. In *ICFP*. ACM, 2015.

[6] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS ’01, pages 136–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1390-5. URL http://dl.acm.org/citation.cfm?id=874063.875553.

[7] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. Cryptology ePrint Archive, Report 2014/553, 2014. https://eprint.iacr.org/2014/553.

[8] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://www.cs.cornell.edu/~clarkson/papers/clarkson_hyperproperties_journal.pdf.

[9] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science*, Volume 13, Issue 4, October 2017.

[10] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of tls. In *International Conference on Provable Security*, pages 313–327. Springer, 2008.

[11] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. Cryptology ePrint Archive, 2011. URL http://eprint.iacr.org/.

[12] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. Polynomial runtime and composability. *Journal of Cryptology*, 26(3):375–441, 2013.

[13] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–192. ACM, 2014. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL https://cakeml.org/popl14.pdf.

[14] Ralf Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Computer Security Foundations Workshop*, pages 309–320. IEEE Computer Society, 2006.

[15] Ralf Küsters and Max Tuengerthal. Universally composable symmetric encryption. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 293–307. IEEE, 2009.

[16] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. URL http://dx.doi.org/10.1007/s10817-009-9155-4.

[17] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011. ISBN 978-3-642-27374-2. doi: 10.1007/978-3-642-27375-9_3. URL https://doi.org/10.1007/978-3-642-27375-9_3.

[18] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming*, pages 103–116. ACM, 2016.

[19] Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperties Preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium CSF 2017, Santa Barbara, USA*, CSF 2017, 2017.

[20] Marco Patrignani and Deepak Garg. Robustly safe compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*, ESOP'19, 2019.

[21] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, January 2019.